22 | 思路拓展：如何打造高性能的 React 应用?

2020/12/23 修言

React 应用也是前端应用，如果之前你知道一些前端项目普适的性能优化手段，比如资源加载过程中的优化、减少重绘与回流、服务端渲染、启用 CDN 等，那么这些手段对于 React 来说也是同样奏效的。

不过对于 React 项目来说，它有一个区别于传统前端项目的重要特点，就是**以 React 组件的形式来组织逻辑**：组件允许我们将 UI 拆分为独立可复用的代码片段，并对每个片段进行独立构思。因此，除了前面所提到的普适的前端性能优化手段之外，React 还有一些充满了自身特色的性能优化思路，这些思路基本都围绕“组件性能优化”这个中心思想展开。本讲我将带你认识其中最关键的 3 个思路：

1. **使用 shouldComponentUpdate 规避冗余的更新逻辑**
2. **PureComponent + Immutable.js**
3. **React.memo 与 useMemo**

> 注：**这 3 个思路同时也是 React 面试中“性能优化”这一环的核心所在**。大家在回答类似题目的时候，不管其他的细枝末节的优化策略能不能想起来，以上三点一定要尽量答全。

朴素思路：善用 shouldComponentUpdate

shouldComponentUpdate 是 React 类组件的一个生命周期。关于 shouldComponentUpdate 是什么，我们已经在第 02 讲有过介绍，这里先简单复习一下。

shouldComponentUpdate 的调用形式如下：

```
1. shouldComponentUpdate(nextProps, nextState)
```

复制代码

render 方法由于伴随着对虚拟 DOM 的构建和对比，过程可以说相当耗时。而在 React 当中，很多时候我们会不经意间就频繁地调用了 render。为了避免不必要的 render 操作带来的性能开销，React 提供了 shouldComponentUpdate 这个口子。**React 组件会根据 shouldComponentUpdate 的返回值，来决定是否执行该方法之后的生命周期，进而决定是否对组件进行 re-render（重渲染）**。

shouldComponentUpdate 的默认值为 true，也就是说 **“无条件 re-render”**。在实际的开发中，我们往往通过手动往 shouldComponentUpdate 中填充判定逻辑，来实现“有条件的 re-render”。

接下来我们通过一个 Demo，来感受一下 shouldComponentUpdate 到底是如何解决问题的。在这个 Demo 中会涉及 3 个组件：子组件 ChildA、ChildB 及父组件 App 组件。

首先我们来看两个子组件的代码，这里为了尽量简化与数据变更无关的逻辑，ChildA 和 ChildB 都只负责从父组件处读取数据并渲染，它们的编码分别如下所示。

ChildA.js：

复制代码

```
import React from "react";
export default class ChildA extends React.Component {
  render() {
    console.log("ChildA 的render方法执行了");
    return (
      <div className="childA">
        子组件A的内容:
        {this.props.text}
      </div>
    );
  }
}
```

ChildB.js：

复制代码

```
import React from "react";
export default class ChildB extends React.Component {
  render() {
    console.log("ChildB 的render方法执行了");
    return (
      <div className="childB">
        子组件B的内容:
        {this.props.text}
      </div>
    );
  }
}
```

在共同的父组件 App.js 中，会将 ChildA 和 ChildB 组合起来，并分别向其中注入数据：

复制代码

```
import React from "react";
import ChildA from './ChildA'
import ChildB from './ChildB'
class App extends React.Component {
  state = {
    textA: '我是A的文本',
```

