React 进阶实践指南 - 我不是外星人 - 掘金课程

juejin.cn/book/6945998773818490884/section/6948969962421616654

一 前言

在 React 世界里，一切皆组件，我们写的 React 项目全部起源于组件。组件可以分为两类，一类是类（ Class ）组件，一类是函数（ Function ）组件。

本章节，我们将一起探讨 React 中类组件和函数组件的定义，不同组件的通信方式，以及常规组件的强化方式，帮助你全方位认识 React 组件，从而对 React 的底层逻辑有进一步的理解。

二 什么是React组件?

想要理解 React 组件是什么？我们首先要来分析一下组件和常规的函数和类到底有什么本质的区别。

```
/* 类 */
class textClass {
    sayHello=()=>console.log('hello, my name is alien')
}
/* 类组件 */
class Index extends React.Component{
    state={ message:`hello , world!` }
    sayHello=()=> this.setState({ message : 'hello, my name is alien' })
    render(){
        return <div style={{ marginTop:'50px' }} onClick={ this.sayHello } > {
this.state.message }  </div>
    }
}
/* 函数 */
function textFun (){
    return 'hello, world'
}
/* 函数组件 */
function FunComponent(){
    const [ message , setMessage ] = useState('hello,world')
    return <div onClick={ ()=> setMessage('hello, my name is alien')  } >{ message }</div>
}
```

我们从上面可以清楚地看到，组件本质上就是类和函数，但是与常规的类和函数不同的是，组件承载了渲染视图的 **UI** 和更新视图的 **setState** 、 **useState** 等方法。React 在底层逻辑上会像正常实例化类和正常执行函数那样处理的组件。

因此，函数与类上的特性在 React 组件上同样具有，比如原型链，继承，静态属性等，所以不要把 React 组件和类与函数独立开来。

接下来，我们一起着重看一下 React 对组件的处理流程。

> 对于类组件的执行，是在react-reconciler/src/ReactFiberClassComponent.js中：

```
function constructClassInstance(
    workInProgress, // 当前正在工作的 fiber 对象
    ctor,           // 我们的类组件
    props           // props
){
     /* 实例化组件，得到组件实例 instance */
     const instance = new ctor(props, context)
}
```

> 对于函数组件的执行，是在react-reconciler/src/ReactFiberHooks.js中

```
function renderWithHooks(
  current,          // 当前函数组件对应的 `fiber`， 初始化
  workInProgress,   // 当前正在工作的 fiber 对象
  Component,        // 我们函数组件
  props,            // 函数组件第一个参数 props
  secondArg,        // 函数组件其他参数
  nextRenderExpirationTime, //下次渲染过期时间
){
     /* 执行我们的函数组件，得到 return 返回的 React.element对象 */
     let children = Component(props, secondArg);
}
```

从中，找到了执行类组件和函数组件的函数。那么为了搞清楚 React 底层是如何处理组件的，首先来看一下类和函数组件是什么时候被实例化和执行的？

在 React 调和渲染 fiber 节点的时候，如果发现 fiber tag 是 ClassComponent = 1，则按照类组件逻辑处理，如果是 FunctionComponent = 0 则按照函数组件逻辑处理。当然 React 也提供了一些内置的组件，比如说 Suspense 、Profiler 等。

三 二种不同 React 组件

1 class类组件

类组件的定义

在 class 组件中，除了继承 React.Component ，底层还加入了 updater 对象，组件中调用的 setState 和 forceUpdate 本质上是调用了 updater 对象上的 enqueueSetState 和 enqueueForceUpdate 方法。

那么，React 底层是如何定义类组件的呢？

> react/src/ReactBaseClasses.js

```
 function Component(props, context, updater) {
  this.props = props;      //绑定props
  this.context = context;  //绑定context
  this.refs = emptyObject; //绑定ref
  this.updater = updater || ReactNoopUpdateQueue; //上面所属的updater 对象
}
/* 绑定setState 方法 */
Component.prototype.setState = function(partialState, callback) {
  this.updater.enqueueSetState(this, partialState, callback, 'setState');
}
/* 绑定forceupdate 方法 */
 Component.prototype.forceUpdate = function(callback) {
  this.updater.enqueueForceUpdate(this, callback, 'forceUpdate');
}
```

