# 类组件的事件

## 1.绑定事件

### 我们还是使用reactdemo3-components项目，在src下面新建一个目录：reactdemo3-components/src/02-class-component-events，然后在这个文件夹下面新建一个文件：ClassEvent.jsx,使用快捷键rcc+回车插件一个类组件

|  |
| --- |
| **import *React***, {Component} **from 'react'**;  **class** ClassEvent **extends** Component {  render() {  **return** (  <**div**>    </**div**>  );  } }  **export default** ClassEvent; |

### 我们在div标签里面添加个按钮，

|  |
| --- |
| **import *React***, {Component} **from 'react'**;  **class** ClassEvent **extends** Component {  render() {  **return** (  <**div**>  <**button**>按钮</**button**>  </**div**>  );  } }  **export default** ClassEvent; |

### 我们想给按钮添加一个点击事件，需要这么写，查看onClick的代码

|  |
| --- |
| **//reactdemo3-components/src/02-class-component-events/ClassEvent.jsx**  **import *React***, {Component} **from 'react'**;  **class** ClassEvent **extends** Component {  render() {  **return** (  <**div**>  <**button onClick=**{()=>{***window***.alert(**'hello'**)}}>按钮</**button**>  </**div**>  );  } }  **export default** ClassEvent; |

### 然后需要在App.jsx添加对这个类组件的引用

|  |
| --- |
| **import** { *useState* } **from 'react'** *// import reactLogo from './assets/react.svg' // import viteLogo from '/vite.svg' // import './App.css'* **import** *FuncComponent* **from "./01-compoment-basic/FuncComponent"**; **import** ClassComponent **from "./01-compoment-basic/ClassComponent"**; **import** ClassEvent **from "./02-class-component-events/ClassEvent"**;   **function** *App*() {    **return** (  <>  <**h1**>React demo3</**h1**>  <**FuncComponent**/>  <**ClassComponent**/>  <**ClassEvent**/>  </>  ) }  **export default** *App* |

### 运行效果：点击按钮会弹出对话框

|  |
| --- |
|  |

## 2.事件方法

### 这种写法不太好，我们改进一下

|  |
| --- |
| **import *React***, {Component} **from 'react'**;  **function** doClick() {  **return** () => {  ***window***.alert(**'hello from react'**)  }; }  **class** ClassEvent **extends** Component {  render() {  **return** (  <**div**>  <**button onClick=**{doClick()}>按钮</**button**>  </**div**>  );  } }  **export default** ClassEvent; |

### 可以怎么写

|  |
| --- |
| **import *React***, {Component} **from 'react'**;  **class** ClassEvent **extends** Component {  render() {  **let** doClick = **function** () {  **return** () => {  ***window***.alert(**'hello from react'**)  };  }  **return** (  <**div**>  <**button onClick=**{doClick()}>按钮</**button**>  </**div**>  );  } }  **export default** ClassEvent; |

### 或者这么写

|  |
| --- |
| **import *React***, {Component} **from 'react'**;  *// function doClick() { // return () => { // window.alert('hello from react') // }; // }* **class** ClassEvent **extends** Component {  doClick(){  **return** () => {  ***window***.alert(**'hello from react'**)  };  }   render() {  **return** (  <**div**>  <**button onClick=**{**this**.doClick()}>按钮</**button**>  </**div**>  );  } }  **export default** ClassEvent; |

### 还可以这么写

|  |
| --- |
| **import *React***, {Component} **from 'react'**;  **class** ClassEvent **extends** Component {    doClick2(){  ***console***.log(**'hello from react again'**)  }   render() {  **return** (  <**div**>  <**button onClick=**{**this**.doClick2}>按钮</**button**>  </**div**>  );  } }  **export default** ClassEvent; |

#### 注意：如果函数没有返回值，传递参数的时候不需要小括号

### 我们建议将这个函数改为箭头函数，这样子可以避免this引起的混乱

|  |
| --- |
| **import *React***, {Component} **from 'react'**; **class** ClassEvent **extends** Component {  doClick(){  **return** () => {  ***window***.alert(**'hello from react'**)  };  }  *doClick2*=()=>{  ***console***.log(**'hello from react again'**)  }   render() {  **return** (  <**div**>  <**button onClick=**{**this**.*doClick2*}>按钮</**button**>  </**div**>  );  } }  **export default** ClassEvent; |

