props.children

Component.defaultProps

## 网页中使用

在html文件底部添加

<script src="https://unpkg.com/react@16/umd/react.development.js" crossorigin></script>

<script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js" crossorigin></script>

<script src="index.js"></script>

前两个<script>标签加载 React。第三个加载组件代码。

react.development.js用于开发，react.production.min.js用于生产环境

亲手尝试了官方教程上所有添加JSX语法的方法，都不行。所以直接通过<script>引入的react就不能用JSX

## 创建单页应用

npx create-react-app my-app

cd my-app

npm start

npm run build

会在 build 文件夹内生成你应用的优化版本。

## 目录结构

my-app/

README.md

node\_modules/

package.json

public/

index.html

favicon.ico

src/

App.css

App.js

App.test.js

index.css

index.js

logo.sv

* index.html里<body>有一个<div id="root"></div>

## React元素

* 元素是构成 React 应用的最小砖块。
* React 元素可以通过ReactDOM.render(element, DOMElement)方法渲染到DOM 节点中
* React 元素是不可变对象。一旦被创建，你无法更改它的子元素或者属性。一个元素就像电影的单帧：它代表了某个特定时刻的 UI。也就是说，一旦通过

const ele = <h1>{name}</h1>;

这样的语句创建了React元素，里边的name变量就已字符串的形式固定下来，再不能通过改变name的值改变ele的值。想要改变渲染内容只能重新创建一个React元素然后重新渲染

* React元素有两种类型：1.形如<div></div>的DOM标签；2.用户自定义的React组件<Nihao name=”Tiancheng” />

## React 组件

### class组件

是React.Component的子类

import React from "react";

import ReactDOM from "react-dom";

class Nihao extends React.Component {

render() {

return <div>你好</div>

}

}

ReactDOM.render(<Nihao />, document.getElementById("root"));

* render()方法返回一个 React 元素
* 当组件的状态数据props、state改变时，组件会再次调用 render() 方法重新渲染。所以render()里的const也不恒定
* 每次组件更新时 render 方法都会被调用，但只要在相同的 DOM 节点中渲染，就仅有一个组件的 class 实例被创建使用。这就使得我们可以使用如 state 或生命周期方法等很多其他特性。

class Father extends React.Component{

render(){

return <Nihao name="TianCheng"/>

}

}

ReactDOM.render(<Father />, document.getElementById("root"));

render()返回值中可以有其他组件，形成父子关系

组件接收一些参数props，在render() 返回的视图中可以引用

<div> 你好 {this.props.name} </div>

* 在 React 应用中，数据通过 props 的传递，从父组件流向子组件。
* 这通常会被叫做“自上而下”或是“单向”的数据流。任何的 state 总是所属于特定的组件，而且只能影响树中“低于”它们的组件。
* 如果你把一个以组件构成的树想象成一个 props 的数据瀑布的话，那么每一个组件的 state 就像是在任意一点上给瀑布增加额外的水源，但是它只能向下流动
* 所有 React 组件都必须像纯函数一样保护它们的 props 不被更改，即组件中不能出现类似props.start=1这样的语句

return <Nihao name={this.state.n}/>

如果像这样传值，父组件的state改变之后，子组件的props.name属性也相应改变。应该是重新绘制了

除了render()，其他方法也可以返回一个视图，然后再组件类中渲染

renderSquare(i) {

return <Square value={i} />;

}

如上，renderSquare返回一个<Square>组件。并且没有加 ()

render()中可以调用

<div> {this.renderSquare(0)} </div>

constructor(props) {

super(props);

}

组建类constructor写法

super()之后已经完成了父组件给子组件的属性传值

### State

constructor(props) {

super(props);

this.state = {

value: null,

};

}

this.state 应该被视为一个组件的私有属性。

State 的更新可能是异步的，出于性能考虑，React 可能会把多个 setState() 调用合并成一个调用。也就是说，setState之后，并没有立即setState，而是等着多个setState一起调用，而这个时候state可能早就变了

