

React 组件性能优化最佳实践

React 组件性能优化的核心是减少渲染真实 DOM 节点的频率，减少 Virtual DOM 比对的频率。

1. 组件卸载前进行清理操作

在组件中为 window 注册的全局事件，以及定时器，在组件卸载前要清理掉，防止组件卸载后继续执行影响应用性能。

需求：开启定时器然后卸载组件，查看组件中的定时器是否还在运行。

```
import React, { useState, useEffect } from "react"
import ReactDOM from "react-dom"

const App = () => {
  let [index, setIndex] = useState(0)
  useEffect(() => {
    let timer = setInterval(() => {
      setIndex(prev => prev + 1)
      console.log('timer is running...')
    }, 1000)
    return () => clearInterval(timer)
  }, [])
  return (
    <button onClick={() => ReactDOM.unmountComponentAtNode(document.getElementById("root"))}>
      {index}
    </button>
  )
}

export default App
```

2. PureComponent

1. 什么是纯组件

纯组件会对组件输入数据进行浅层比较，如果当前输入数据和上次输入数据相同，组件不会重新渲染。

2. 什么是浅层比较

比较引用数据类型在内存中的引用地址是否相同，比较基本数据类型的值是否相同。

3. 如何实现纯组件

类组件继承 PureComponent 类，函数组件使用 memo 方法

4. 为什么不直接进行 diff 操作，而是要先进行浅层比较，浅层比较难道没有性能消耗吗

和进行 diff 比较操作相比，浅层比较将消耗更少的性能。diff 操作会重新遍历整颗 virtualDOM 树，而浅层比较只操作当前组件的 state 和 props。

5. 需求：在状态对象中存储 name 值为张三，组件挂载完成后将 name 属性的值再次更改为张三，然后分别将 name 传递给纯组件和非纯组件，查看结果。

```

import React from "react"
export default class App extends React.Component {
  constructor() {
    super()
    this.state = {name: "张三"}
  }
  updateName() {
    setInterval(() => this.setState({name: "张三"}), 1000)
  }
  componentDidMount() {
    this.updateName()
  }
  render() {
    return (
      <div>
        <RegularComponent name={this.state.name} />
        <PureChildComponent name={this.state.name} />
      </div>
    )
  }
}

class RegularComponent extends React.Component {
  render() {
    console.log("RegularComponent")
    return <div>{this.props.name}</div>
  }
}

class PureChildComponent extends React.PureComponent {
  render() {
    console.log("PureChildComponent")
    return <div>{this.props.name}</div>
  }
}

```

3. shouldComponentUpdate

纯组件只能进行浅层比较，要进行深层比较，使用 `shouldComponentUpdate`，它用于编写自定义比较逻辑。

返回 `true` 重新渲染组件，返回 `false` 阻止重新渲染。

函数的第一个参数为 `nextProps`，第二个参数为 `nextState`。

需求：在页面中展示员工信息，员工信息包括，姓名，年龄，职位。但是在页面中只想展示姓名和年龄。也就是说只有姓名和年龄发生变化时才有必要重新渲染组件，如果员工的其他信息发生了变化没必要重新渲染组件。

```
import React from "react"

export default class App extends React.Component {
  constructor() {
    super()
    this.state = {name: "张三", age: 20, job: "waiter"}
  }
  componentDidMount() {
    setTimeout(() => this.setState({ job: "chef" }), 1000)
  }

  shouldComponentUpdate(nextProps, nextState) {
    if (this.state.name !== nextState.name || this.state.age !== nextState.age) {
      return true
    }
    return false
  }

  render() {
    console.log("rendering")
    let { name, age } = this.state
    return <div>{name} {age}</div>
  }
}
```

4. React.memo

1. memo 基本使用

将函数组件变为纯组件，将当前 props 和上一次的 props 进行浅层比较，如果相同就阻止组件重新渲染。

需求：父组件维护两个状态，index 和 name，开启定时器让 index 不断发生变化，name 传递给子组件，查看父组件更新子组件是否也更新了。

```

import React, { memo, useEffect, useState } from "react"

function ShowName({ name }) {
  console.log("showName render...")
  return <div>{name}</div>
}

const ShowNameMemo = memo(ShowName)

function App() {
  const [index, setIndex] = useState(0)
  const [name] = useState("张三")
  useEffect(() => {
    setInterval(() => {
      setIndex(prev => prev + 1)
    }, 1000)
  }, [])
  return (
    <div>
      {index}
      <ShowNameMemo name={name} />
    </div>
  )
}

export default App