如上可以看出 Component 底层 React 的处理逻辑是，类组件执行构造函数过程中会在实例上绑定 props 和 context ，初始化置空 refs 属性，原型链上绑定setState、forceUpdate 方法。对于 updater，React 在实例化类组件之后会单独绑定 update 对象。

| --------问与答--------- |

问：如果没有在 constructor 的 super 函数中传递 props，那么接下来 constructor 执行上下文中就获取不到 props ，这是为什么呢？

```
/* 假设我们在 constructor 中这么写 */
constructor(){
    super()
    console.log(this.props) // 打印 undefined 为什么？
}
```

答案很简单，刚才的 Component 源码已经说得明明白白了，绑定 props 是在父类 Component 构造函数中，执行 super 等于执行 Component 函数，此时 props 没有作为第一个参数传给 super() ，在 Component 中就会找不到 props 参数，从而变成 undefined ，在接下来 constructor 代码中打印 props 为 undefined 。

```
/* 解决问题 */
constructor(props){
    super(props)
}
```

| ---------**end**---------- |

为了更好地使用 **React** 类组件，我们首先看一下类组件各个部分的功能：

```
class Index extends React.Component{
    constructor(...arg){
       super(...arg)                          /* 执行 react 底层 Component 函数 */
    }
    state = {}                                /* state */
    static number = 1                         /* 内置静态属性 */
    handleClick= () => console.log(111)       /* 方法：  箭头函数方法直接绑定在this实例上 */
    componentDidMount(){                      /* 生命周期 */
        console.log(Index.number,Index.number1) // 打印 1 , 2
    }
    render(){                                 /* 渲染函数 */
        return <div style={{ marginTop:'50px' }} onClick={ this.handerClick }
>hello,React!</div>
    }
}
Index.number1 = 2                             /* 外置静态属性 */
Index.prototype.handleClick = ()=> console.log(222) /* 方法: 绑定在 Index 原型链的 方法*/
```

上面把类组件的主要组成部分都展示给大家了。针对 state ，生命周期等部分，后续会有专门的章节进行讲解。

| --------问与答--------- |

问：上述绑定了两个 handleClick ，那么点击 div 之后会打印什么呢？

答：结果是 111 。因为在 class 类内部，箭头函数是直接绑定在实例对象上的，而第二个 handleClick 是绑定在 prototype 原型链上的，它们的优先级是：实例对象上方法属性 > 原型链对象上方法属性。

| ---------end---------- |

对于 `pureComponent` 会在 React 渲染优化章节，详细探讨。

2 函数组件

ReactV16.8 hooks 问世以来，对函数组件的功能加以强化，可以在 function 组件中，做类组件一切能做的事情，甚至完全取缔类组件。函数组件的结构相比类组件就简单多了，比如说，下面写了一个常规的函数组件：

```
function Index(){
    console.log(Index.number) // 打印 1
    const [ message , setMessage  ] = useState('hello,world') /* hooks  */
    return <div onClick={() => setMessage('let us learn React!')  } > { message } </div>
/* 返回值 作为渲染ui */
 }
 Index.number = 1 /* 绑定静态属性 */
```

注意：不要尝试给函数组件 prototype 绑定属性或方法，即使绑定了也没有任何作用，因为通过上面源码中 React 对函数组件的调用，是采用直接执行函数的方式，而不是通过new的方式。

那么，函数组件和类组件本质的区别是什么呢？

对于类组件来说，底层只需要实例化一次，实例中保存了组件的 **state** 等状态。对于每一次更新只需要调用 **render** 方法以及对应的生命周期就可以了。但是在函数组件中，每一次更新都是一次新的函数执行，一次函数组件的更新，里面的变量会重新声明。

