https://zh-hans.reactjs.org/docs/getting-started.html

（一）Hello World

（二）JSX 简介

（1）使用{}嵌入单行表达式

（2）引号将属性值指定为字符串字面量

（3）使用camelCase（小驼峰命名）定义属性名，不使用html属性名的命名约定。

（4）类样式class变成了className

（5）如果标签children没有内容，可以使用 /> 闭合标签

（6）插入内容在渲染之前会被转换成了字符串。可以防止 XSS（cross-site-scripting, 跨站脚本）攻击

（7）babel（巴）把jsx转译成React.createElement()函数调用，创建 “react 元素”对象。构建虚拟dom保持更新

（三）元素渲染

（1）<div id="root"></div>根节点内的内容由React管理，渲染用ReactDOM.render(element, document.getElementById('root'));

（四）组件 & Props

（1）分为函数组件和类组件

（2）原生DOM标签名称以小写字母开头，组件名称以大写字母开头

（3）props（破）不能修改

（五）State & 生命周期

（1）可以向类组件实例中添加不参与数据流的自定义类组件实例属性

componentDidMount() {

this.timerID = setInterval(

() => this.doSomething(),

1000

);

}

componentWillUnmount() {

clearInterval(this.timerID);

}

（2）使用

this.setState({

key：value,

});

更新state

（3）性能考虑，React可能把多个setState合并成一个调用

（4）解决this.props和this.state可能异步更新问题：

this.setState((state, props) => ({

key: state.key + props.key

}));

（六）事件处理

（1）原生DOM事件与React事件区别

（1.1）原生DOM事件名称纯小写，React事件名称小驼峰式（camelCase）

（1.2）原生DOM事件处理函数传入字符串"方法名()"，React事件处理函数传入函数

（1.3）原生DOM事件阻止默认行为可以使用返回false的方式，React事件阻止默认行为不能使用返回false的方式，必须使用e.preventDefault();

（2）js中，class方法默认不会绑定this，绑定方法有：

（2.1） onClick={this.方法名.bind(this)}

方法名(){

}

（2.2） onClick={this.方法名} class fields 语法

方法名= () => {

}

（2.3） onClick={() => this.方法名()} 缺点：每次渲染都会创建不同的回调函数

方法名(){

}

（3）传递参数

（3.1） onClick={this.方法名.bind(this,...param)}

方法名(...param,e){

}

（3.2） onClick={this.方法名}

方法名= (e) => {

}

（3.3） onClick={(e) => this.方法名(...param,e)}

方法名(...param,e){

}

（七）条件渲染

（1）使用if

（2）

false或数字0 && expression返回 false

true && expression返回 expression

（3）condition ? expressionTrue : expressionFalse

（4）render方法return null; （闹）不进行渲染

（八）列表 & Key

（1）数组中项默认用索引做key，应用唯一的字符串做key

（九）表单

（1）受控组件：表单内部state与组件state相结合

<form onSubmit={this.onSubmit}>

<input value={this.state.value} onChange={this.onChange} type="text" />

<input type="submit" value="提交" />

</form>

其他：

<textarea value={this.state.value} onChange={this.onChange} />

<select value={this.state.value} onChange={this.onChange} >

<option value="a">a</option>

<option value="b">b</option>

</select>

<select value={this.state.value} onChange={this.handleChange} multiple={true} >

<option value="a">a</option>

<option value="b">b</option>

</select>

this.state.value是['a', 'b']形式

（2）非受控组件：表单内部state不与组件state相结合

<input type="file" /> 因其value只读

（十）状态提升：将多个组件中需要共享的state向上移动到它们的最近共同父组件中，实现共享state

（十一）组合vs继承

（1）组件间的渲染内容通过props.children传递进组件

（2）渲染内容也可以通过属性传递进组件

（十二）React 哲学

（十三）无障碍辅助功能

（十四）代码分割

（1）动态导入方法

import("./文件名").then(文件名 => {

console.log(文件名.文件名中方法名());

});

（2）动态导入组件

（2.1）

import React, { lazy,Suspense } from 'react';

const 组件名 = lazy(() => import('./组件名'));

<Suspense fallback={<div>Loading...</div>}>

<组件名 />

</Suspense>

（2.2）lazy目前只支持默认导出（default exports）

（十五）Context（上）

（1）

const NameContext = React.createContext(defaultValue); defaultValue可以是任意数据类型

<NameContext.Provider value={value}>

<该组件及子组件使用距其最近的NameContext.Provider提供的value />

</NameContext.Provider>

使用Object.is检测value变化。value变化时，使用的组件会重渲染且不受制于shouldComponentUpdate或React.memo

class 该组件及子组件 extends React.Component {

在方法中通过this.context使用value

}

该组件及子组件.contextType = NameContext;