因为 this.props 和 this.state 可能会异步更新，所以你不要依赖他们的值来更新下一个状态。例如不要这样更新

this.setState({

counter: this.state.counter + this.props.increment,

});

要解决这个问题，可以让 setState() 接收一个函数而不是一个对象。这个函数用上一个 state 作为第一个参数，props 做为第二个参数：

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

counter: state.counter + props.increment

}));

注意，箭头函数返回一个对象时要在对象外边加小括号，不然按代码块编译了

说实话，咱也不知道为什么这样就解决问题了

你setState调用之后不立即更新，而是等几秒更新，那还玩儿个屁啊，早尼玛耽误了

如果state是{a:1}, 那么调用this.setState({b:2})并不会把a这个属性去掉，而是变为{a:1, b:2}

不要直接操作state，用this.setState()方法

除了构造函数中，给state赋值必须用setSate()。构造函数中可以this.state={a:1}这样

componentDidMount() {

this.timerID = 1

}

尽管 this.props 和 this.state 是 React 本身设置的，且都拥有特殊的含义，但是其实你可以向 class 中随意添加不参与数据流的额外字段。

### 函数组件

function Compo1(props){

return <h1>nihao, {props.name}</h1>

}

const ele1=<Compo1 name="甜橙"></Compo1>

ReactDOM.render(ele1, document.getElementById('root'))

* 该函数是一个React组件，称为函数组件，本质上就是一个JavaScript函数
* 它接收唯一带有数据的props参数，返回一个 React 元素
* 此时，React 元素ele1为用户自定义组件Comp1
* ele1将 JSX 所接收的属性以及子组件转换为单个对象传递给组件，这个对象就是props
* 组件名称必须以大写字母开头。React 会将以小写字母开头的组件视为原生 DOM 标签。例如，<div /> 代表 HTML 的 div 标签，而 <Welcome /> 则代表一个组件
* 组件可以在其输出中引用其他组件

function CombineCompoo1(props){

return (

<div>

<Compo1 name='甜橙'></Compo1>

<Compo1 name='甜橙2'></Compo1>

<Compo1 name='甜橙3'></Compo1>

</div>

);

}

同理组件可以引用其他用React元素代表的组件

nihao=<Compo1 name=”tiancheng”></Compo1>

return (

<div>

<div>nihao</div>

{nihao}

<div>

)

## JSX

const element = <h1>Hello, world!</h1>;

JSX是一个 JavaScript 的语法扩展

JSX可以生成 React “元素”。element就是元素变量

JSX语法中，可以在大括号内放置任何有效的 JavaScript 表达式。例如，2 + 2，user.firstName

JSX必须有一个父元素包裹所有的元素，不能const e = <p></p><p></p>

JSX中，不推荐<div class="nihao></div>这样使用，因为JavaScript有class关键字。class应换成className。当然，纯HTML不能用className

const name = 'Josh Perez';

const element1 = <h1>Hello, {name}</h1>;

const element2 = (

<h1>

Hello, {formatName(user)}!

</h1>

);

建议将内容包裹在括号中，再把JSX 拆分为多行

通过使用引号，来将属性值指定为字符串字面量：

const element = <div tabIndex="0"></div>;

也可以使用大括号，在属性值中插入一个 JavaScript 表达式：

const element = <img src={user.avatarUrl}></img>;

或者

const element = <img src={user.avatarUrl} />;

Babel 会把 JSX 转译成一个名为 React.createElement() 函数调用。

以下两种代码完全等效：

const element = (

<h1 className="greeting">

Hello, world!

</h1>

);

const element = React.createElement(

'h1',

{className: 'greeting'},

'Hello, world!'