为了能让函数组件可以保存一些状态，执行一些副作用钩子，React Hooks 应运而生，它可以帮助记录 React 中组件的状态，处理一些额外的副作用。

四 组件通信方式

React 一共有 5 种主流的通信方式：

1. props 和 callback 方式
2. ref 方式。
3. React-redux 或 React-mobx 状态管理方式。
4. context 上下文方式。
5. event bus 事件总线。

这里主要讲一下第1种和第5种，其余的会在对应章节详细解读。

① **props** 和 **callback** 方式

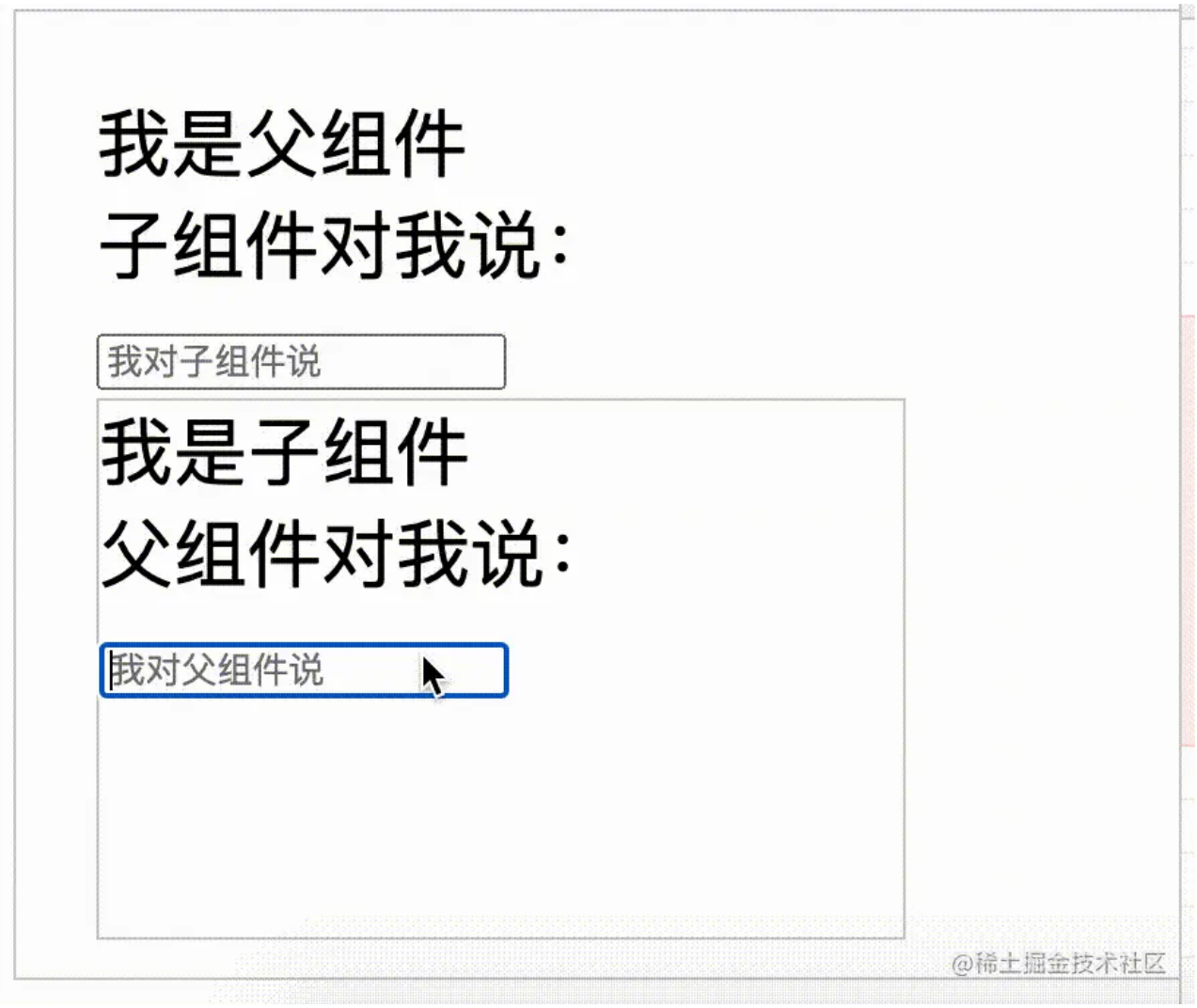
props 和 callback 可以作为 React 组件最基本的通信方式，父组件可以通过 props 将信息传递给子组件，子组件可以通过执行 props 中的回调函数 callback 来触发父组件的方法，实现父与子的消息通讯。

