HOC学习总结

Created and last modified by 王文磊 on Mar 07, 2022

一、定义

高阶组件（HOC）是 React 中用于复用组件逻辑的一种高级技巧。HOC 自身不是 React API 的一部分，它是一种基于 React 的组合特性而形成的设计模式。

高阶组件本身不是组件，它是一个参数为组件，返回值也是一个组件的函数。高阶作用于强化组件，复用逻辑，提升渲染性能等作用。

二、发展历史

2.1 mixin模式

在react初期提供一种组合方法。通过React.createClass,加入mixins属性，具体用法和vue 中mixins相似。具体实现如下：

```
const customMixin = {
  componentDidMount(){
    console.log( '------componentDidMount------' )
  },
  say(){
    console.log(this.state.name)
  }
}

const APP = React.createClass({
  mixins: [ customMixin ],
  getInitialState(){
    return {
      name:'alien'
    }
  },
  render(){
    const { name  } = this.state
    return <div> hello ,world , my name is { name } </div>
  }
})
```

这种mixins只能存在createClass中，后来React.createClass连同mixins这种模式被废弃了。mixins会带来一些负面的影响：

1. mixins引入了隐式依赖关系
2. 不同mixins之间可能会有先后顺序甚至代码冲突覆盖的问题
3. mixin代码会导致雪球式的复杂性

2.2 原型链继承

createClass的废弃，不代表mixin模式退出react舞台，在有状态组件class，我们可以通过原型链继承来实现mixins

```
const customMixin = {  /* 自定义 mixins */
  componentDidMount(){
    console.log( '------componentDidMount------' )
  },
  say(){
    console.log(this.state.name)
  }
}

function componentClassMixins(Component,mixin){ /* 继承 */
  for(let key in mixin){
    Component.prototype[key] = mixin[key]
  }
}

class Index extends React.Component{
  constructor(){
    super()
    this.state={  name:'alien' }
  }
  render(){
    return <div> hello,world
      <button onClick={ this.say.bind(this) } > to say </button>
    </div>
  }
}
componentClassMixins(Index,customMixin)
```

2.3 extends继承模式

在class组件盛行之后，我们可以通过继承的方式进一步的强化我们的组件。这种模式的好处在于，可以封装基础功能组件，然后根据需要去extends我们的基础组件，按需强化组件，但是值得注意的是，必须要对基础组件有足够的掌握，否则会造成一些列意想不到的情况发生

```
class Base extends React.Component{
  constructor(){
    super()
    this.state={
      name:'alien'
    }
  }
  say(){
    console.log('base components')
  }
  render(){
    return <div> hello,world <button onClick={ this.say.bind(this) } >点击</button>  </div>
  }
}
class Index extends Base{
  componentDidMount(){
    console.log( this.state.name )
  }
  say(){ /* 会覆盖基类中的 say  */
    console.log('extends components')
  }
}
export default Index
```

2.4 HOC模式

一个简单的HOC

```
function HOC(Component) {
  return class wrapComponent extends React.Component{
     constructor(){
       super()
       this.state={
         name:'alien'
       }
     }
     render=()=><Component { ...this.props } { ...this.state } />
  }
}
@HOC
class Index extends React.Component{
  say(){
    const { name } = this.props
    console.log(name)
  }
  render(){
    return <div> hello,world <button onClick={ this.say.bind(this) } >点击</button>  </div>
  }
}
```

2.5 自定义hooks模式

hooks的诞生，一大部分原因是解决无状态组件没有state和逻辑难以复用问题。hooks可以将一段逻辑封装起来，做到开箱即用

三、HOC使用模式

- 使用：装饰器模式和函数包裹模式

类组件 —— 装饰器模式：

```
@withStyles(styles)
@withRouter
@keepaliveLifeCycle
class Index extends React.Componen{
    /* ... */
}
```

函数组件 —— 包裹模式：

```
function Index(){
    /* .... */
}
export default withStyles(styles)(withRouter( keepaliveLifeCycle(Index) ))
```

- 嵌套HOC

对于不需要传递参数的HOC，我们编写模型我们只需要嵌套一层就可以

```
function withRouter(){
    return class wrapComponent extends React.Component{
        /* 编写逻辑 */
    }
}
```

对于需要参数的HOC，我们需要一层代理

```
function connect (mapStateToProps){
    /* 接受第一个参数 */
    return function connectAdvance(wrapCompoent){
        /* 接受组件 */
        return class WrapComponent extends React.Component{   }
    }
}
```