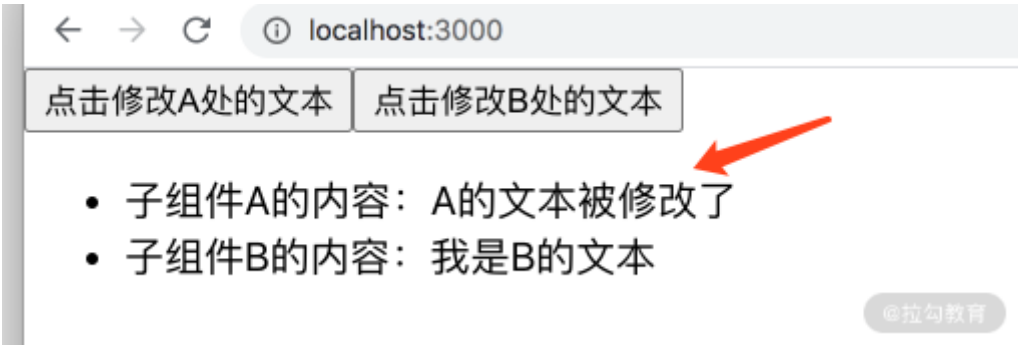
```
    textB: '我是B的文本'
  }
  changeA = () => {
    this.setState({
      textA: 'A的文本被修改了'
    })
  }
  changeB = () => {
    this.setState({
      textB: 'B的文本被修改了'
    })
  }
  render() {
    return (
    <div className="App">
      <div className="container">
        <button onClick={this.changeA}>点击修改A处的文本</button>
        <button onClick={this.changeB}>点击修改B处的文本</button>
        <ul>
          <li>
            <ChildA text={this.state.textA}/>
          </li>
        <li>
          <ChildB text={this.state.textB}/>
        </li>
        </ul>
      </div>
    </div>
  );
  }
}
export default App;
```

App 组件最终渲染到界面上的效果如下图所示，两个子组件在图中分别被不同颜色的标注圈出：

通过点击左右两个按钮，我们可以分别对 ChildA 和 ChildB 中的文案进行修改。

由于初次渲染时，两个组件的 render 函数都必然会被触发，因此控制台在挂载完成后的输出内容如下图所示：

接下来我点击左侧的按钮，尝试对 A 处的文本进行修改。我们可以看到界面上只有 A 处的渲染效果发生了改变，如下图箭头处所示：



但是如果我们打开控制台，会发现输出的内容如下图所示：

这样的输出结果告诉我们，在刚刚的点击动作后，不仅 ChildA 的 re-render 被触发了，ChildB 的 re-render 也被触发了。

在 React 中，**只要父组件发生了更新，那么所有的子组件都会被无条件更新**。这就导致了 ChildB 的 props 尽管没有发生任何变化，它本身也没有任何需要被更新的点，却还是会走一遍更新流程。

> 注：同样的情况也适用于组件自身的更新：当组件自身调用了 setState 后，那么不管 setState 前后的状态内容是否真正发生了变化，它都会去走一遍更新流程。

而在刚刚这个更新流程中，shouldComponentUpdate 函数没有被手动定义，因此它将返回“true”这个默认值。“true”则意味着对更新流程不作任何制止，也即所谓的“无条件 re-render”。在这种情况下，我们就可以考虑使用 shouldComponentUpdate 来对更新过程进行管控，避免没有意义的 re-render 发生。

现在我们就可以为 ChildB 加装这样一段 shouldComponentUpdate 逻辑：

```
shouldComponentUpdate(nextProps, nextState) {
  // 判断 text 属性在父组件更新前后有没有发生变化，若没有发生变化，则返回 false
  if(nextProps.text === this.props.text) {
    return false
  }
  // 只有在 text 属性值确实发生变化时，才允许更新进行下去
  return true
}
```

在这段逻辑中，我们对 ChildB 中的可变数据，也就是 this.props.text 这个属性进行了判断。

这样，当父组件 App 组件发生更新、进而试图触发 ChildB 的更新流程时，shouldComponentUpdate 就会充当一个“守门员”的角色：它会检查新下发的 props.text 是否和之前的值一致，如果一致，那么就没有更新的必要，直接返回“false”将整个 ChildB 的更新生命周期中断掉即可。只有当 props.text 确实发生变化时，它才会“准许” re-render 的发生。

在 shouldComponentUpdate 的加持下，当我们再次点击左侧按钮，试图修改 ChildA 的渲染内容时，控制台的输出就会变成下图这样：

我们看到，控制台中现在只有 ChildA 的 re-render 提示。ChildB “稳如泰山”，成功躲开了一次多余的渲染。

使用 shouldComponentUpdate 来调停不必要的更新，避免无意义的 re-render 发生，这是 React 组件中最基本的性能优化手段，也是最重要的手段。许多看似高级的玩法，都是基于 shouldComponentUpdate 衍生出来的。我们接下来要讲的 PureComponent，就是这类玩法中的典型。