（2）函数组件和一个组件使用多个context情况使用

<NameContext.Consumer>

{value => 基于context值进行渲染}

</NameContext.Consumer>

（3）NameContext.displayName=""; React DevTools显示NameContext的名称

（十六）错误边界

（1）错误边界是一种React组件，可以捕获并打印发生在其子组件树任何位置的JavaScript错误，并渲染备用UI

（2）错误边界无法捕获：

（2.1）事件处理（使用try catch捕获）

（2.2）异步代码

（2.3）自身错误

（2.4）服务端渲染

（3）只有class组件中定义getDerivedStateFromError(渲染备用UI)或componentDidCatch(打印错误)至少一个，即为错误边界

（4）自React 16起，未被错误边界捕获的错误会导致整个React组件树被卸载

（十七）ref 转发

（1）

let 父组件名 = React.forwardRef((props, ref) => {

return <使用ref ref={ref} />;

});

const ref = React.createRef();

<父组件名 ref={ref}></父组件名>

（2）创建ref后，可通过非ref名的props向子组件传递该ref

（3）

（3.1）return外层默认方法，DevTools中显示ForwardRef

（3.2）return外层name方法，DevTools中显示ForwardRef(name)

（3.3）return外层name方法

name.displayName="newName"，DevTools中显示ForwardRef(newName)

（十八）Fragments（片）

（1）返回多个同级，外层可用<React.Fragment></React.Fragment>或<> </>

（2）<React.Fragment>只支持key，<>不支持key

（十九）高阶组件（HOC）

（1）高阶组件是函数，参数中有组件，返回新组件。解决横切关注点问题

（2）不要在render中使用高阶组件。每次返回新组件，diff算法均判定不同，会卸载原组件并重新挂载新组件，导致状态丢失。

（3）新组件不会有参数中有组件的静态方法。如果要

（3.1）将参数中有组件的静态方法复制到新组件

（3.2）单独导出参数中有组件的静态方法

（十九）与第三方库协同

（1）操作无需更新的React元素，如一个空<div></div>

（二十）深入 JSX

（1）点语法，如antd的<DatePicker.RangePicker></DatePicker.RangePicker>

（2）在运行时选择类型

const component = {

componentKeyA: ComponentValueA,

componentKeyB: ComponentValueB,

};

<components["componentKeyA"] />;

错误！JSX 类型不能是一个表达式

const component = {

componentKeyA: ComponentValueA,

componentKeyB: ComponentValueB,

};

const Component = components["componentKeyA"];

<Component />;

正确！JSX 类型可以是大写字母开头的变量

（3）字符串字面量赋值给props时，值未转义。下面相等

<Component propsName="&lt;&gt;" />

<Component propsName={'<>'} />

（4）props默认值为True。下面相等

<Component propsName />

<Component propsName={true} />

（5）props属性展开

<Component {...props} />

（6）布尔类型、Null、Undefined会忽略。下面相等

<div></div>

<div>{true}</div>

<div>{false}</div>

<div>{null}</div>

<div>{undefined}</div>

（二十一）性能优化

（1）计算某组件是否重渲染，顺序：

（1.1）shouldComponentUpdate返回false该组件及子组件不重新渲染

（1.2）shouldComponentUpdate返回true，虚拟dom相同，该组件及子组件不重新渲染

（1.3）shouldComponentUpdate返回true，虚拟dom不同，该组件重新渲染，计算子组件是否重渲染

（2）shouldComponentUpdate默认返回true

（3）React.PureComponent浅比较当前与之前props与state实现shouldComponentUpdate

（4）浅比较为引用相等，深比较为原值相等

（二十二）Portals

（1）ReactDOM.createPortal(component, container) 将子组件渲染到父组件外的dom节点

（2）portal表现在dom树，结构在React树，其行为与普通子组件行为一致

（二十三）Profiler

（1）Profiler能放在React树中的任何地方测量这部分渲染带来的开销，需要两个props

<Profiler id="id" onRender={onRender}>

<Component />

</Profiler>

（二十四）不使用 ES6

（二十五）不使用 JSX

（二十六）协调

（1）diff算法O(n)

（2）diff算法实现

（2.1）元素类型不同，卸载原来虚拟dom建立新的虚拟dom

（2.2）元素类型相同，更新虚拟dom

（2.3）列表元素，根据key更新虚拟dom

（二十七）ref & DOM

（1）适合使用ref情况：

（1.1）管理焦点，文本选择或媒体播放

（1.2）触发强制动画

（1.3）集成第三方 DOM 库

（2）创建ref

class 类组件名 extends React.Component {

constructor(props) {

super(props);

this.ref名 = React.createRef();