四、两种HOC

正向属性代理

所谓正向属性代理，就是用组件包裹一层代理组件，在代理组件上，我们可以做一些，对源组件的代理操作。在fiber tree 上，先mounted代理组件，然后才是我们的业务组件。我们可以理解为父子组件关系，父组件对子组件进行一系列强化操作。

例子：

```
class Index extends React.Component{
  render(){
    return <div> hello,world  </div>
  }
}
Index.say = function(){
  console.log('my name is alien')
}
function HOC(Component) {
  return class wrapComponent extends React.Component{
     render(){
       return <Component { ...this.props } { ...this.state } />
     }
  }
}
const newIndex =  HOC(Index)
console.log(newIndex.say)
```

优点：

- 正常属性代理可以和业务组件低耦合，零耦合，对于条件渲染和props属性增强,只负责控制子组件渲染和传递额外的props就可以，所以无须知道，业务组件做了些什么。所以正向属性代理，更适合做一些开源项目的hoc，目前开源的HOC基本都是通过这个模式实现的。
- 同样适用于class声明组件，和function声明的组件。
- 可以完全隔离业务组件的渲染,相比反向继承，属性代理这种模式。可以完全控制业务组件渲染与否，可以避免反向继承带来一些副作用，比如生命周期的执行。
- 可以嵌套使用，多个hoc是可以嵌套使用的，而且一般不会限制包装HOC的先后顺序。

缺点：

- 一般无法直接获取业务组件的状态，如果想要获取，需要ref获取组件实例。
- 无法直接继承静态属性。如果需要继承需要手动处理，或者引入第三方库。

反向继承

反向继承和属性代理有一定的区别，在于包装后的组件继承了业务组件本身，所以我们我无须在去实例化我们的业务组件。当前高阶组件就是继承后，加强型的业务组件。这种方式类似于组件的强化，所以你必要要知道当前业务组件内部的状态。

例子：

```
class Index extends React.Component{
  render(){
    return <div> hello,world  </div>
  }
}
Index.say = function(){
  console.log('my name is alien')
}
function HOC(Component) {
  return class wrapComponent extends Component{
  }
}
const newIndex =  HOC(Index)
console.log(newIndex.say)
```

优点：

- ① 方便获取组件内部状态，比如state，props ,生命周期,绑定的事件函数等
- ② es6继承可以良好继承静态属性。我们无须对静态属性和方法进行额外的处理。

缺点：

- ① 无状态组件无法使用。
- ② 和被包装的组件强耦合，需要知道被包装的组件的内部状态，具体是做什么?
- ③ 如果多个反向继承hoc嵌套在一起，当前状态会覆盖上一个状态。这样带来的隐患是非常大的，比如说有多个componentDidMount，当前componentDidMount会覆盖上一个componentDidMount。这样副作用串联起来，影响很大。

五、编写实践

使用场景一：强化props

- **混入props**

这个是高阶组件最常用的功能，承接上层的props,在混入自己的props，来强化组件。

类HOC（属性代理）：

```
function classHOC(WrapComponent){
    return class  Idex extends React.Component{
        state={
            name:'alien'
        }
        componentDidMount(){
           console.log('HOC')
        }
        render(){
            return <WrapComponent { ...this.props }  { ...this.state }   />
        }
    }
}
function Index(props){
  const { name } = props
  useEffect(()=>{
     console.log( 'index' )
  },[])
  return <div>
    hello,world , my name is { name }
  </div>
}
export default classHOC(Index)
```

函数HOC（属性代理）：

```
function functionHoc(WrapComponent){
    return function Index(props){
        const [ state , setState ] = useState({ name :'alien'  })
        return  <WrapComponent { ...props }  { ...state }   />
    }
}
```

- **抽离state控制更新**

高阶组件可以将HOC的state的配合起来，控制业务组件的更新。

改造上述类HOC：

```
function classHOC(WrapComponent){
  return class  Idex extends React.Component{
      constructor(){
        super()
        this.state={
          name:'alien'
        }
      }
      changeName(name){
        this.setState({ name })
      }
      render(){
          return <WrapComponent { ...this.props }  { ...this.state } changeName={this.changeN
      }
  }
}
function Index(props){
  const [ value ,setValue ] = useState(null)
  const { name ,changeName } = props
  return <div>
    <div>   hello,world , my name is { name }</div>
    改变name <input onChange={ (e)=> setValue(e.target.value)  }  />
    <button onClick={ ()=>  changeName(value) }  >确定</button>
  </div>
}
export default classHOC(Index)
```

