⚡一文弄懂 React HOC

1. 提出问题

1. HOC 能解决什么问题？
2. HOC 的使用场景？

2. HOC 能解决什么问题？

1. 拦截组件渲染，包括是否渲染组件、懒加载组件
2. 往组件的 props 中混入所需的东西，比如给非 Route 组件的 props 混入 history 对象，使其能够支持路由跳转
3. 监控组件内部状态，如添加额外生命周期、对组件某些事件进行监听等

3. 两种高阶组件 -- 属性代理和反向继承

3.1. 属性代理

将原始组件包裹在代理组件中进行增加，并且原始组件是在代理组件中进行渲染的，因此代理组件可以掌控对原始组件的控制权限

tsx复制代码

```tsx
/** @description 属性代理 */

import React from 'react'

class HOCDemo1 extends React.Component {
  render(): React.ReactNode {
    const TargetComponent = HOC(Foo)
    return <TargetComponent />
  }
}

const HOC = (Component: typeof React.Component) => {
  return class WrappedComponent extends React.Component {
    render(): React.ReactNode {
      // 对原始组件的强化操作 -- 混入 props
      const fooEnhancedProps: FooEnhancedProps = {
        name: 'foo',
      }

      // 原始组件在代理组件中渲染
      return <Component {...fooEnhancedProps} />
    }
```

```tsx
  }
}

type FooRawProps = {}
type FooEnhancedProps = { name: string }
type FooProps = FooRawProps & FooEnhancedProps
class Foo extends React.Component<FooProps> {
  render(): React.ReactNode {
    const { name } = this.props
    return <div>name: {name}</div>
  }
}

export { HOCDemo1 }
```

这里通过属性代理往原始组件的 props 中混入了新的属性

3.1.1. 优点

1. 代理组件和原始组件低耦合
2. 类组件和函数组件均可使用
3. 可以控制原始组件是否渲染
4. 可以嵌套使用

3.1.2. 缺点

1. 无法直接获取原始组件的状态，需要通过 ref 获取
2. 无法直接继承静态属性，需要额外实现或者使用第三方库才行
3. 如果需要保持 ref 的正确指向，需要配合 forwardRef 转发 ref 到原始组件上

3.2. 反向继承

返回一个继承自原始组件的组件

tsx复制代码

```tsx
/** @description 反向继承 */

import React from 'react'

class HOCDemo2 extends React.Component {
  render(): React.ReactNode {
    const TargetComponent = HOC(Foo)
    return <TargetComponent />
  }
}

const HOC = (Component: typeof Foo) => {
  return class WrappedComponent extends Component {
```

```
    state: Readonly<FooState> = {
      name: 'foo',
    }
  }
}

interface FooState {
  name: string
}
class Foo extends React.Component<{}, FooState> {
  render(): React.ReactNode {
    const { name } = this.state
    return <div>name: {name}</div>
  }
}

export { HOCDemo2 }
```

这里通过反向继承修改原始组件的 state 实现了和上面属性代理的 Demo 中相同的效果

3.2.1. 优点

1. 能够很方便地获取组件内部的状态，如 state、props、生命周期、事件处理函数等
2. 基于 ES6 的继承可以很好地继承静态属性，无需额外实现或者第三方库即可管理静态属性

3.2.2. 缺点

1. 代理组件与原始组件高耦合
2. 函数组件无法使用
3. 嵌套使用有风险，内层组件的生命周期会覆盖外层组件的生命周期

4. HOC 的使用场景

4.1. 强化 props

也就是在原始组件的 props 上混入别的 props 以强化原始组件的功能，比如 React Router 的 withRouter

tsx复制代码

```
function withRouter(Component) {
  const displayName = `withRouter(${Component.displayName || Component.name}
  const C = (props) => {
    /*  获取 */
    const { wrappedComponentRef, ...remainingProps } = props
    return (
      <RouterContext.Consumer>
        {(context) => {
          return (
```

```
            <Component
              {...remainingProps} // 组件原始的props
              {...context} // 存在路由对象的上下文，history  location 等
              ref={wrappedComponentRef}
            />
          )
        }}
      </RouterContext.Consumer>
    )
  }

  C.displayName = displayName
  C.WrappedComponent = Component
  /* 继承静态属性 */
  return hoistStatics(C, Component)
}
export default withRouter
```