```

2. 为 memo 传递比较逻辑

使用 memo 方法自定义比较逻辑，用于执行深层比较。

比较函数的第一个参数为上一次的 props，比较函数的第二个参数为下一次的 props，比较函数返回 true，不进行渲染，比较函数返回 false，组件重新渲染。

```

function App() {
  const [person, setPerson] = useState({ name: "张三", age: 20, job: "waiter" })
  return <>
    <ShowPerson person={person} />
    <button onClick={() => setPerson({...person, job: "chef"})}>button</button>
  </>
}

export default App

function ShowPerson({ person }) {
  console.log("ShowPerson render...")
  return (
    <div>
      {person.name} {person.age}
    </div>
  )
}

```

```

import React, { memo, useEffect, useState } from "react"

const ShowPersonMemo = memo(ShowPerson, comparePerson)

function comparePerson(prevProps, nextProps) {
  if (
    prevProps.person.name !== nextProps.person.name ||
    prevProps.person.age !== nextProps.person.age
  ) {
    return false
  }
  return true
}

function App() {
  const [person, setPerson] = useState({ name: "张三", age: 20, job: "waiter" })
  return <>
    <ShowPersonMemo person={person} />
    <button onClick={() => setPerson({...person, job: "chef"})}>button</button>
  </>
}
export default App

```

5. 使用组件懒加载

使用组件懒加载可以减少 bundle 文件大小, 加快组件呈递速度.

1. 路由组件懒加载

```

import React, { lazy, Suspense } from "react"
import { BrowserRouter, Link, Route, Switch } from "react-router-dom"

const Home = lazy(() => import(/* webpackChunkName: "Home" */ "./Home"))
const List = lazy(() => import(/* webpackChunkName: "List" */ "./List"))

function App() {
  return (
    <BrowserRouter>
      <Link to="/">Home</Link>
      <Link to="/list">List</Link>
      <Switch>
        <Suspense fallback={<div>Loading</div>}>
          <Route path="/" component={Home} exact />
          <Route path="/list" component={List} />
        </Suspense>
      </Switch>
    </BrowserRouter>
  )
}

export default App

```

2. 根据条件进行组件懒加载

适用于组件不会随条件频繁切换

```
import React, { lazy, Suspense } from "react"

function App() {
  let LazyComponent = null
  if (true) {
    LazyComponent = lazy(() => import(/* webpackChunkName: "Home" */ "../Home"))
  } else {
    LazyComponent = lazy(() => import(/* webpackChunkName: "List" */ "../List"))
  }
  return (
    <Suspense fallback={<div>Loading</div>}>
      <LazyComponent />
    </Suspense>
  )
}

export default App
```

6. 使用 Fragment 避免额外标记

React 组件中返回的 jsx 如果有多个同级元素, 多个同级元素必须要有一个共同的父级.

```
function App() {
  return (
    <div>
      <div>message a</div>
      <div>message b</div>
    </div>
  )
}
```

为了满足这个条件我们通常都会在最外层添加一个div, 但是这样的话就会多出一个无意义的标记, 如果每个组件都多出这样的无意义标记的话, 浏览器渲染引擎的负担就会加剧.

为了解决这个问题, React 推出了 fragment 占位符标记. 使用占位符标记既满足了拥有共同父级的要求又不会多出额外的无意义标记.

```
import { Fragment } from "react"

function App() {
  return (
    <Fragment>
      <div>message a</div>
      <div>message b</div>
    </Fragment>
  )
}
```

```
function App() {  
  return (  
    <>  
      <div>message a</div>  
      <div>message b</div>  
    </>  
  )  
}
```

7. 不要使用内联函数定义

在使用内联函数后, render 方法每次运行时都会创建该函数的新实例, 导致 React 在进行 Virtual DOM 比对时, 新旧函数比对不相等, 导致 React 总是为元素绑定新的函数实例, 而旧的函数实例又要交给垃圾回收器处理.

```
import React from "react"  
  
export default class App extends React.Component {  
  constructor() {  
    super()  
    this.state = {  
      inputValue: ""  
    }  
  }  
  render() {  
    return (  
      <input  
        value={this.state.inputValue}  
        onChange={e => this.setState({ inputValue: e.target.value })}  
      />  
    )  
  }  
}
```

正确的做法是在组件中单独定义函数, 将函数绑定给事件.

```
import React from "react"

export default class App extends React.Component {
  constructor() {
    super()
    this.state = {
      inputValue: ""
    }
  }
  setInputValue = e => {
    this.setState({ inputValue: e.target.value })
  }
  render() {
    return (
      <input value={this.state.inputValue} onChange={this.setInputValue} />
    )
  }
}
```

8. 在构造函数中进行函数this绑定

在类组件中如果使用 `fn() {}` 这种方式定义函数, 函数 `this` 默认指向 `undefined`. 也就是说函数内部的 `this` 指向需要被更正.