使用场景二：控制渲染

条件渲染：

- **动态渲染**

对于属性代理的高阶组件，虽然不能在内部操控渲染状态，但是可以在外层控制当前组件是否渲染，这种情况应用于，权限隔离，懒加载，延时加载等场景。

具体实现就是使用HOC中的state控制业务组件挂载与否

- **分片渲染**

实现一个懒加载功能的HOC，可以实现组件的分片渲染,用于分片渲染页面，不至于一次渲染大量组件造成白屏效果

```
const renderQueue = []
let isFirstrender = false
const tryRender = ()=>{
  const render = renderQueue.shift()
  if(!render) return
  setTimeout(()=>{
    render() /* 执行下一段渲染 */
  },300)
}
/* HOC */
function renderHOC(WrapComponent){
    return function Index(props){
      const [ isRender , setRender ] = useState(false)
      useEffect(()=>{
        renderQueue.push(()=>{  /* 放入待渲染队列中 */
          setRender(true)
        })
        if(!isFirstrender) {
          tryRender() /**/
          isFirstrender = true
        }
      },[])
      return isRender ? <WrapComponent tryRender={tryRender}  { ...props }  /> : <div classNa
    }
}
/* 业务组件 */
class Index extends React.Component{
  componentDidMount(){
    const { name , tryRender} = this.props
    /* 上一部分渲染完毕，进行下一部分渲染 */
    tryRender()
    console.log( name+'渲染')
  }
  render(){
    return <div>
        <img src="https://ss2.bdstatic.com/70cFvnSh_Q1YnxGkpoWK1HF6hhy/it/u=294206908,2427609
    </div>
  }
}
/* 高阶组件包裹 */
const Item = renderHOC(Index)
export default () => {
  return <React.Fragment>
      <Item name="组件一" />
      <Item name="组件二" />
      <Item name="组件三" />
  </React.Fragment>
}
```

- **异步组件（懒加载）**

HOC不一定要和组件绑定，可以利用React Component的一些特性（生命周期）来衍生其他操作

```
/* 路由懒加载HOC */
export default function AsyncRouter(loadRouter) {
  return class Content extends React.Component {
    state = {Component: null}
    componentDidMount() {
      if (this.state.Component) return
      loadRouter()
        .then(module => module.default)
        .then(Component => this.setState({Component},
         ))
    }
    render() {
      const {Component} = this.state
      return Component ? <Component {
      ...this.props
      }
      /> : null
    }
  }
}

/* 使用 */
const Index = AsyncRouter(()=>import('../pages/index'))
```

- **反向继承： 渲染劫持**

HOC反向继承模式，可以实现颗粒化的渲染劫持，也就是可以控制基类组件的render函数，还可以篡改props，或者是children

```
const HOC = (WrapComponent) =>
  class Index  extends WrapComponent {
    render() {
      if (this.props.visible) {
        return super.render()
      } else {
        return <div>暂无数据</div>
      }
    }
  }
```

- **反向继承：修改渲染树**

修改渲染状态(劫持render替换子节点)

```
class Index extends React.Component{
  render(){
    return <div>
       <ul>
         <li>react</li>
         <li>vue</li>
         <li>Angular</li>
       </ul>
    </div>
  }
}
function HOC (Component){
  return class Advance extends Component {
    render() {
      const element = super.render()
      const otherProps = {
        name:'alien'
      }
      /* 替换 Angular 元素节点 */
      const appendElement = React.createElement('li' ,{} , `hello ,world , my name  is ${ oth
      const newchild =  React.Children.map(element.props.children.props.children,(child,index
           if(index === 2) return appendElement
           return  child
      })
      return  React.cloneElement(element, element.props, newchild)
    }
  }
}
export  default HOC(Index)
```

节流渲染：

- **基础节流（搭配Hooks）**

```
function HOC (Component){
     return function renderWrapComponent(props){
       const { num } = props
       const RenderElement = useMemo(() =>  <Component {...props}  /> ,[ num ])
       return RenderElement
     }
}
class Index extends React.Component{
  render(){
     console.log(`当前组件是否渲染`,this.props)
     return <div>hello,world, my name is alien </div>
  }
}
const IndexHoc = HOC(Index)
export default ()=> {
    const [ num ,setNumber ] = useState(0)
    const [ num1 ,setNumber1 ] = useState(0)
    const [ num2 ,setNumber2 ] = useState(0)
    return <div>
        <IndexHoc  num={ num } num1={num1} num2={ num2 }  />
        <button onClick={() => setNumber(num + 1) } >num++</button>
        <button onClick={() => setNumber1(num1 + 1) } >num1++</button>
        <button onClick={() => setNumber2(num2 + 1) } >num2++</button>
    </div>
}
```

