

# 新年开篇：如何减少react组件不必要的重新渲染



## 什么是重新渲染

react重新渲染是当你的组件已经渲染在屏幕上，由于数据状态变化，又重新触发了组件的渲染。重新渲染是正常现象，但有时由于自身代码问题造成不必要的重新渲染。



## 不必要的重新渲染

由于错误的代码引发组件的重新渲染。例如：与react组件本身无关的状态变化引起的组件的重新渲染。虽然react组件重新渲染了，但由于渲染很快，通常用户并不会感知到。但如果重新渲染发生比较复杂的组件上，可能会导致界面卡顿，甚至造成长时间的卡死现象。



## react组件重新渲染情况

组件的重新渲染分为以下几种情况：

### state变化

count变化会引起组件的重新渲染。

```
const App = () => {  
  const [count, setCount] = useState(0);  
  
  const handleClick = () => {  
    setCount(count+1)  
  }  
  
  return <div onClick={handleClick}>{count}</div>  
}
```

## 父组件的重新渲染

当组件的父组件重新渲染时，该组件必定重新渲染。一般情况下子组件的渲染不会触发父组件的渲染。

实际上**props变化**也是由于父组件状态变化引起的，只要父组件重新渲染，子组件不管props有没有变化都会重新渲染。（没有使用优化时）

```
import { useState } from "react";  
  
const Child = () => {  
  console.log("子组件重新渲染");  
  const [childCount, setChildCount] = useState(0);  
  
  const handleChildClick = () => {  
    setChildCount(childCount + 1);  
  };  
  
  return (  
    <>  
      <div onClick={handleChildClick}>Child {childCount}</div>  
    </>  
  );  
};  
  
export default function StateChange() {  
  const [count, setCount] = useState(0);  
  
  const handleClick = () => {  
    setCount(count + 1);  
  };  
  
  console.log("父组件重新渲染", count);  
}
```

```

return (
  <div>
    <div onClick={handleClick}>parent {count}</div>
    <Child />
  </div>
);
}

```

## context变化

当context的Provider提供的值放生变化，所有使用该context的组件都将重新渲染。

javascript 复制代码

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

```
const Context = createContext({ val: 0 });
```

```
const Provider = ({ children }) => {
  const [val, setVal] = useState(0);
  const handleClick = () => {
    setVal(val + 1);
  };
};
```

```
const value = useMemo(() => {
  return {
    val: val
  };
}, [val]);
```

```
return (
  <Context.Provider value={value}>
    {children}
    <button onClick={handleClick}>context change</button>
  </Context.Provider>
);
};
```

```
const useVal = () => useContext(Context);
```

```
const Child1 = () => {
  const { val } = useVal();
  console.log("Child1重新渲染", val);

  return <div>Child1</div>;
};
```

```
const Child2 = () => {
```

```

const { val } = useVal();
console.log("Child2重新渲染", val);
return <div>Child2</div>;
};

export default function ContextChange() {
  return (
    <Provider>
      <Child1 />
      <Child2 />
    </Provider>
  );
}

```

Child1  
Child2

context change

Console 0	Problems 2	React DevTools 0	Filter
console was created			
Child1重新渲染	0		
Child2重新渲染	0		
Child1重新渲染	1		
Child2重新渲染	1		@稀土掘金技术社区

## 在组件内创建组件

✗：这种做法非常消耗性能。当组件重新渲染，内部组件都会重新mount。这种做法容易导致很多bug出现。

```

import { useState, useEffect } from "react";

const Component = () => {
  const [state, setState] = useState(1);

  const onClick = () => {
    setState(state + 1);
  };
};

```

javascript 复制代码

```

const InComponent = () => {
  console.log("内部组件重新渲染");
  useEffect(() => {
    console.log("内部组件重新mount");
  }, []);
  return <div>inComponent</div>;
};

return (
  <>
    <button onClick={onClick}>点我</button>
    <InComponent />
  </>
);
};

export default function ComInComponent() {
  return (
    <>
      <Component />
    </>
  );
}

```



## 一些减少重新渲染的方法

### useState初始值使用函数形式

看一个例子：useState的初始值经过 $a + 1$ 计算得到: 此时的useState的参数是一个函数执行，也就是一个值

```

import { useState } from "react";

function getInitState() {
  console.count("获取初始化的值");
  const a = 1;
  return a + 1;
}

```

ini 复制代码

```
const App = () => {  
  const [value, setValue] = useState(getInitState());  
  const onChange = (event) => setValue(event.target.value);  
  
  return <input type="text" value={value} onChange={onChange} />;  
};  
  
export default App;
```

当我们在input中继续输入，可以看到getInitState的console次数：



当我们把useState第一个参数改为函数时：（该函数会自动调用）

```
import { useState } from "react";  
  
function getInitState() {  
  console.count("获取初始化的值");  
  const a = 1;
```

ini 复制代码

```
    return a + 1;
  }

  const App = () => {
    const [value, setValue] = useState(getInitState);
    const onChange = (event) => setValue(event.target.value);

    return <input type="text" value={value} onChange={onChange} />;
  };

  export default App;
```