进阶玩法：PureComponent + Immutable.js

PureComponent：提前帮你安排好更新判定逻辑

shouldComponentUpdate 虽然一定程度上帮我们解决了性能方面的问题，但每次避免 re-render，都要手动实现一次 shouldComponentUpdate，未免太累了。作为一个不喜欢重复劳动的前端开发者来

说，在写了不计其数个 shouldComponentUpdate 逻辑之后，难免会怀疑人生，进而发出由衷的感叹——“这玩意儿要是能内置到组件里该多好啊！”。

哪里有需求，哪里就有产品。React 15.3 很明显听到了开发者的声音，它新增了一个叫 PureComponent 的类，恰到好处地解决了“程序员写 shouldComponentUpdate 写出腱鞘炎”这个问题。

PureComponent 与 Component 的区别点，就在于它内置了对 shouldComponentUpdate 的实现：PureComponent 将会在 shouldComponentUpdate 中对组件更新前后的 props 和 state 进行**浅比较**，并根据浅比较的结果，决定是否需要继续更新流程。

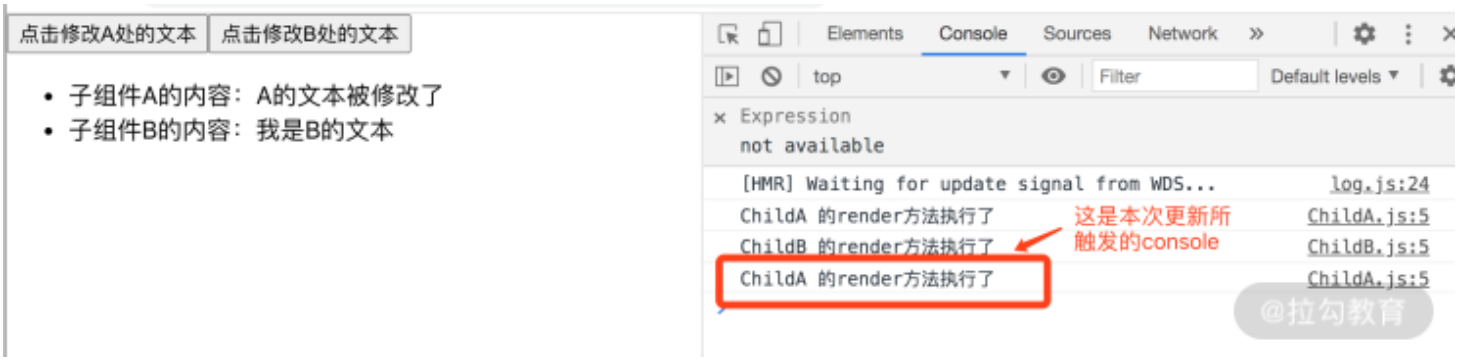
“浅比较”将针对值类型数据对比其值是否相等，而针对数组、对象等引用类型的数据则对比其引用是否相等。

在我们开篇的 Demo 中，若把 ChildB 的父类从 Component 替换为 PureComponent（修改后的代码如下所示），那么无须手动编写 shouldComponentUpdate，也可以达到同样避免 re-render 的目的。

复制代码

```
import React from "react";
export default class ChildB extends React.PureComponent {
  render() {
    console.log("ChildB 的render方法执行了");
    return (
      <div className="childB">
        子组件B的内容：
        {this.props.text}
      </div>
    );
  }
}
```

此时再去修改 ChildA 中的文本，我们会发现 ChildB 同样不受影响。点击左侧按钮后，控制台对应的输出内容如下图高亮处所示：



在值类型数据这种场景下，PureComponent 可以说是战无不胜。但是如果数据类型为引用类型，那么这种基于浅比较的判断逻辑就会带来这样两个风险：

1. 若数据内容没变，但是引用变了，那么浅比较仍然会认为“数据发生了变化”，进而触发一次不必要的更新，导致过度渲染；
2. 若数据内容变了，但是引用没变，那么浅比较则会认为“数据没有发生变化”，进而阻断一次更新，导致不渲染。