);

Babel 是一个工具链，主要用于将 ECMAScript 2015+ 版本的代码转换为向后兼容的 JavaScript 语法，以便能够运行在当前和旧版本的浏览器或其他环境中

## 条件渲染

function WarningBanner(props) {

if (!props.warn) {

return null;

}

return <div className="warning">Warning! </div>;

}

可以通过return null阻止渲染

## 生命周期

当组件第一次被渲染到 DOM 中的时候，挂载mount

当 DOM 中组件被删除的时候，卸载unmount

componentDidMount() {

}

componentDidMount() 方法会在组件已经被渲染到 DOM 中后运行。理解为angular里的 ngOnInit()

componentWillMount()

componentDidMount()

componentWillUpdate(object nextProps, object nextState)

componentDidUpdate(object prevProps, object prevState)

componentWillUnmount()

重新渲染不等于重新装载，重新render()时不触发componentWillUnmount()。只有组件从 DOM 中被移除时才会unmount

## 事件处理

class Comp1 extends React.Component{

constructor(props){

super(props);

this.hello=this.hello.bind(this);

}

hello(e){

console.log('hello');

this.setState({});

e.preventDefault();

}

render(){

return (

<div onClick={this.hello}>feafew</div>

)

}

}

* React事件处理和传统HTML有所不同。在传统HTML中，绑定事件形如

<div onclick=”hello()”>aaa</div>

而React 事件的命名采用小驼峰式onClick，而不是纯小写；使用 JSX 语法时需要传入一个函数作为事件处理函数，而不是一个字符串。传统HTML引号里hello带()，react没有引号，是{}，里边也不带括号

* 在 React 中不能通过返回false的方式阻止默认行为，必须显式使用preventDefault
* 只要handleClick方法中有类似this.setState()的语句，用到了this，就必须在constructor中设置this.handleClick = this.handleClick.bind(this);

因为在 JavaScript 中，class 的方法默认不会绑定 this。如果你忘记绑定 this.handleClick 并把它传入了 onClick，此时调用这个函数的时候 this 的值为 undefined。

除了绑定this，还有两种解决方法

* 方法1，如果你正在使用实验性的 public class fields 语法，你可以使用 class fields 正确的绑定回调函数：Create React App 默认启用此语法。

handleClick = () => {

console.log('this is:', this);

}

* 方法2，用箭头函数绑定事件

<button onClick={ () => this.handleClick() }>show</button>

因为箭头函数中的this指向声明时的this，而不是使用时的。官网不推荐这种

render(i) {

return (

<Child value={this.state.nihao}

onClick={() => this.handleClick(i)} />

);

}

注意，给组件注册onClick事件，这个onClick就有特殊意义了。不是整个组件点击任何部分都能触发，而是可以绑定给组件的元素。

其实吧，叫了个onClick，和叫value是一样的，都是props的属性

在子组件内部，可以使用props.onClick绑定事件处理方法

function Child(props){

<button onClick={props.onClick}>登录</button>

}

向事件处理程序传递参数

<button onClick={(e) => this.deleteRow(id, e)}>Delete Row</button>

## key

function Nihao (props){

const arr = [324, 32, 3214, 432, 432];

let items = arr.map( a => (<li>这里是{a}</li>))

return <ul>{items}</ul>

}

报Warning: Each child in a list should have a unique "key" prop

let items = arr.map( a => (<li key={a.toString()} >这里是{a}</li>))

给予每个列表项一个独一无二的key，warning消失

* key帮助 React 识别哪些元素改变了。应当给列表中的每一个元素赋予一个确定的标识。
* 一个元素的 key 最好是这个元素在列表中拥有的一个独一无二的字符串。通常使用数据中的 id 来作为元素的 key
* 当元素没有确定 id 的时候，可以使用元素索引 index 作为 key。但极力不推荐这么做
* 如果不指定显式的 key 值，那么 React 将默认使用索引用作为列表项目的 key 值。
* 一个好的经验法则是：在 map() 方法中的元素需要设置 key 属性。

function ListItem(props) {

return <li>{props.value}</li>;

}

function NumberList(props) {

const numbers = props.numbers;

const listItems = numbers.map((number) =>

<ListItem key={number.toString()} value={number} />

);

return (

<ul>

{listItems}

</ul>

);

}

注意，如果列表项是一个组件，应该把 key 保留在数组中的这个 <ListItem /> 元素上，而不是放在 ListItem 组件中的 <li> 元素上。

虽然给ListItem组件传递了key属性，但组件并不接收这个信息，React接收了。ListItem.props.key是读不出来的。如果需要请用其他属性显示传递这个值