}

render() {

return <div ref={this.ref名} />;

}

}

（3）访问ref

const 访问节点名 = this.ref名.current;

（3.1）ref用于原生DOM标签，访问节点为该原生DOM标签

（3.2）ref用于类组件，访问节点为该类组件实例

（3.3）ref不能用于函数组件，因为其没有实例

（4）回调ref

class 类组件名 extends React.Component {

constructor(props) {

super(props);

this.访问节点名 = null;

this.回调函数名 = 访问节点名 => {

this.访问节点名 = 访问节点名;

};

}

render() {

return <div ref={this.回调函数名} />;

}

}

（5）关于回调 ref 的说明

如果回调ref是内联函数，更新时会执行两次，第一次传入null，第二次传入访问节点名。因为每次更新时会创建新的内联函数实例，会清空旧的ref并设置新的ref

（二十八）Render Props（润）

<组件名 props名={函数返回渲染}/>

（二十九）静态类型检查

（1）Flow和TypeScript等静态类型检查器，可在运行前识别类型问题。建议在大型代码库中使用Flow或 TypeScript来代替PropTypes。

（三十）严格模式

（1）

<React.StrictMode>

<该组件及子组件问题检查和警告 />

</React.StrictMode>

（2）严格模式在开发模式运行，在生产构建不运行

（3）严格模式故意重复调用方法发现问题

（三十一）使用 PropTypes 进行类型检查

（1）

import PropTypes from 'prop-types';

组件名.propTypes = {

props名: props验证

};

（2）props验证包括：

（2.1）PropTypes.array（组）

（2.2）PropTypes.bool（布）

（2.3）PropTypes.func（方）

（2.4）PropTypes.number（数）

（2.5）PropTypes.object（对）

（2.6）PropTypes.string（串）

（2.7）PropTypes.symbol（符）

（2.8）PropTypes.node（节） 可被渲染的元素

（2.9）PropTypes.element（元） React元素

（2.10）PropTypes.elementType（类） React元素类型

（2.11）PropTypes.instanceOf(类名)（实） 类实例

（2.12）PropTypes.oneOf([...值])（一） 枚举值中之一

（2.13）PropTypes.oneOfType([...类型])（类） 枚举类型中之一

（2.14）PropTypes.arrayOf(类型)（全） 数组全是该类型

（2.15）PropTypes.objectOf(类型)（全） 对象全是该类型

（2.16）PropTypes.shape({（形） 对象

键名: 类型

...

}),

（2.17）PropTypes.exact({（正） 正好是描述的对象

键名: 类型

...

}),

（2.18）PropTypes.any（任） 任意

（2.19）PropTypes.属性后加isRequired必须提供

（2.20）自定义验证

function(props, propsName, componentName) {

if (正则匹配) {

return new Error("验证失败信息");

}

}

（2.21）arrayOf或objectOf自定义验证（全）

PropTypes.arrayOf或objectOf(function(props, propsName, componentName, location, propFullName) {

if (正则匹配) {

return new Error("验证失败信息");

}

})

（3）children也可以被验证（子）

（4）默认Props

组件名.defaultProps = {

props名: 默认值

};

（5）验证发生在默认Props之后

（三十二）非受控组件

（三十三）Web Components

（三十四）React 顶层 API

（1）React.memo（记）

（1.1）const 高阶函数组件 = React.memo(函数组件);

浅比较当前与之前props，相同不重新渲染，不同重新渲染

（1.2）const 高阶函数组件 = React.memo(函数组件,自定义比较当前与之前props的函数);

自定义比较当前与之前props的函数(prevProps, nextProps){

//相同返回true，不同返回false

}

（2）React.cloneElement 克隆并返回新的React元素

（3）React.isValidElement(object) 验证对象是否为 React 元素

（4）React.Children

（4.1）React.Children.map(children, function[(thisArg)]) （图）遍历每一项，返回数组

如果children是Fragment，视为单一子节点，不会被遍历。

（4.2）React.Children.forEach(children, function[(thisArg)]) （为）遍历每一项，不返回数组

如果children是Fragment，视为单一子节点，不会被遍历。

（4.3）React.Children.count(children) 返回children中的子节点数量

（4.4）React.Children.only(children) 验证children是否只有一个子节点

（4.5）React.Children.toArray(children)

将children以数组的方式扁平展开并返回，并为每个子节点分配一个key

（三十五）React.Component

（1）类组件的生命周期

（1.1）挂载

constructor()

static getDerivedStateFromProps()

render()

componentDidMount()