### 运行一下，结果是一样的

|  |
| --- |
|  |

## 3.类组件的事件的传参

### 我们新建一个按钮，给他绑定一个箭头函数，在箭头函数里面调用类的函数标签菜单类的参与遍历name，这就是带参数调用

|  |
| --- |
| **import *React***, {Component} **from 'react'**;  *// function doClick() { // return () => { // window.alert('hello from react') // }; // }* **class** ClassEvent **extends** Component {  **name** = **'Harry berry'**  *doClick2*=()=>{  ***console***.log(**'hello from react again'**)  }  handleClick(name){  ***console***.log(**'hello '**+name);  }  render() {  **return** (  <**div**>  <**button onClick=**{**this**.*doClick2*}>按钮</**button**>  <**button onClick=**{()=>{  **this**.handleClick(**this**.**name**)  }}>按钮2</**button**>  </**div**>  );  } }  **export default** ClassEvent; |

### 运行效果

|  |
| --- |
|  |

### 还可以简写，为了方便查看结果，我们把name的值改为Jassica jade

|  |
| --- |
| **import *React***, {Component} **from 'react'**; **class** ClassEvent **extends** Component {  **name** = **'Jessica jade'**  *doClick2*=()=>{  ***console***.log(**'hello from react again'**)  }  handleClick(name){  ***console***.log(**'hello '**+name);  }  render() {  **return** (  <**div**>  <**button onClick=**{**this**.*doClick2*}>按钮</**button**>  <**button onClick=**{()=> **this**.handleClick(**this**.**name**)}>按钮2</**button**>  </**div**>  );  } }  **export default** ClassEvent; |

### 效果如下

|  |
| --- |
|  |

### 我们以后需要传参的时候多使用这种方式

## 4.事件对象event

### 我们再创建一个按钮，在他的onClick事件里面添加代码

|  |
| --- |
| **import *React***, {Component} **from 'react'**; **class** ClassEvent **extends** Component {  **name** = **'Jessica jade'** doClick(){  **return** () => {  ***window***.alert(**'hello from react'**)  };  }  *doClick2*=()=>{  ***console***.log(**'hello from react again'**)  }  handleClick(name){  ***console***.log(**'hello '**+name);  }  render() {  **return** (  <**div**>  <**button onClick=**{**this**.*doClick2*}>按钮</**button**>  <**button onClick=**{()=> **this**.handleClick(**this**.**name**)}>按钮2</**button**>  <**button onClick=**{event => ***console***.log(event)}>按钮3</**button**>  </**div**>  );  } }  **export default** ClassEvent; |

### 效果如下：点击按钮3会输出一个事件对象

|  |
| --- |
|  |

### 注意：有一种特殊现象，如果在按钮的点击事件里面调用一个有形参的函数而调用时没有传递参数，系统会自动传递一个event对象给这个形参，如，我们新建一个按钮，添加事件代码如下

|  |
| --- |
| **import *React***, {Component} **from 'react'**; *// function doClick() { // return () => { // window.alert('hello from react') // }; // }* **class** ClassEvent **extends** Component {  **name** = **'Jessica jade'** doClick(){  **return** () => {  ***window***.alert(**'hello from react'**)  };  }  *doClick2*=(name)=>{  ***console***.log(name + **'hello from react again'**)  }   handleClick(name){ *//如果在按钮的点击事件里面调用时没有传递参数，系统会自动传递一个event对象给这个形参* ***console***.log(name);  }  render() {  **return** (  <**div**>  <**button onClick=**{**this**.*doClick2*}>按钮</**button**>  <**button onClick=**{()=> **this**.handleClick(**this**.**name**)}>按钮2</**button**>  <**button onClick=**{event => ***console***.log(event)}>按钮3</**button**>  <**button onClick=**{**this**.handleClick}>按钮4</**button**>  </**div**>  );  } }  **export default** ClassEvent; |

