day02-props进阶-生命周期

1. Props进阶

### children属性

**props.children**

React的文档提到“children是一个不透明的数据结构”。从本质上来讲， props.children 可以是任何的类型，比如数组、函数、对象等等.

React提供了一系列的函数助手来使得操作children更加方便.

**循环**

React.Children.map( this.props.children, (item,index)=>{} )

React.Children.forEach( this.props.children, (item,index)=>{} )

**计数**

React.Children.count( this.props.children )

### props类型验证

有时在组件之间进行传递数据时需要对数据的类型进行校验，react 中提供了类型检查的模块 prop-types 通过该模块可以对传递的数据进行类型检测，出于性能方面的考虑，propTypes 仅在开发模式下进行检查.

使用时首先要导入该模块 : import PropTypes from 'prop-types'

导入完成后，为接收数据的组件设置类型检查：

myComponent.propTypes = {

optionalArray: PropTypes.array, // 检查 optionalArray 的值是否为数组

optionalBool: PropTypes.bool, // 检查 optionalBool 的值是否为布尔类型

}

### props默认值

**组件外指定组件默认属性**

组件外指定组件默认属性 Com.defaultProps = {}

Header.*defaultProps* = {  
 id: **'1000'**,  
 url: **'file://'**}**;**

**组件内指定组件默认属性**

组件内指定组件默认属性,需要添加 static 修饰符

class Header extends React.Component{  
 //在组件内指定 默认属性,需要添加 static 修饰符  
 static *defaultProps* = {  
 id:'1000',  
 url:'file://'  
 };  
 render(){  
 return (  
 <div className="header">  
 {/\* 组件模板中使用 组件的属性 \*/}  
 <h1>{this.props.id},{this.props.url}</h1>  
 </div>  
 )  
 }  
}

1. 生命周期

以下生命周期指的是Class组件的生命周期方法.

### 实例化期

constructor:在实例化时调用1次

componentWillMount : 在渲染前调用

getDerivedStateFromProps : 当state需要从props初始化时使用, 每次render都会调用, 尽量不使用，维护俩者状态需要消耗额外资源，增加复杂度

render : 每个React组件内都会有render函数，该函数会在每次渲染的时候执行，返回一个React元素；在之后的过程中，会根据React元素生成的虚拟DOM进行DOM操作

componentDidMount : 在第一次渲染后调用，只在客户端。之后组件已经生成了对应的DOM结构，可以通过this.getDOMNode()来进行访问。 如果你想和其他JavaScript框架一起使用，可以在这个方法中调用setTimeout, setInterval或者发送AJAX请求等操作(防止异步操作阻塞UI)

### 存在期( 更新期 )

componentWillReceiveProps ( 已废弃 )

初始渲染的时候不会执行, 那也就是说当一个组件被第二次元素化的时候会执行，那么第二次元素化时传递的props很有可能和第一次是一样的.

componentWillReceiveProps(nextProps){

console.log(this.props) // 之前的属性

console.log(nextProps) // 新的props

}

shouldComponentUpdate

该周期函数发生在视图将要更新前，并且可以让我们自己决定视图是否要真的更新，在**初次渲染的时候，不会调用该函数**；我们可以通过在该函数的末尾返回布尔类型的值来决定是否需要重新渲染，如果返回true，那就正常渲染，如果返回false，那么就不渲染，并且会直接跳过render,componentWillUpdate以及componentDidUpdate阶段；

并且该周期函数会接收 新的属性(nextProps)和新的状态(nextState)作为参数;

React.createClass({

shouldComponentUpdate(nextProps,nextState){

// ……

return true/fales;

}

});

componentWillUpdate

该函数在你的组件一出现渲染的时候就会被调用。你无法在该函数中使用setState。该函数也会接收新的属性和状态作为参数

React.createClass({

componentWillUpdate (nextProps,nextState){

// ……

}

});

getSnapshotBeforeUpdate : 在render之前调用，state已更新, 主要用于获取render之前的dom状态.

componentDidUpdate

该函数仅在DOM的所有渲染更新被处理完后被执行。因为此时已经是变更后的状态了，所以在该函数内部访问的this.props和this.state是新的属性和状态，该函数会接收上一次的属性和状态作为参数；

React.createClass({

componentDidUpdate (prevProps,prevState){

// ……

}

});

### 销毁期

componentWillUnmount

把组件渲染在DOM上时，叫作配备(挂载)。组件不再在DOM上配备的时候(卸载)，该函数立马被调用.

1. refs与DOM

react的思想是通过更加节省性能的虚拟DOM去管理真实的DOM更新，一般情况下，我们避免了DOM操作，因为react在底层已经帮我们做了；但是，有些时候我们确实需要真实的DOM操作，react给我们提供了方法.

### findDOMNode

findDOMNode接收组件实例，返回该组件所对应的DOM节点

import {findDOMNode} from ‘react-dom’ //导入该方法

let realDom = findDOMNode(组件实例)

### unmountComponentAtNode

import {unmountComponentAtNode} from ‘react-dom’

该方法可以让我们卸载由React加载的所有组件，需要提供一个DOM容器，如下

unmountComponentAtNode(container)

注意：container必须是我们挂载react应用的dom节点；

在 React 18 中，unmountComponentAtNode 已被 root.unmount() 取代。

### refs

**ref的值是一个字符串.**

<h1 ref=”hh”></h1>

获取真实的DOM节点: this.refs.hh

**ref的值为一个函数**

<h1 ref={(h)=>{console.log(h);//h就是h1这个真实dom节点}}></h1>

**ref应用在一个组件元素上**

<HelloWorld ref={string/回调} />

如上述代码，此时ref帮我们获取的就不再是dom节点了，而是组件实例

1. 综合案例

<cdf-会员购> 店铺首页-今日值得买(秒杀)

1. 作业

作业一:

<cdf-会员购> 全部商品页-商品排序(综合排序,销量排序,价格排序)

1.用class组件 + jsx语法 开发组件;

2.使用Mockjs模拟商品数据, 在合适的生命周期方法中请求数据;

3.页面样式要100%还原.

4.封装头部输入组件 MyHeader;

5.封装商品列表组件MyList;

6.封装底部MyTabbar组件;

7.实现点击综合/销量/新品/价格 降序排序;

8.实现点击综合/销量/新品/价格 按钮高亮且只有一个高亮(红色);

作业二:

<cdf-会员购> 分类页-选项卡切换

1.用class组件 + jsx语法 开发组件;

2.使用Mockjs模拟分类数据, 在合适的生命周期方法中请求数据;

3.页面样式要100%还原.

5.封装选项卡组件MyTab;

6.封装底部MyTabbar组件;

7.实现点击左侧一级分类按钮, 动态渲染右侧二级分类;

8.实现点击左侧一级分类按钮, 按钮高亮且只有一个高亮(红色);