怎么办呢？Immutable.js 来帮忙！

Immutable：“不可变值”让“变化”无处遁形

PureComponent 浅比较带来的问题，本质上是对“变化”的判断不够精准导致的。那有没有一种办法，能够让引用的变化和内容的变化之间，建立一种必然的联系呢？

这就是 Immutable.js 所做的事情。

Immutable 直译过来是“不可变的”，顾名思义，Immutable.js 是对“不可变值”这一思想的贯彻实践。它在 2014 年被 Facebook 团队推出，Facebook 给它的定位是“实现持久性数据结构的库”。**所谓“持久性数据”，指的是这个数据只要被创建出来了，就不能被更改。我们对当前数据的任何修改动作，都会导致一个新的对象的返回**。这就将数据内容的变化和数据的引用严格地关联了起来，使得“变化”无处遁形。

这里我用一个简单的例子，来演示一下 Immutable.js 的效果。请看下面代码：

复制代码

```
// 引入 immutable 库里的 Map 对象，它用于创建对象
import { Map } from 'immutable'
// 初始化一个对象 baseMap
const baseMap = Map({
  name: '修言',
  career: '前端',
  age: 99
})
// 使用 immutable 暴露的 Api 来修改 baseMap 的内容
const changedMap = baseMap.set({
  age: 100
})
// 我们会发现修改 baseMap 后将会返回一个新的对象，这个对象的引用和 baseMap 是不同的
console.log('baseMap === changedMap', baseMap === changedMap)
```

由此可见，PureComonent 和 Immutable.js 真是一对好基友！在实际的开发中，我们也确实经常左手 PureComonent，右手 Immutable.js，研发质量大大地提升呀！

> 值得注意的是，由于 Immutable.js 存在一定的学习成本，并不是所有场景下都可以作为最优解被团队采纳。因此，一些团队也会基于 PureComonent 和 Immutable.js 去打造将两者结合的公共类，通过改写 setState 来提升研发体验，这也是不错的思路。

函数组件的性能优化：React.memo 和 useMemo

以上咱们讨论的都是类组件的优化思路。那么在函数组件中，有没有什么通用的手段可以阻止“过度 re-render”的发生呢？接下来我们就一起认识一下“函数版”的 shouldComponentUpdate/Purecomponent —— React.memo。

React.memo：“函数版”shouldComponentUpdate/PureComponent

React.memo 是 React 导出的一个顶层函数，它本质上是一个高阶组件，负责对函数组件进行包装。基本的调用姿势如下面代码所示：

复制代码

```
import React from "react";
// 定义一个函数组件
function FunctionDemo(props) {
  return xxx
}
// areEqual 函数是 memo 的第二个入参，我们之前放在 shouldComponentUpdate 里面的逻辑就可
function areEqual(prevProps, nextProps) {
  /*
  return true if passing nextProps to render would return
  the same result as passing prevProps to render,
  otherwise return false
  */
}
// 使用 React.memo 来包装函数组件
export default React.memo(FunctionDemo, areEqual);
```

React.memo 会帮我们“记住”函数组件的渲染结果，在组件前后两次 props 对比结果一致的情况下，它会直接复用最近一次渲染的结果。如果我们的组件在相同的 props 下会渲染相同的结果，那么使用 React.memo 来包装它将是个不错的选择。

从示例中我们可以看出，React.memo 接收两个参数，第一个参数是我们需要渲染的目标组件，第二个参数 areEqual 则用来承接 props 的对比逻辑。**之前我们在 shouldComponentUpdate 里面做的事情，现在就可以放在 areEqual 里来做**。

比如开篇 Demo 中的 ChildB 组件，就完全可以用 Function Component + React.memo 来改造。改造后的 ChildB 代码如下：

```
import React from "react";
// 将 ChildB 改写为 function 组件
function ChildB(props) {
  console.log("ChildB 的render 逻辑执行了");
  return (
    <div className="childB">
      子组件B的内容：
      {props.text}
    </div>
  );
}
// areEqual 用于对比 props 的变化
function areEqual(prevProps, nextProps) {
  if(prevProps.text === nextProps.text) {
    return true
  }
  return false
}
// 使用 React.memo 来包装 ChildB
export default React.memo(ChildB, areEqual);
```

复制代码

改造后的组件在效果上就等价于 shouldComponentUpdate 加持后的类组件 ChildB。

这里的 areEqual 函数是一个可选参数，当我们不传入 areEqual 时，React.memo 也可以工作，此时它的作用就类似于 PureComponent——React.memo 会自动为你的组件执行 props 的浅比较逻辑。