该函数只会在初始化的时候调用一次：



## 重新组织组件结构

提取单独的组件，独自维护自己的状态，防止发生不必要的渲染。

```
import { useState } from "react";

const BigComponent = () => {
```

javascript 复制代码

```

    console.log("一个非常复杂的大组件：渲染了");
    return <div>BigComponent</div>;
  };

const AllComponent = () => {
  const [state, setState] = useState(1);

  const onClick = () => {
    setState(state + 1);
  };

  return (
    <>
      <p>重新渲染次数: {state}</p>
      <button onClick={onClick}>点我</button>
      <BigComponent />
    </>
  );
};

const ButtonComponent = () => {
  const [state, setState] = useState(1);

  const onClick = () => {
    setState(state + 1);
  };

  return (
    <>
      <p>重新渲染次数: {state}</p>
      <button onClick={onClick}>点我</button>
    </>
  );
};

const SplitComponent = () => {
  return (
    <>
      <ButtonComponent />
      <BigComponent />
    </>
  );
};

const App = () => {
  return (
    <>
      <p>AllComponent是没有重新组织的组件</p>
      <AllComponent />
    </>
  );
};

```



```

    <hr />
    <p>SplitComponent是重新组织划分的组件，不会触发大组件BigComponent的渲染</p>
    <SplitComponent />
  </>
);
};

export default App;

```

## 巧用props.children

利用props.children 来减少不必要的重复渲染。

javascript 复制代码

```

import { useState } from "react";

const BigComponent = () => {
  console.log("一个非常复杂的大组件：渲染了");
  return <div>BigComponent</div>;
};

const AllComponent = () => {
  const [state, setState] = useState(1);

  const onClick = () => {
    setState(state + 1);
  };

  return (
    <>
      <p>重新渲染次数: {state}</p>
      <button onClick={onClick}>点我</button>
      <BigComponent />
    </>
  );
};

const ComponentWithChildren = ({ children }) => {
  const [state, setState] = useState(1);

  const onClick = () => {
    setState(state + 1);
  };

  return (
    <>
      <p>重新渲染次数: {state}</p>

```

```

        <button onClick={onClick}>点我</button>
      {children}
    </>
  );
};

const SplitComponent = () => {
  return (
    <ComponentWithChildren>
      <BigComponent />
    </ComponentWithChildren>
  );
};

const App = () => {
  return (
    <>
      <p>AllComponent是没有重新组织的组件</p>
      <AllComponent />
      <hr />
      <p>
        SplitComponent是巧用了children的组件，不会触发大组件BigComponent的渲染
      </p>
      <SplitComponent />
    </>
  );
};

export default App;

```

## 把组件当成props传递

把组件当成props传递给其他组件，也可以减少渲染。(与children类似)

```

import { useState } from "react";

const BigComponent = () => {
  console.log("一个非常复杂的大组件：渲染了");
  return <div>BigComponent</div>;
};

const AllComponent = () => {
  const [state, setState] = useState(1);

  const onClick = () => {
    setState(state + 1);
  };
};

```

javascript 复制代码

```

};

return (
  <>
    <p>重新渲染次数: {state}</p>
    <button onClick={onClick}>点我</button>
    <BigComponent />
  </>
);
};

const ComponentWithProps = ({ comp }) => {
  const [state, setState] = useState(1);

  const onClick = () => {
    setState(state + 1);
  };

  return (
    <>
      <p>重新渲染次数: {state}</p>
      <button onClick={onClick}>点我</button>
      {comp}
    </>
  );
};

const comp = <BigComponent />;

const SplitComponent = () => {
  return (
    <>
      <ComponentWithProps comp={comp} />
    </>
  );
};

const App = () => {
  return (
    <>
      <p>AllComponent是没有重新组织的组件</p>
      <AllComponent />
      <hr />
      <p>
        SplitComponent是把组件当成props传递，不会触发大组件BigComponent的渲染
      </p>
      <SplitComponent />
    </>
  );
};

```

```
};

export default App;
```

## React.memo

memo 减少重复渲染，只要子组件的props没有改变（浅比较）。

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

const Child = () => {
  console.log("子组件重新渲染");

  return (
    <>
      <div>Child</div>
    </>
  );
};

const MemoChild = memo(Child);

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

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

  return (
    <div>
      <div onClick={handleClick}>parent {count}</div>
      <MemoChild />
    </div>
  );
}
```

javascript 复制代码

如果props是一个object， array 或者 function， memo必须配合[useMemo来缓存](#)，单独使用都不起作用。

当然也可以单独使用useMemo缓存整个组件。

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

javascript 复制代码

```

const Child = ({ value }) => {
  console.log("Child重新渲染", value.value);
  return <>{value.value}</>;
};

const values = [1, 2, 3];

const UseMemp = () => {
  const [state, setState] = useState(1);

  const onClick = () => {
    setState(state + 1);
  };

  // 使用useMemo缓存组件
  const items = useMemo(() => {
    return values.map((val) => <Child key={val} value={{ value: val }} />);
  }, []);

  return (
    <>
      <button onClick={onClick}>点我 {state}</button>
      <br />
      {items}
    </>
  );
};

export default UseMemp;

```

## useCallback

函数传递可以使用useCallback来缓存函数。

## key值

在循环数组时，有人使用index作为key，可以这样做，但是你要保证你的数组是静态的，没有新增，删除，插入，排序等情况，否则不能使用index作为key。



**完整的代码：**

1. [codesandbox.io/s/hidden-me...](https://codesandbox.io/s/hidden-me...)
2. [mp.weixin.qq.com/s/wJwHbPf7\\_...](https://mp.weixin.qq.com/s/wJwHbPf7_...)



## 参考

1. [reactjs.org/docs/hooks-...](https://reactjs.org/docs/hooks-...)
2. [medium.com/@guptagarud...](https://medium.com/@guptagarud...)