父组件 -> 通过自身 state 改变，重新渲染，传递 props -> 通知子组件

子组件 -> 通过调用父组件 props 方法 -> 通知父组件。

```
/* 子组件 */
function Son(props){
    const {  fatherSay , sayFather  } = props
    return <div className='son' >
         我是子组件
        <div> 父组件对我说：{ fatherSay } </div>
        <input placeholder="我对父组件说" onChange={ (e)=>sayFather(e.target.value) }   />
    </div>
}
/* 父组件 */
function Father(){
    const [ childSay , setChildSay ] = useState('')
    const [ fatherSay , setFatherSay ] = useState('')
    return <div className="box father" >
        我是父组件
       <div> 子组件对我说：{ childSay } </div>
       <input placeholder="我对子组件说" onChange={ (e)=>setFatherSay(e.target.value) }
/>
       <Son fatherSay={fatherSay}  sayFather={ setChildSay }  />
    </div>
}
```

效果



⑤**event bus**事件总线

当然利用 eventBus 也可以实现组件通信，但是在 React 中并不提倡用这种方式，我还是更提倡用 props 方式通信。如果说非要用 eventBus，我觉得它更适合用 React 做基础构建的小程序，比如 Taro。接下来将上述 demo 通过 eventBus 方式进行改造。

```
import { BusService } from './eventBus'
/* event Bus  */
function Son(){
    const [ fatherSay , setFatherSay ] = useState('')
    React.useEffect(()=>{
        BusService.on('fatherSay',(value)=>{  /* 事件绑定 , 给父组件绑定事件 */
            setFatherSay(value)
       })
       return function(){  BusService.off('fatherSay') /* 解绑事件 */ }
    },[])
    return <div className='son' >
         我是子组件
        <div> 父组件对我说: { fatherSay } </div>
        <input placeholder="我对父组件说" onChange={ (e)=>
BusService.emit('childSay',e.target.value)  }   />
    </div>
}
/* 父组件 */
function Father(){
    const [ childSay , setChildSay ] = useState('')
    React.useEffect(()=>{    /* 事件绑定 , 给子组件绑定事件 */
        BusService.on('childSay',(value)=>{
             setChildSay(value)
        })
        return function(){  BusService.off('childSay') /* 解绑事件 */ }
    },[])
    return <div className="box father" >
        我是父组件
       <div> 子组件对我说: { childSay } </div>
       <input placeholder="我对子组件说" onChange={ (e)=>
BusService.emit('fatherSay',e.target.value) }   />
       <Son  />
    </div>
}
```

这样做不仅达到了和使用 props 同样的效果，还能跨层级，不会受到 React 父子组件层级的影响。但是为什么很多人都不推荐这种方式呢？因为它有一些致命缺点。

- 需要手动绑定和解绑。
- 对于小型项目还好，但是对于中大型项目，这种方式的组件通信，会造成牵一发动全身的影响，而且后期难以维护，组件之间的状态也是未知的。
- 一定程度上违背了 React 数据流向原则。