（1.2）更新

static getDerivedStateFromProps()

shouldComponentUpdate()

render()

getSnapshotBeforeUpdate()

componentDidUpdate()

（1.3）卸载

componentWillUnmount()

（1.4）错误处理

static getDerivedStateFromError()

componentDidCatch()

（2）render()

（2.1）返回以下类型之一:

React元素

Fragments（片）

Portals（泡）

数组

字符串

数值

布尔

null（闹）

（2.2）render()应为纯函数

（3）constructor(props)

（3.1）其他语句前调用super(props)，否则this.props未定义

（4）componentDidMount()

（4.1）可进行网络请求或订阅

（5）componentDidUpdate(prevProps, prevState, snapshot)

（5.1）可进行网络请求或订阅

（5.2）getSnapshotBeforeUpdate()的返回值为第三个参数snapshot

（6）componentWillUnmount()

（6.1）可进行取消订阅

（7）shouldComponentUpdate(nextProps, nextState)

（7.1）返回值默认为 true

（8）static getDerivedStateFromProps(nextProps, prevState)

（8.1）返回对象更新state，若返回null不更新state

（8.2）静态方法，无法访问组件实例

（9）getSnapshotBeforeUpdate(prevProps, prevState)

（9.1）更新前记录信息，如滚动位置

（10）static getDerivedStateFromError(error)

（10.1）渲染阶段调用，不允许出现副作用

（10.2）返回对象更新state，根据该对象渲染备用UI

（11）componentDidCatch(error, info)

（11.1）提交阶段调用，允许出现副作用

（12）forceUpdate()

（12.1）强制使该组件从render()方法开始调用

（三十六）ReactDOM

（三十七）ReactDOMServer

（三十八）DOM 元素

（1）插入html并防止跨站脚本（XSS）攻击

<div dangerouslySetInnerHTML={{\_\_html: 插入html}}>

</div>

（2）htmlFor替代for

（3）style={样式对象}

（4）suppressContentEditableWarning

使用contentEditable会发出警告，suppressContentEditableWarning会禁止该警告

（三十九）合成事件

（1）事件处理函数返回false不阻止冒泡。可以用e.stopPropagation()或者e.preventDefault()替代

（四十）Test Utilities

（四十一）Test Renderer

（四十二）JavaScript 环境要求

（四十三）术语表

（四十四）Hook 简介

（1）动机

（1.1）便于在组件之间复用状态逻辑

（1.2）将原来componentDidMount和componentDidUpdate中设置，componentWillUnmount中清除的两处逻辑，归为hook中一处逻辑

（1.3）使用函数组件替换类组件

（2）Hook在类组件内部不起作用

（四十五）Hook 概览

（四十六）使用 State Hook

（1）

import React, { useState } from 'react';

function 函数组件名() {

const [stateParam, setStateParam] = useState(defaultStateParam);

}

（2）读取state

Hook使用闭包机制将stateParam保存在函数组件作用域中，在函数组件作用域中可直接使用stateParam

（3）更新state

函数组件中使用setStateParam(新stateParam)替换原stateParam

（4）一个函数组件中可以使用多个useState

（四十七）使用 Effect Hook

（1）

import React, { useEffect } from 'react';

function 函数组件名() {

useEffect(() => {

});

}

类组件未提供每次渲染后执行的方法。

函数组件提供每次渲染后执行的方法。useEffect相当于componentDidMount和componentDidUpdate，区别在useEffect每次渲染后异步执行

（2）

import React, { useEffect } from 'react';

function 函数组件名() {

useEffect(() => {

return () => { 执行清除操作时调用，相当于componentWillUnmount

};

});

}

（3）一个函数组件中可以使用多个useEffect，按声明顺序调用

（4）对于一个有清除操作的useEffect

首渲染后执行本次effect

重渲染后执行上次清除操作，执行本次effect

销毁前执行上次清除操作

（5）useEffect第二个参数为数组

（5.1）执行该useEffect的依赖为数组中的参数，依赖改变，会重新执行该useEffect

（5.2）某个执行该useEffect的依赖未填写成数组中的参数，该依赖永远为初始值

（5.3）数组为空，仅首渲染后执行该useEffect，相当于componentDidMount

（四十八）Hook 规则

（1）

（1.1）在函数组件或自定义Hook中调用，不要在普通JavaScript函数中调用

（1.2）在最顶层调用，不要在条件、循环、嵌套函数中调用

（2）React能将内部state和对应的Hook进行关联，靠的是Hook的调用顺序

（四十九）自定义 Hook

（1）自定义Hook是一个函数，名称以use开头，函数内可调用其他的Hook

（2）多个组件中使用相同的Hook不会共享state

（五十）Hook API 索引

（1）const value = useContext(NameContext);