- **进阶，定制化渲染流**

```
 function HOC (rule){
     return function (Component){
        return function renderWrapComponent(props){
          const dep = rule(props)
          const RenderElement = useMemo(() =>  <Component {...props}  /> ,[ dep ])
          return RenderElement
        }
     }
}
/* 只有 props 中 num 变化 , 渲染组件  */
@HOC( (props)=> props['num'])
class IndexHoc extends React.Component{
  render(){
     console.log(`组件一渲染`,this.props)
     return <div> 组件一 ：  hello,world </div>
  }
}
/* 只有 props 中 num1 变化 , 渲染组件  */
@HOC((props)=> props['num1'])
class IndexHoc1 extends React.Component{
  render(){
     console.log(`组件二渲染`,this.props)
     return <div> 组件二 ：  my name is alien </div>
  }
}
export default ()=> {
    const [ num ,setNumber ] = useState(0)
    const [ num1 ,setNumber1 ] = useState(0)
    const [ num2 ,setNumber2 ] = useState(0)
    return <div>
        <IndexHoc  num={ num } num1={num1} num2={ num2 }  />
        <IndexHoc1  num={ num } num1={num1} num2={ num2 }  />
        <button onClick={() => setNumber(num + 1) } >num++</button>
        <button onClick={() => setNumber1(num1 + 1) } >num1++</button>
        <button onClick={() => setNumber2(num2 + 1) } >num2++</button>
    </div>
}
```

使用场景三：赋能组件

高阶组件除了上述两种功能之外，还可以赋能组件，比如加一些额外生命周期，劫持事件，监控日志等等。

劫持原型链-劫持生命周期，事件函数：

① 属性代理实现

```
function HOC (Component){
  const proDidMount = Component.prototype.componentDidMount
  Component.prototype.componentDidMount = function(){
     console.log('劫持生命周期：componentDidMount')
     proDidMount.call(this)
  }
  return class wrapComponent extends React.Component{
      render(){
        return <Component {...this.props}  />
      }
  }
}
@HOC
class Index extends React.Component{
   componentDidMount(){
     console.log('——didMounted——')
   }
   render(){
     return <div>hello,world</div>
   }
}
```

② 反向继承实现

```
function HOC (Component){
  const didMount = Component.prototype.componentDidMount
  return class wrapComponent extends Component{
      componentDidMount(){
        console.log('------劫持生命周期------')
        if (didMount) {
           didMount.apply(this) /* 注意 `this` 指向问题。 */
        }
      }
      render(){
        return super.render()
      }
  }
}
@HOC
class Index extends React.Component{
   componentDidMount(){
     console.log('——didMounted——')
   }
   render(){
     return <div>hello,world</div>
   }
}
```

事件监控：

① 组件内的事件监听

```
function ClickHoc (Component){
  return  function Wrap(props){
    const dom = useRef(null)
    useEffect(()=>{
     const handerClick = () => console.log('发生点击事件')
     dom.current.addEventListener('click',handerClick)
     return () => dom.current.removeEventListener('click',handerClick)
    },[])
    return  <div ref={dom}  ><Component  {...props} /></div>
  }
}
@ClickHoc
class Index extends React.Component{
   render(){
     return <div  className='index'  >
       <p>hello, world</p>
       <button>组件内部点击</button>
    </div>
   }
}
export default ()=>{
  return <div className='box'  >
     <Index />
     <button>组件外部点击</button>
  </div>
}
```

ref助力操控组件实例：

对于属性代理我们虽然不能直接获取组件内的状态，但是我们可以通过ref获取组件实例,获取到组件实例，就可以获取组件的一些状态，或是手动触发一些事件，进一步强化组件，但是注意的是：class声明的有状态组件才有实例，function声明的无状态组件不存在实例。