和 shouldComponentUpdate 不同的是，React.memo 只负责对比 props，而不会去感知组件内部状态（state）的变化。

useMemo：更加“精细”的 memo

通过上面的分析我们知道，React.memo 可以实现类似于 shouldComponentUpdate 或者 PureComponent 的效果，对组件级别的 re-render 进行管控。但是有时候，我们希望复用的并不是整个组件，而是组件中的某一个或几个部分。这种更加“精细化”的管控，就需要 useMemo 来帮忙了。

简而言之，React.memo 控制是否需要重渲染一个组件，而 useMemo 控制的则是是否需要重复执行某一段逻辑。

useMemo 的使用方式如下面代码所示：

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

复制代码

我们可以把目标逻辑作为第一个参数传入，把逻辑的依赖项数组作为第二个参数传入。这样只有当依赖项数组中的某个依赖发生变化时，useMemo 才会重新执行第一个入参中的目标逻辑。

这里我仍然以开篇的示例为例，现在我尝试向 ChildB 中传入两个属性：text 和 count，它们分别是一段文本和一个数字。当我点击右边的按钮时，只有 count 数字会发生变化。改造后的 App 组件代码如下：

复制代码

```
class App extends React.Component {
  state = {
    textA: '我是A的文本',
    stateB: {
      text: '我是B的文本',
      count: 10
    }
  }
  changeA = () => {
    this.setState({
      textA: 'A的文本被修改了'
    })
  }
  changeB = () => {
    this.setState({
      stateB: {
        ...this.state.stateB,
        count: 100
      }
    })
  }
  render() {
    return (
    <div className="App">
      <div className="container">
        <button onClick={this.changeA}>点击修改A处的文本</button>
        <button onClick={this.changeB}>点击修改B处的文本</button>
        <ul>
          <li>
            <ChildA text={this.state.textA}/>
          </li>
        <li>
          <ChildB {...this.state.stateB}/>
        </li>
        </ul>
      </div>
    </div>
  );
  }
}
export default App;
```

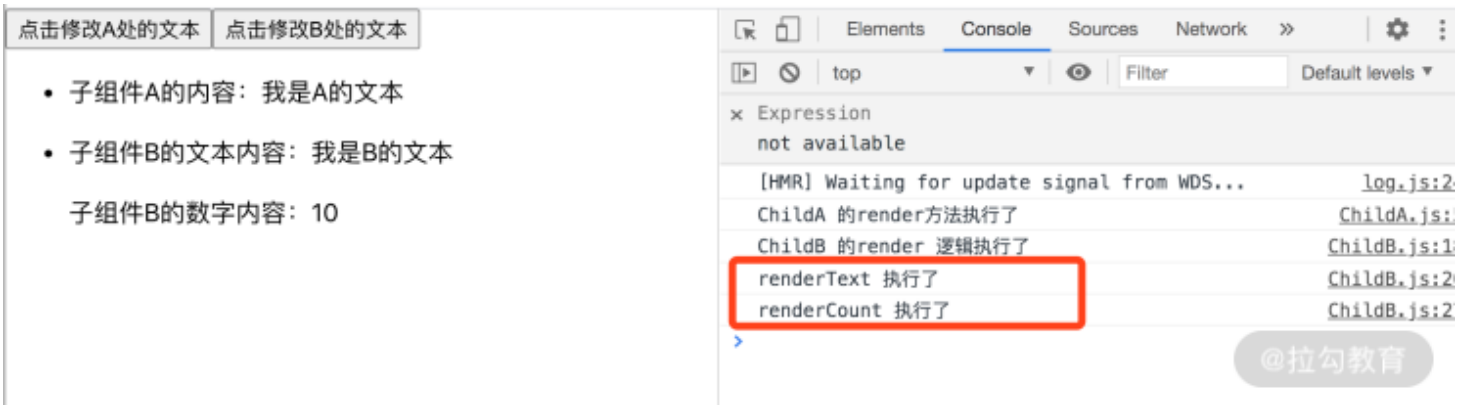
在 ChildB 中，使用 useMemo 来加持 text 和 count 各自的渲染逻辑。改造后的 ChildB 代码如下所示：