（2）useReducer

（2.1）

const [state, dispatch] = useReducer(reducer, initState);

（2.2）惰性初始化，仅首次渲染初始化

const [state, dispatch] = useReducer(reducer, initState,init);

function init(initState) {

return 计算后的initState

}

（2.3）使用Object.is检测state变化。state变化时，使用的组件会重渲染

（3）useCallback和useMemo和useEffect

（3.1）useCallback和useMemo与useEffect参数一致。区别是useCallback和useMemo不可以处理副作用，useEffect可以处理副作用

（3.2）useMemo和useCallback都在函数组件首渲染时执行，之后在其依赖的变量改变时执行。都返回缓存的值，useMemo返回变量，useCallback返回函数

（4）useRef

（4.1）const refObject = useRef(refObjectPointCurrentInitialValue);

refObject.current的初始值为refObjectPointCurrentInitialValue

refObject在函数组件整个生命周期不变

（4.2）ref.current只能设置dom节点。useRef.current可以设置任何值，类似于类组件实例变量

（4.3）自建{current: ...}对象，每次渲染返回不同对象。useRef每次渲染返回相同对象

（4.4）useRef.current发生变化不会通知，不会重渲染

（5）useImperativeHandle

（5.1）定义子组件暴露给父组件的可调用的useRef.current

（5.2）

let 父组件名 = React.forwardRef((props, ref) => {

const childRef = useRef();

useImperativeHandle(ref, () => ({

fatherFocus: () => {

childRef.current.focus();

}

}));

return <input ref={childRef} ... />;

});

<父组件名 ref={fatherRef} />

fatherRef.current.fatherFocus()。

（6）useLayoutEffect和useEffect

（6.1）useLayoutEffect和useEffect参数一致。区别是useLayoutEffect每次渲染后同步执行，useEffect每次渲染后异步执行

（7）useDebugValue

（7.1）在React开发者工具中该自定义hook旁显示

（7.2）useDebugValue(debugValue)

function 自定义hook名() {

const [stateParam, setStateParam] = useState(defaultStateParam);

//在React开发者工具中该自定义hook旁显示"自定义hook名: yes"

useDebugValue(stateParam ? 'yes' : 'no');

return stateParam;

}

（7.3）useDebugValue第二个参数为格式化函数，接受debugValue为参数，返回格式化后的值。该函数只在自定义hook被检查时调用。

useDebugValue(date,date => date.toDateString());

（五十一）Hooks FAQ

（1）16.8.0版本开始，支持Hook

（2）Hook不足

（2.1）Hook没有覆盖class的getSnapshotBeforeUpdate和getDerivedStateFromError和componentDidCatch生命周期

（2.2）一些第三方库不兼容Hook

（2.3）上一轮props或state需自己存useRef.current，Hook目前未提供存

（3）先点击alert，然后三秒内点击add。alert的结果是0不是1

function Example() {

const [count, setCount] = useState(0);

function onClickAlert() {

setTimeout(() => {

alert(count);

}, 3000);

}

return (

<div>

<p>{count}</p>

<button onClick={() => setCount(count + 1)}>

add

</button>

<button onClick={onClickAlert}>

alert

</button>

</div>

);

}

（4）实现getDerivedStateFromProps

function Example(newCount) {

const [count, setCount] = useState(0);

if (count !== newCount) {

setCount(newCount);

}

return (

<p>{count}</p>

);

}

（5）监听dom节点变化，使用callback ref

function Example() {

const [height, setHeight] = useState(0);

const callbackRef = useCallback(node => {

if (node !== null) {

setHeight(node.getBoundingClientRect().height);

}

}, []);

return (

<>

<h1 ref={callbackRef}></h1>

<h2>{height}</h2>

</>

);

}

（6）初次渲染设置定时器

function Example() {

const [count, setCount] = useState(0);

useEffect(() => {

const id = setInterval(() => {

setCount(c => c + 1);

}, 1000);

return () => clearInterval(id);

}, []); //

return <h1>{count}</h1>;

}

使用setCount(c => c + 1);不引用state，指定state如何改变

不用setCount(count + 1); 引用state，永远是初始值0

（7）惰性初始化，仅首次渲染初始化

（7.1）const [stateParam, setStateParam] = useState(createInitStateParam(props.initStateParam));

每次渲染都会调用createInitStateParam

（7.2）const [stateParam, setStateParam] = useState(() => createInitStateParam(props.initStateParam));

仅首次渲染会调用createInitStateParam