① 属性代理-添加额外生命周期

```
function Hoc(Component){
  return class WrapComponent extends React.Component{
      constructor(){
        super()
        this.node = null
      }
      UNSAFE_componentWillReceiveProps(nextprops){
          if(nextprops.number !== this.props.number ){
            this.node.handerNumberChange  &&  this.node.handerNumberChange.call(this.node)
          }
      }
      render(){
        return <Component {...this.props} ref={(node) => this.node = node }  />
      }
  }
}
@Hoc
class Index extends React.Component{
  handerNumberChange(){
      /* 监听 number 改变 */
  }
  render(){
    return <div>hello,world</div>
  }
}
```

六、总结

对于属性代理HOC，我们可以：

- **强化props & 抽离state。**
- **条件渲染，控制渲染，分片渲染，懒加载。**
- **劫持事件和生命周期**
- **ref控制组件实例**
- **添加事件监听器，日志**

对于反向代理的HOC,我们可以：

- **劫持渲染，操纵渲染树**
- **控制/替换生命周期，直接获取组件状态，绑定事件。**

七、注意事项

1. 谨慎修改原型链条

```
function HOC (Component){
  const proDidMount = Component.prototype.componentDidMount
  Component.prototype.componentDidMount = function(){
     console.log('劫持生命周期：componentDidMount')
     proDidMount.call(this)
  }
  return  Component
}
```

这样做会产生一些不良后果。比如如果你再用另一个同样会修改 componentDidMount 的 HOC 增强它，那么前面的 HOC 就会失效！同时，这个 HOC 也无法应用于没有生命周期的函数组件。

2. 继承静态属性

在用属性代理的方式编写HOC的时候，要注意的是就是，静态属性丢失的问题，前面提到了，如果不做处理，静态方法就会全部丢失。

① 手动继承

我们可以手动将原始组件的静态方法copy到 hoc组件上来，但前提是必须准确知道应该拷贝哪些方法。

```
function HOC(Component) {
  class WrappedComponent extends React.Component {
      /*...*/
  }
  // 必须准确知道应该拷贝哪些方法
  WrappedComponent.staticMethod = Component.staticMethod
  return WrappedComponent
}
```

② 引入第三方库

这样每个静态方法都绑定会很累，尤其对于开源的hoc，对原生组件的静态方法是未知的,我们可以使用 hoist-non-react-statics 自动拷贝所有的静态方法:

```
import hoistNonReactStatic from 'hoist-non-react-statics'
function HOC(Component) {
  class WrappedComponent extends React.Component {
      /*...*/
  }
  hoistNonReactStatic(WrappedComponent,Component)
  return WrappedComponent
}
```

3. 跨层级捕获ref

高阶组件的约定是将所有 props 传递给被包装组件，但这对于 refs 并不适用。那是因为 ref 实际上并不是一个 prop - 就像 key 一样，它是由 React 专门处理的。如果将 ref 添加到 HOC 的返回组件中，则 ref 引用指向容器组件，而不是被包装组件。我们可以通过forwardRef来解决这个问题。

```
/**
 *
 * @param {*} Component 原始组件
 * @param {*} isRef  是否开启ref模式
 */
function HOC(Component,isRef){
  class Wrap extends React.Component{
     render(){
        const { forwardedRef ,...otherprops  } = this.props
        return <Component ref={forwardedRef}  {...otherprops}  />
     }
  }
    if(isRef){
      return  React.forwardRef((props,ref)=> <Wrap forwardedRef={ref} {...props} /> )
    }
    return Wrap
}
class Index extends React.Component{
  componentDidMount(){
      console.log(666)
  }
  render(){
    return <div>hello,world</div>
  }
}
const HocIndex =  HOC(Index,true)
export default ()=>{
  const node = useRef(null)
  useEffect(()=>{
     /* 就可以跨层级，捕获到 Index 组件的实例了 */
    console.log(node.current.componentDidMount)
  },[])
  return <div><HocIndex ref={node}  /></div>
}
```

4. render中不要声明HOC

错误写法：

```
class Index extends React.Component{
  render(){
     const WrapHome = HOC(Home)
     return <WrapHome />
  }
}
```

如果这么写，会造成一个极大的问题，因为每一次HOC都会返回一个新的WrapHome,react diff会判定两次不是同一个组件，那么每次Index 组件 render触发，WrapHome，会重新挂载，状态会全都丢失。如果想要动态绑定HOC,请参考如下方式。

正确写法：

```
const WrapHome = HOC(Home)
class index extends React.Component{
  render(){
     return <WrapHome />
  }
}
```

学习资料：

https://zh-hans.reactjs.org/docs/higher-order-components.html

https://juejin.cn/post/6940422320427106335#heading-54

Like Be the first to like this