五 组件的强化方式

①类组件继承

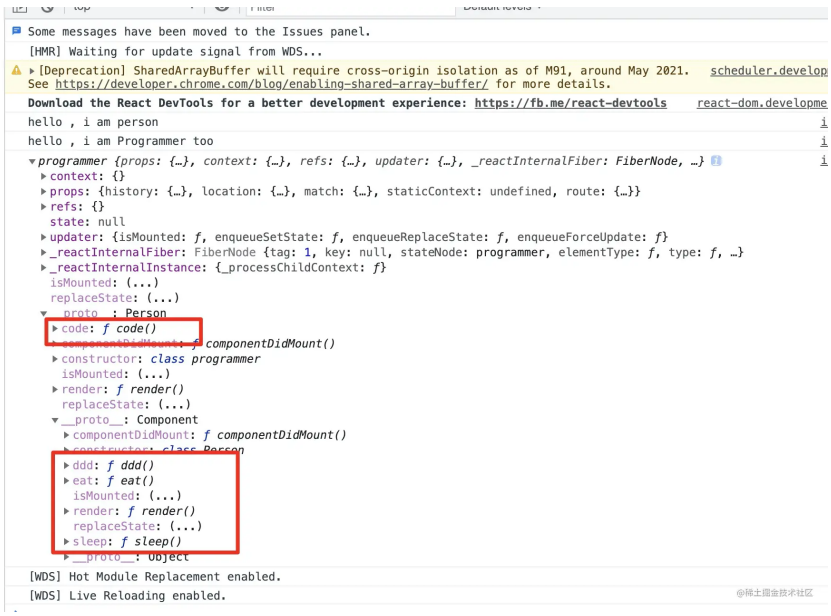
对于类组件的强化，首先想到的是继承方式，之前开发的开源项目 react-keepalive-router 就是通过继承 React-Router 中的 Switch 和 Router ，来达到缓存页面的功能的。因为 React 中类组件，有良好的继承属性，所以可以针对一些基础组件，首先实现一部分基础功能，再针对项目要求进行有方向的改造、强化、添加额外功能。

基础组件：

```
/* 人类 */
class Person extends React.Component{
    constructor(props){
        super(props)
        console.log('hello , i am person')
    }
    componentDidMount(){ console.log(1111)  }
    eat(){    /* 吃饭 */ }
    sleep(){  /* 睡觉 */  }
    ddd(){   console.log('打豆豆')  /* 打豆豆 */ }
    render(){
        return <div>
            大家好，我是一个person
        </div>
    }
}
/* 程序员 */
class Programmer extends Person{
    constructor(props){
        super(props)
        console.log('hello , i am Programmer too')
    }
    componentDidMount(){  console.log(this)  }
    code(){ /* 敲代码 */ }
    render(){
        return <div style={ { marginTop:'50px' } } >
            { super.render() } { /* 让 Person 中的 render 执行 */ }
            我还是一个程序员！     { /* 添加自己的内容 */ }
        </div>
    }
}
export default Programmer
```

效果：

大家好，我是一个person
我还是一个程序员！



我们从上面不难发现这个继承增强效果很优秀。它的优势如下：

1. 可以控制父类 render，还可以添加一些其他的渲染内容；
2. 可以共享父类方法，还可以添加额外的方法和属性。

但是也有值得注意的地方，就是 state 和生命周期会被继承后的组件修改。像上述 demo 中，Person 组件中的 componentDidMount 生命周期将不会被执行。

②函数组件自定义 **Hooks**

在自定义 hooks 章节，会详细介绍自定义 hooks 的原理和编写。

③**HOC**高阶组件

在 HOC 章节，会详细介绍高阶组件 HOC 。

六 总结

从本章节学到了哪些知识：

- 知道了 React 组件本质——UI + update + 常规的类和函数 = React 组件 ， 以及 React 对组件的底层处理逻辑。
- 明白了函数组件和类组件的区别。
- 掌握组件通信方式。
- 掌握了组件强化方式。

下一章节，我们将走进 React 状态管理 state 的世界中，一起探讨 State 的奥秘。

留言

全部评论（118）

加载中...