### 运行程序，我们点击按钮4，发现输出的是一个event对象

|  |
| --- |
|  |

### 事件对象有一个event..preventDefault()方法可以阻止某个事件的默认行为和一个event.preventPropagation()方法可以阻止事件流的传播，其中第一个方法用的比较多

## 总结一下，这里主要有4个知识点

### 1.事件绑定，通过onClick={事件方法}

### 2.事件方法就是把事件的处理写在外面，然后把方法名传递给onClick

### 3.事件传参，就是在事件方法外面包一个箭头函数，然后把需要传参的事件方法写在里面

### 4.事件对象，事件方法默认就可以获取一个事件对象

# 类组件的状态

## 类组件的状态，指的是类组件内部的数据

## 还是上面的示例，我们新建一个文件夹，叫做03class-component-state的文件夹，在文件夹里面新建一个ClassState.jsx文件，在这个文件里面新建一个类组件，快捷键rcc+回车

|  |
| --- |
| **import *React***, {Component} **from 'react'**;  **class** ClassState **extends** Component {  render() {  **return** (  <**div**>    </**div**>  );  } }  **export default** ClassState; |

## 然后在App.jsx里面引入这个组件，为了清晰起见，我们把前面的组件注释了

|  |
| --- |
| **import** { *useState* } **from 'react'** *// import reactLogo from './assets/react.svg' // import viteLogo from '/vite.svg' // import './App.css'* **import** *FuncComponent* **from "./01-compoment-basic/FuncComponent"**; **import** ClassComponent **from "./01-compoment-basic/ClassComponent"**; **import** ClassEvent **from "./02-class-component-events/ClassEvent"**; **import** ClassState **from "./03class-component-state/ClassState"**;  **function** *App*() {   **return** (  <>  <**h1**>React demo3</**h1**>  {*/\*<FuncComponent/>\*/*}  {*/\*<ClassComponent/>\*/*}  {*/\*<ClassEvent/>\*/*}  <**ClassState**></**ClassState**>  </>  ) }  **export default** *App* |

## 1.设置状态，数据初始化

### 怎么设置类组件的状态？它有一个固定的写法，在render函数的外面给一个叫做state的变量赋值

|  |
| --- |
| **import *React***, {Component} **from 'react'**;  **class** ClassState **extends** Component {  **state** = {  **name**:**'Jessie'**,  **age**:20  }  render() {  **return** (  <**div**>  <**p**>ClassState</**p**>  </**div**>  );  } }  **export default** ClassState; |

## 2.渲染数据

### 由于state是属于类的，而且它又是一个对象，所以需要使用this.state.属性名的方式来使用

|  |
| --- |
| **import *React***, {Component} **from 'react'**;  **class** ClassState **extends** Component {  **state** = {  **name**:**'Jessie'**,  **age**:20  }  render() {  **return** (  <**div**>  <**p**>姓名:{**this**.**state**.**name**}</**p**>  <**p**>年龄:{**this**.**state**.**age**}</**p**>  </**div**>  );  } }  **export default** ClassState; |

### 效果

|  |
| --- |
|  |

## 3.修改状态（数据）

### 我们新建两个按钮一个用于增加年龄一个用于减小年龄，添加如下代码，我们想当然认为这样做就可以改变年龄

|  |
| --- |
| **import *React***, {Component} **from 'react'**;  **class** ClassState **extends** Component {  **state** = {  **name**:**'Jessie'**,  **age**:20  }  increment(){  **this**.**state**.**age**++  }  decrement(){  **this**.**state**.**age**--  }  render() {  **return** (  <**div**>  <**h3**>姓名:{**this**.**state**.**name**}</**h3**>  <**h3**>年龄:{**this**.**state**.**age**}</**h3**>  {*/\*<button onClick={()=>{this.state.age++}}>增加年龄</button>\*/*}  {*/\*<button onClick={()=>{this.state.age--}}>减小年龄</button>\*/*}  <**button onClick=**{()=>{**this**.increment()}}>增加年龄</**button**>  <**button onClick=**{()=>{**this**.decrement()}}>减小年龄</**button**>  </**div**>  );  } }  **export default** ClassState; |