## 表单

this.state = {

name: "woshjinizuzong"

};

this.aaa=this.aaa.bind(this);

aaa(e){

this.setState({

name: e.target.value

})

e.preventDefault();

}

render() {

return (

<input name="name" value={this.state.name} onChange={this.aaa}></input>

);

}

* 把<input>标签的value属性绑定this.state中的属性
* 此时运行会警告此组件是只读的。必须设置onChange事件回调或readOnly属性
* <input>标签onChange事件绑定回调函数。当然不要忘了在constructor中设置bind(this)
* 在 React 里，表单元素的工作方式和纯HTML里不同。HTML里给<input>设置value属性后可以输入任意数据，<input>的value随之改变。React里如果只给<input>设置value={this.state.name}则不能输入，不能改变<input>的value。React中表单状态通过state更新，state是唯一数据源，表单输入元素就叫“受控组件”。也可以不通过state更新，绑定一个变量this.a1也行。

## 状态提升

这一章教程举了一个很长很长的例子，用摄氏度和华氏度的同步转换，告诉了大概以下几个道理

* 只要组件的state和props发生改变，不管render()返回值中有没有用到{this.state}或{this.props}，都要重新绘制一遍DOM。所以形如

render(){

const x = this.state.name;

return <div>{x}</div>

}

在pros或state发生改变后，不管发生了什么改变，x值有没有变化，都要重新绘制一遍。当然如果state.name发生变化，就会将结果反映到DOM中

* 并列的子组件之间不能直接通信，必须通过状态提升，把值传给父组件，然后在父组件中处理，再将结果返回给子组件。

将值传给父组件也不能直接传递，必须通过把父组件的方法绑定到子组件的属性当中，然后利用子组件调用父组件传来的函数，将变化后的值以参数形式传回父组件，这样父组件就接收到了值的改变。举个例子

//父组件中

changeMethod(newValue){

this.setState({ name: newValue });

}

<Child method={this.changeMethod}</Child>

//子组件中

changeMethod(e){

this.props.method(e.target.value)

}

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

这样，子组件input改变一次值，父组件state.name就跟着变化一次

* .js文件中可以直接编写函数，相当于全局作用域。class里也可以直接用

## children prop

* 在以上内容中，使用组件时就是单个使用，然后加几个props，组件内部是空的

function WelcomeDialog(){

return <FancyBorder color='blue' />

}

* 可以使用一个特殊的 children prop ，通过父组件，将子组件的内容传递给子组件

function FancyBorder(props) {

return (

<div className={'FancyBorder FancyBorder-' + props.color}>

{props.children}

</div>

);

}

function WelcomeDialog() {

return (

<FancyBorder color="blue">

<h1 className="Dialog-title">Welcome</h1>

<p className="Dialog-message">Thank you for visiting!</p>

</FancyBorder>

);

}

这样，如果子组件预先不知道组件中的内容是什么，这种方法就可以解决问题。

通过JSX嵌套，可以将任意组件作为子组件的子组件传给子组件

如果组件中未知的内容比较多，而且渲染位置也不在一块，就可以不用children prop，用普通的prop，来传递内容

function SplitPane(props) {

return (

<div className="SplitPane">

<div className="SplitPane-left">

{props.left}

</div>

<div className="SplitPane-right">

{props.right}

</div>

</div>

);

}

function App() {

return (

<SplitPane left={<Contacts />} right={<Chat />} />

);

}

组件可以作为普通prop传给组件，这样就形成了父子关系的组件。而以前的父子组件声明比较直接

{ }里可以写React元素，形如<Nihao />

## Context

Context 主要应用场景在于很多不同层级的组件需要访问同样的数据。

如果App组件有一个子组件Toolbar，Toolbar有一个子组件ThemedButton。在App组件中设置theme="dark"，需要将dark以prop的方式传递给Toolbar，再以prop的方式传递给ThemedButton，即自上而下逐层传递。