4.2. 劫持控制渲染逻辑

利用反向继承的特点，能够通过 `super.render()` 调用原始组件的渲染函数完成渲染，并且能够获取和修改渲染后的 React Element

tsx复制代码

```
/** @description 劫持控制渲染 */

import React from 'react'

const HOCDemo3: React.FC = () => {
  const TargetComponent = HOC(Foo)
  return <TargetComponent />
}

const HOC = (Component: typeof React.Component) => {
  return class WrappedComponent extends Component {
    render(): React.ReactNode {
      // 调用原始组件的 render 方法获取渲染后的 React Element
      const el = super.render()
      // @ts-ignore
      const rawChildren = el.props.children

      // 修改 el
      const modifiedChildren = React.Children.map(rawChildren, (child, idx)
        if (idx === 0) {
          return <li>React 666</li>
        }

        return child
      })

      // @ts-ignore
      return React.cloneElement(el, el.props, modifiedChildren)
    }
  }
```

```tsx
}

class Foo extends React.Component {
  render(): React.ReactNode {
    return (
      <ul>
        <li>React</li>
        <li>Vue</li>
        <li>Solid</li>
        <li>Svelte</li>
      </ul>
    )
  }
}

export { HOCDemo3 }
```

4.3. 动态加载组件

tsx复制代码

```tsx
/** @description 动态加载组件 */

import React, { useEffect, useState } from 'react'

const HOCDemo4: React.FC = () => {
  const TargetComponent = DynamicLoadHOC(() => {
    return new Promise((resolve) => {
      setTimeout(() => {
        resolve(import('./component'))
      }, 5000)
    })
  })

  return (
    <div>
      <TargetComponent />
    </div>
  )
}

interface WrappedComponentState {
  Component: typeof React.Component | null
}
const DynamicLoadHOC = (loader: () => Promise<any>) => {
  return class WrappedComponent extends React.Component<
    {},
    WrappedComponentState
  > {
    state: Readonly<WrappedComponentState> = {
      Component: null,
    }

    componentDidMount(): void {
      if (this.state.Component) return
      loader()
```

```
          .then((module) => module.default)
          .then((Component) => this.setState({ Component }))
    }

    render(): React.ReactNode {
      const { Component } = this.state
      return Component ? <Component {...this.props} /> : <Loading />
    }
  }
}

const Loading: React.FC = () => {
  return <div>Loading...</div>
}

export { HOCDemo4 }
```

4.4. 利用装饰器模式监听组件事件

使用一个外部容器元素包裹原始组件，给这个外部容器元素添加相应事件监听器，本质上就是利用事件代理机制起到一个监听原始组件相应事件的作用

tsx复制代码

```
import React from 'react'

import { createLoggerWithScope } from '~/utils'

const logger = createLoggerWithScope('HOCDemo5')

const OnClickHOC = (Component: typeof React.Component) => {
  class WrappedComponent extends React.Component {
    handleClick() {
      logger.log('检测到 click 事件触发')
    }

    render(): React.ReactNode {
      return (
        <div onClick={() => this.handleClick()}>
          <Component {...this.props} />
        </div>
      )
    }
  }

  return WrappedComponent
}
```

```tsx
@OnClickHOC
class HOCDemo5 extends React.Component {
  render(): React.ReactNode {
    return <button>Button</button>
  }
}

export { HOCDemo5 }
```