### 然而，并没有效果，我们在increment方法里面添加一个alert语句发现其实年龄是改变了

|  |
| --- |
| **import *React***, {Component} **from 'react'**;  **class** ClassState **extends** Component {  **state** = {  **name**:**'Jessie'**,  **age**:20  }  increment(){  **this**.**state**.**age**++  *alert*(**this**.**state**.**age**)  }  decrement(){  **this**.**state**.**age**--  }  render() {  **return** (  <**div**>  <**h3**>姓名:{**this**.**state**.**name**}</**h3**>  <**h3**>年龄:{**this**.**state**.**age**}</**h3**>  {*/\*<button onClick={()=>{this.state.age++}}>增加年龄</button>\*/*}  {*/\*<button onClick={()=>{this.state.age--}}>减小年龄</button>\*/*}  <**button onClick=**{()=>{**this**.increment()}}>增加年龄</**button**>  <**button onClick=**{()=>{**this**.decrement()}}>减小年龄</**button**>  </**div**>  );  } }  **export default** ClassState; |

### 效果，我们点击一下增加年龄按钮，会弹出新年龄，但是没有把新年龄渲染出来

|  |
| --- |
|  |

### 这是为什么？因为在react中修改state的数据需要一个专用方法setState，我们先用setState来修改姓名，我们新增一个修改姓名的按钮并且添加事件处理

|  |
| --- |
| **import *React***, {Component} **from 'react'**;  **class** ClassState **extends** Component {  **state** = {  **name**:**'Jessie'**,  **age**:20  }  *increment*= () =>{  **this**.setState({  **age**:**this**.**state**.**age**+1  })  }  *decrement*=()=>{  **this**.setState({  **age**:**this**.**state**.**age**-1  })  }  render() {  **return** (  <**div**>  <**h3**>姓名:{**this**.**state**.**name**}</**h3**>  <**h3**>年龄:{**this**.**state**.**age**}</**h3**>  {*/\*<button onClick={()=>{this.state.age++}}>增加年龄</button>\*/*}  {*/\*<button onClick={()=>{this.state.age--}}>减小年龄</button>\*/*}  <**button onClick=**{()=>{  **this**.setState({  **name**:**'Jennifer'** })  }}>修改姓名</**button**>  <**button onClick=**{**this**.*increment*}>增加年龄</**button**>  <**button onClick=**{**this**.*decrement*}>减小年龄</**button**>  </**div**>  );  } }  **export default** ClassState; |

### 效果，此时修改数据就能反映到网页上面

|  |
| --- |
|  |

### 上面的代码可以优化一下，使用解构表达式把state的属性单独拿出来

|  |
| --- |
| **import *React***, {Component} **from 'react'**;  **class** ClassState **extends** Component {  **state** = {  **name**:**'Jessie'**,  **age**:20  }  *increment*= () =>{  **this**.setState({  **age**:**this**.**state**.**age**+1  })  }  *decrement*=()=>{  **this**.setState({  **age**:**this**.**state**.**age**-1  })  }  render() {  **let** {name,age} = **this**.**state  return** (  <**div**>  <**h3**>姓名:{name}</**h3**>  <**h3**>年龄:{age}</**h3**>  <**button onClick=**{()=>{  **this**.setState({  **name**:**'Jennifer'** })  }}>修改姓名</**button**>  <**button onClick=**{**this**.*increment*}>增加年龄</**button**>  <**button onClick=**{**this**.*decrement*}>减小年龄</**button**>  </**div**>  );  } }  **export default** ClassState; |