可以在构造函数中对函数的 `this` 进行更正, 也可以在行内进行更正, 两者看起来没有太大区别, 但是对性能的影响是不同的.

```
export default class App extends React.Component {
  constructor() {
    super()
    // 方式一
    // 构造函数只执行一次, 所以函数 this 指向更正的代码也只执行一次.
    this.handleClick = this.handleClick.bind(this)
  }
  handleClick() {
    console.log(this)
  }
  render() {
    // 方式二
    // 问题: render 方法每次执行时都会调用 bind 方法生成新的函数实例.
    return <button onClick={this.handleClick.bind(this)}>按钮</button>
  }
}
```

9. 类组件中的箭头函数

在类组件中使用箭头函数不会存在 `this` 指向问题, 因为箭头函数本身并不绑定 `this`.


```
export default class App extends React.Component {  
  handleClick = () => console.log(this)  
  render() {  
    return <button onClick={this.handleClick}>按钮</button>  
  }  
}
```

箭头函数在 this 指向问题上占据优势, 但是同时也有不利的一面.

当使用箭头函数时, 该函数被添加为类的实例对象属性, 而不是原型对象属性. 如果组件被多次重用, 每个组件实例对象中都将会有一个相同的函数实例, 降低了函数实例的可重用性造成了资源浪费.

综上所述, 更正函数内部 this 指向的最佳做法仍是在构造函数中使用 bind 方法进行绑定

10. 避免使用内联样式属性

当使用内联 style 为元素添加样式时, 内联 style 会被编译为 JavaScript 代码, 通过 JavaScript 代码将样式规则映射到元素的身上, 浏览器就会花费更多的时间执行脚本和渲染 UI, 从而增加了组件的渲染时间.

```
function App() {  
  return <div style={{ backgroundColor: "skyblue" }}>App works</div>  
}
```

在上面的组件中, 为元素附加了内联样式, 添加的内联样式为 JavaScript 对象, backgroundColor 需要被转换为等效的 CSS 样式规则, 然后将其应用到元素, 这样涉及到脚本的执行.

更好的办法是将 CSS 文件导入样式组件. 能通过 CSS 直接做的事情就不要通过 JavaScript 去做, 因为 JavaScript 操作 DOM 非常慢.

11. 优化条件渲染

频繁的挂载和卸载组件是一项耗性能的操作, 为了确保应用程序的性能, 应该减少组件挂载和卸载的次数.

在 React 中我们经常会根据条件渲染不同的组件. 条件渲染是一项必做的优化操作.

```
function App() {
  if (true) {
    return (
      <>
        <AdminHeader />
        <Header />
        <Content />
      </>
    )
  } else {
    return (
      <>
        <Header />
        <Content />
      </>
    )
  }
}
```

在上面的代码中, 当渲染条件发生变化时, React 内部在做 Virtual DOM 比对时发现, 刚刚第一个组件是 AdminHeader, 现在第一个组件是 Header, 刚刚第二个组件是 Header, 现在第二个组件是 Content, 组件发生了变化, React 就会卸载 AdminHeader、Header、Content, 重新挂载 Header 和 Content, 这种挂载和卸载就是没有必要的.

```
function App() {
  return (
    <>
      {true && <AdminHeader />}
      <Header />
      <Content />
    </>
  )
}
```

12. 避免重复无限渲染

当应用程序状态发生更改时, React 会调用 render 方法, 如果在 render 方法中继续更改应用程序状态, 就会发生 render 方法递归调用导致应用报错.

Error: Maximum update depth exceeded. This can happen when a component repeatedly calls setState inside componentWillUpdate or componentDidUpdate. React limits the number of nested updates to prevent infinite loops.

```
export default class App extends React.Component {
  constructor() {
    super()
    this.state = {name: "张三"}
  }
  render() {
    this.setState({name: "李四"})
    return <div>{this.state.name}</div>
  }
}
```

与其他生命周期函数不同, render 方法应该被作为纯函数. 这意味着, 在 render 方法中不要做以下事情, 比如不要调用 setState 方法, 不要使用其他手段查询更改原生 DOM 元素, 以及其他更改应用程序的任何操作. render 方法的执行要根据状态的改变, 这样可以保持组件的行为和渲染方式一致.

13. 为组件创建错误边界

默认情况下, 组件渲染错误会导致整个应用程序中断, 创建错误边界可确保在特定组件发生错误时应用程序不会中断.

错误边界是一个 React 组件, 可以捕获子级组件在渲染时发生的错误, 当错误发生时, 可以将错误记录下来, 可以显示备用 UI 界面.