复制代码

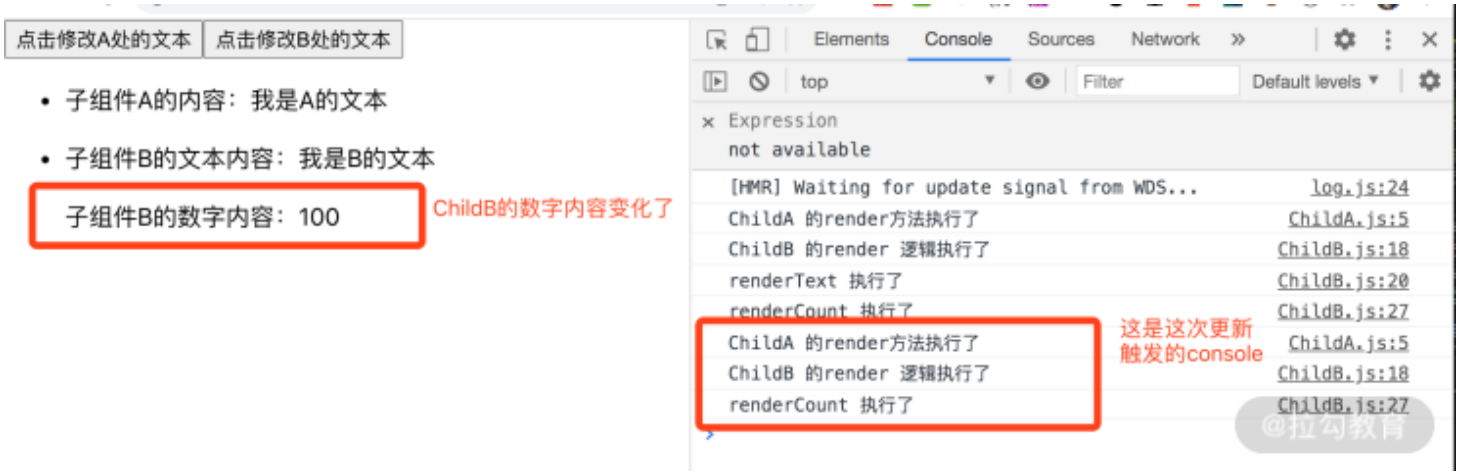
```
import React,{ useMemo } from "react";
export default function ChildB({text, count}) {
  console.log("ChildB 的render 逻辑执行了");
  // text 文本的渲染逻辑
  const renderText = (text)=> {
    console.log('renderText 执行了')
    return <p>
    子组件B的文本内容:
      {text}
  </p>
  }
  // count 数字的渲染逻辑
  const renderCount = (count) => {
    console.log('renderCount 执行了')
    return <p>
      子组件B的数字内容:
        {count}
    </p>
  }

  // 使用 useMemo 加持两段渲染逻辑
  const textContent = useMemo(()=>renderText(text),[text])
  const countContent = useMemo(()=>renderCount(count),[count])
  return (
    <div className="childB">
      {textContent}
      {countContent}
    </div>
  );
}
```

渲染 App 组件，我们可以看到初次渲染时，renderText 和 renderCount 都执行了，控制台输出如下图所示：



点击右边按钮，对 count 进行修改，修改后的界面会发生如下的变化：



可以看出，由于 count 发生了变化，因此 useMemo 针对 renderCount 的逻辑进行了重计算。而 text 没有发生变化，因此 renderText 的逻辑压根没有执行。

使用 useMemo，我们可以对函数组件的执行逻辑进行更加细粒度的管控（尤其是定向规避掉一些高开销的计算），同时也弥补了 React.memo 无法感知函数内部状态的遗憾，这对我们整体的性能提升是大有裨益的。

总结

本讲，我们学习了 React 组件性能优化中最重要的 3 个思路。

这 3 个思路不仅可以作为大家日常实战的知识储备，更能够帮助你在面试场景下做到言之有物。事实上，在“React 性能优化”这个问题下，许多候选人的回答犹如隔靴搔痒，总在一些无关紧要的细节上使劲儿。若你能把握好本讲的内容，择其中一个或多个方向深入探究，相信你已经超越了大部分的同行。

下一讲，我们将学习 React 组件的设计模式，为打造“高质量应用”做知识储备。