而theme再App组件树中相当于是全局的数据，这样做显得繁琐。

const ThemeContext = React.createContext('light');

class App extends React.Component {

render() {

return (

<ThemeContext.Provider value="dark">

<Toolbar />

</ThemeContext.Provider>

);

}

}

function Toolbar() {

return (

<div>

<ThemedButton />

</div>

);

}

class ThemedButton extends React.Component {

static contextType = ThemeContext;

render() {

return <Button theme={this.context} />;

}

}

* React.createContext()方法为当前的 theme 创建一个 context对象
* ThemeContext对象有个Provider React组件，接受一个value属性，传递给使用value的组件，也叫消费组件，此例中是ThemedButton组件。亲测value可以是对象
* value发生变化时，内部所有消费组件重新渲染。创建时ThemeContext的值为light，传递时可以修改为“dark”。
* react里原生class自带一个contextType属性。现在需要把class上的contextType属性重赋值为Context对象。

一种方法是在class定义之外

ThemedButton.contextType = ThemeContext;

另一种方法就是上面这样，使用实验性的 public class fields 语法，static属性初始化

* 之后，就可以通过this.context在包括render()的各生命周期函数中访问value

如果value绑定的是{this.state}，那么state发生改变之后App重新渲染，消费组件也重新渲染，接收到的this.context随即发生改变

Context 可以使数据无须明确地传遍每一个组件，就能深入传递进组件树。

如果不用context也不想一层一层传递数据，我觉得一个全局变量即可解决。当然，这样增加了耦合，不利于组件复用

另一种无需 context 的解决方案是将ThemedButton组件自身传递下去，因而中间组件无需知道ThemedButton的props

const themeButton=<button className='dark'></button>

this.state={

themeButton: themeButton

}

render(){

return <Toolbar themeButton={this.state.themeButton} />

}

然后一层一层向下传递themeButton的值即可。不用知道themeButton里的具体内容，最后渲染时加上themeButton属性即可。不能不传递直接在最后一层直接使用，那样就接收不到。所以绝对不需要再中间件传值的只有context

## Refs

Refs 提供了一种访问 DOM 节点或在 render 方法中创建的 React 元素的方式。毕竟这种组件化的应用中，很难通过document.getElementById()得到DOM元素

class App1 extends React.Component{

constructor(props){

super(props);

this.inputRef=React.createRef();

this.getFocus=this.getFocus.bind(this);

this.inputChange=this.inputChange.bind(this);

}

getFocus(){

this.inputRef.current.focus();

}

inputChange(e){

this.nihao=e.target.value;

}

render(){

return (

<div>

<input type="text" value={this.nihao} ref={this.inputRef} onChange={this.inputChange}></input>

<input type="button" value="点击获取焦点" onClick={this.getFocus}></input>

</div>

)

}

}

* 使用 React.createRef() 创建Ref
* 通过 ref 属性把Ref附加到 React 元素。
* 通过Ref.current获取React元素的实例
* Ref感觉就是reference，引用的意思
* 上例效果：点击button后input获得焦点
* ref 的值根据节点的类型而有所不同：
* 当 ref 属性用于 HTML 元素时，ref接收底层 DOM 元素作为其 current 属性。
* 当 ref 属性用于自定义 class 组件时，ref 对象接收组件的挂载实例作为其 current 属性。
* 不能在函数组件上使用 ref 属性，因为他们没有实例。
* 如果App1有一个父组件，那么可以给App1绑定一个Ref，然后访问Ref.current. getFocus()方法

## 回调Refs

this.inputEle=null;

this.setInputRef= element => {

this.inputEle=element;

}

* 不通过React.createRef()创建ref，而是通过设置一个回调函数
* 将React元素的ref属性绑定回调函数
* 回调函数接受React元素作为参数，以使它们能在其他地方被存储和访问。函数内容就是将传入的参数绑定至一个变量inputEle
* inputEle获取React元素的实例，不在需要Ref.current
* React将在组件挂载时，调用ref回调函数并传入React元素，当卸载时调用它并传入 null。