错误边界涉及到两个生命周期函数, 分别为 `getDerivedStateFromError` 和 `componentDidCatch`.

`getDerivedStateFromError` 为静态方法, 方法中需要返回一个对象, 该对象会和state对象进行合并, 用于更改应用程序状态.

`componentDidCatch` 方法用于记录应用程序错误信息. 该方法的参数就是错误对象.

```
// ErrorBoundaries.js
import React from "react"
import App from "./App"

export default class ErrorBoundaries extends React.Component {
  constructor() {
    super()
    this.state = {
      hasError: false
    }
  }
  componentDidCatch(error) {
    console.log("componentDidCatch")
  }
  static getDerivedStateFromError() {
    console.log("getDerivedStateFromError")
    return {
      hasError: true
    }
  }
  render() {
    if (this.state.hasError) {
      return <div>发生了错误</div>
    }
    return <App />
  }
}

// App.js
import React from "react"

export default class App extends React.Component {
  render() {
    // throw new Error("lalala")
    return <div>App works</div>
  }
}

// index.js
import React from "react"
import ReactDOM from "react-dom"
import ErrorBoundaries from "./ErrorBoundaries"

ReactDOM.render(<ErrorBoundaries />, document.getElementById("root"))
```

注意: 错误边界不能捕获异步错误, 比如点击按钮时发生的错误.

14. 避免数据结构突变

组件中 props 和 state 的数据结构应该保持一致, 数据结构突变会导致输出不一致.

```
import React, { Component } from "react"

export default class App extends Component {
  constructor() {
    super()
    this.state = {
      employee: {
        name: "张三",
        age: 20
      }
    }
  }
  render() {
    const { name, age } = this.state.employee
    return (
      <div>
        {name}
        {age}
        <button
          onClick={() =>
            this.setState({
              ...this.state,
              employee: {
                ...this.state.employee,
                age: 30
              }
            })
          }
        >
          change age
        </button>
      </div>
    )
  }
}
```

15. 依赖优化

在应用程序中经常会依赖第三方包, 但我们不想引用包中的所有代码, 我们只想用到哪些代码就包含哪些代码. 此时可以使用插件对依赖项进行优化. [优化资源](#)

当前我们就使用 lodash 举例子. 应用基于 create-react-app 脚手架创建。

1. 下载依赖 yarn add react-app-rewired customize-cra lodash babel-plugin-lodash

1. react-app-rewired: 覆盖 create-react-app 的默认配置

```
module.exports = function (oldConfig) {
  return newConfig
}
// 参数中的 oldConfig 就是默认的 webpack config
```

2. customize-cra: 导出了一些辅助方法, 可以让以上写法更加简洁

```
const { override, useBabelRc } = require("customize-cra")

module.exports = override(
  (oldConfig) => newConfig,
  (oldConfig) => newConfig
)
```

override: 可以接收多个参数, 每个参数都是一个配置函数, 函数接收 oldConfig, 返回 newConfig

useBabelRc: 允许使用 .babelrc 文件进行 babel配置

3. babel-plugin-lodash: 对应用中的 lodash 进行精简

2. 在项目的根目录下新建 config-overrides.js 并加入配置代码

```
const { override, useBabelRc } = require("customize-cra")

module.exports = override(useBabelRc())
```

3. 修改 package.json 文件中的构建命令

```
"scripts": {
  "start": "react-app-rewired start",
  "build": "react-app-rewired build",
  "test": "react-app-rewired test --env=jsdom",
  "eject": "react-scripts eject"
}
```

4. 创建 .babelrc 文件并加入配置

```
{
  "plugins": ["lodash"]
}
```

5. 生产环境下的三种 JS 文件

1. main.[hash].chunk.js: 这是你的应用程序代码, App.js 等.
2. 1.[hash].chunk.js: 这是第三方库的代码, 包含你在 node_modules 中导入的模块
3. runtime~main.[hash].js webpack运行时代码


41.22 KB	build/static/js/2.9742c8ff.chunk.js	64.96 KB	build/static/js/2.4f6c9b73.chunk.js
787 B	build/static/js/runtime-main.db1ce76c.js	787 B	build/static/js/runtime-main.db1ce76c.js
280 B	build/static/js/main.846e031a.chunk.js	309 B	build/static/js/main.f9d109c9.chunk.js
46.83 KB	build/static/js/2.b7a5538f.chunk.js		
787 B	build/static/js/runtime-main.db1ce76c.js		
313 B	build/static/js/main.cf812781.chunk.js		

6. App 组件

```
import React from "react"
import _ from "lodash"

function App() {
  console.log(_.chunk(["a", "b", "c", "d"], 2))
  return <div>App works</div>
}

export default App
```

 联系方式