可以看到，`HOCDemo5` 组件中并没有添加事件监听器，但是只要加上 `OnClickHOC` 装饰器，就可以监听到它的点击事件触发，十分方便！

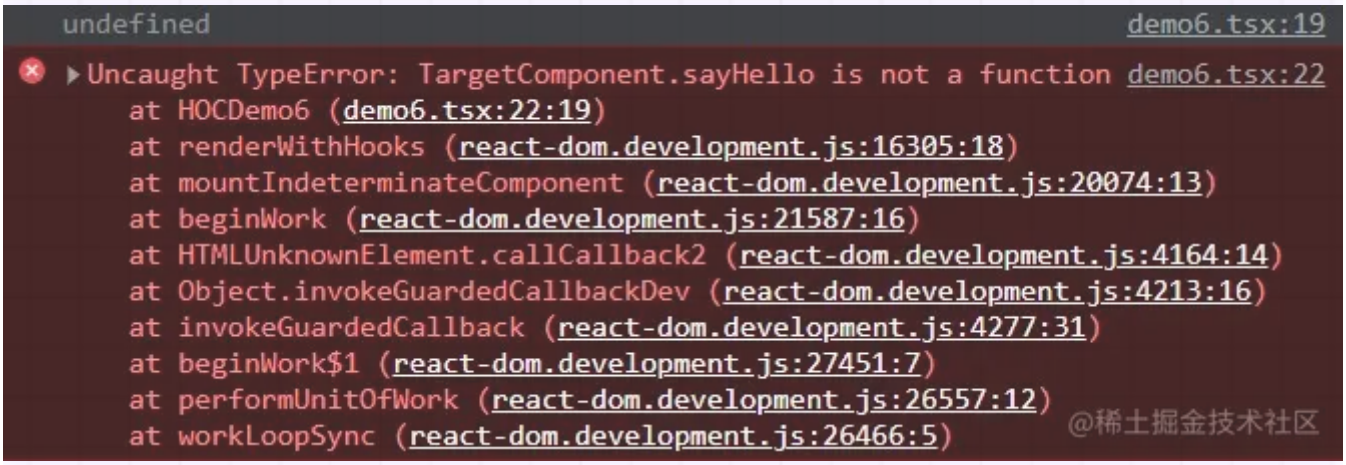
5. 如何处理静态属性丢失问题？

考虑一下下面这个场景，我们的原始组件上有静态属性和方法，但是 HOC 返回的 WrappedComponent 中是不存在原始组件的静态属性和方法的，这就导致用户在使用了我们的 HOC 后，原来绑定的静态属性和方法莫名其妙丢失了！

tsx复制代码

```tsx
/** @description HOC 原始类组件静态属性和方法丢失 */

import React from 'react'

const HOC = (Component: typeof React.Component) => {
  class WrappedComponent extends React.Component {
    render(): React.ReactNode {
      return <Component />
    }
  }

  return WrappedComponent
}

const HOCDemo6 = () => {
  const TargetComponent = HOC(Foo)

  // @ts-ignore
  console.log(TargetComponent.age)

  // @ts-ignore
  TargetComponent.sayHello()

  return <TargetComponent />
}

class Foo extends React.Component {
  static age = 21
```

```tsx
  static sayHello() {
    console.log('hello')
  }

  render(): React.ReactNode {
    return <div>Foo</div>
  }
}

export { HOCDemo6 }
```



那么该如何解决呢？一个很自然的想法是手动将原始组件上的静态属性引用拷贝到 `WrappedComponent` 上

tsx{8-12}复制代码

```tsx
const HOC = (Component: typeof React.Component) => {
  class WrappedComponent extends React.Component {
    render(): React.ReactNode {
      return <Component />
    }
  }

  // @ts-ignore
  WrappedComponent.age = Component.age

  // @ts-ignore
  WrappedComponent.sayHello = Component.sayHello

  return WrappedComponent
}
```

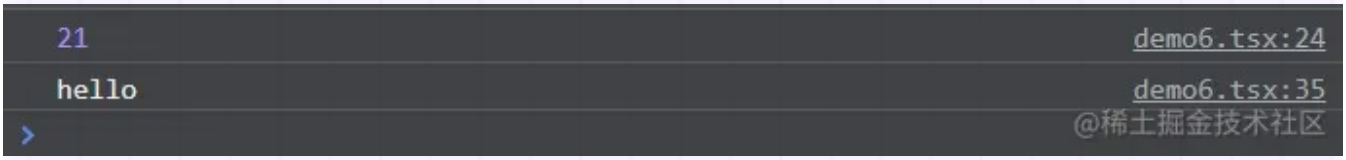
但这样子其实不太合理，毕竟我们不可能知道用户在使用我们的 HOC 时传入的原始类组件上有什么静态属性和方法，这里推荐使用一个名为 `hoist-non-react-statics` 的库，它可以帮我们拷贝一个类的静态属性和方法到另一个类上

shell复制代码

```shell
pnpm i hoist-non-react-statics
pnpm i @types/hoist-non-react-statics -D
```

tsx复制代码

```tsx
const HOC = (Component: typeof React.Component) => {
  class WrappedComponent extends React.Component {
    render(): React.ReactNode {
      return <Component />
    }
  }

  // 拷贝 Component 静态属性和方法到 WrappedComponent 上
  hoistNonReactStatics(WrappedComponent, Component)

  return WrappedComponent
}
```



6. 总结

本篇文章我们学习到了：

1. 为什么要用 HOC，HOC 解决了什么问题
2. HOC 的两种使用方式 -- 属性代理和反向继承，并分别介绍了它们的优缺点
3. 介绍了四种 HOC 的使用场景，并通过 Demo 加深理解
4. 如何解决 HOC 中原始类组件的静态属性和方法丢失