### 运行一下，效果是一样的

|  |
| --- |
|  |

#### 注意：解构表达式里面的变量是有作用域的。

## 4.关于setState方法的一些特性

### 我们在increment方法里面修改了state之后再添加一句代码输出state的age属性

|  |
| --- |
| **import *React***, {Component} **from 'react'**;  **class** ClassState **extends** Component {  **state** = {  **name**:**'Jessie'**,  **age**:20  }  *increment*= () =>{  **this**.setState({  **age**:**this**.**state**.**age**+1  })  ***console***.log(**this**.**state**.**age**)  }  *decrement*=()=>{  **this**.setState({  **age**:**this**.**state**.**age**-1  })  }  render() {  **let** {name,age} = **this**.**state  return** (  <**div**>  <**h3**>姓名:{name}</**h3**>  <**h3**>年龄:{age}</**h3**>  <**button onClick=**{()=>{  **this**.setState({  **name**:**'Jennifer'** })  }}>修改姓名</**button**>  <**button onClick=**{**this**.*increment*}>增加年龄</**button**>  <**button onClick=**{**this**.*decrement*}>减小年龄</**button**>  </**div**>  );  } }  **export default** ClassState; |

### 我们发现，修改数据后，界面的数据是新的，但是控制台输出的是上一次的数据

|  |
| --- |
|  |

### 为什么会这样呢？因为这个setState是一个异步方法。它其实还可以接受一个回调函数，我们可以在这个回调函数里面输出新值，我们把代码修改如下

|  |
| --- |
| **import *React***, {Component} **from 'react'**;  **class** ClassState **extends** Component {  **state** = {  **name**:**'Jessie'**,  **age**:20  }  *increment*= () =>{  **this**.setState({  **age**:**this**.**state**.**age**+1  },()=>{  ***console***.log(**this**.**state**.**age**)  })  }   *decrement*=()=>{  **this**.setState({  **age**:**this**.**state**.**age**-1  })  }  render() {  **let** {name,age} = **this**.**state  return** (  <**div**>  <**h3**>姓名:{name}</**h3**>  <**h3**>年龄:{age}</**h3**>  <**button onClick=**{()=>{  **this**.setState({  **name**:**'Jennifer'** })  }}>修改姓名</**button**>  <**button onClick=**{**this**.*increment*}>增加年龄</**button**>  <**button onClick=**{**this**.*decrement*}>减小年龄</**button**>  </**div**>  );  } }  **export default** ClassState; |

### 效果如下

|  |
| --- |
|  |

### 还有一点需要注意，如果在一次的触发事件里面调用多次setState(),数据只会改变一次也就是触发一次，改变一次，我们把increment的代码里面的setState调用复制一份，

|  |
| --- |
| **import *React***, {Component} **from 'react'**;  **class** ClassState **extends** Component {  **state** = {  **name**:**'Jessie'**,  **age**:20  }  *increment*= () =>{  **this**.setState({  **age**:**this**.**state**.**age**+1  },()=>{  *// console.log(this.state.age)* })  **this**.setState({  **age**:**this**.**state**.**age**+1  },()=>{  *// console.log(this.state.age)* })   }   *decrement*=()=>{  **this**.setState({  **age**:**this**.**state**.**age**-1  })  }  render() {  **let** {name,age} = **this**.**state  return** (  <**div**>  <**h3**>姓名:{name}</**h3**>  <**h3**>年龄:{age}</**h3**>  <**button onClick=**{()=>{  **this**.setState({  **name**:**'Jennifer'** })  }}>修改姓名</**button**>  <**button onClick=**{**this**.*increment*}>增加年龄</**button**>  <**button onClick=**{**this**.*decrement*}>减小年龄</**button**>  </**div**>  );  } }  **export default** ClassState; |

### 效果，点击一下他就只增加一次，而不是我们想象的2次

|  |
| --- |
|  |

#### 原理是:如果在一次触发事件里面有多个并行的setState调用并且修改state的同一个属性，react会把他们合并，只有最后的一次起作用。

### 我们把代码修改一下，把第二个setState放入第一个setState的回调函数里面

|  |
| --- |
| **import *React***, {Component} **from 'react'**;  **class** ClassState **extends** Component {  **state** = {  **name**:**'Jessie'**,  **age**:20  }  *increment*= () =>{  **this**.setState({  **age**:**this**.**state**.**age**+1  },()=>{  **this**.setState({  **age**:**this**.**state**.**age**+1  })  })   }   *decrement*=()=>{  **this**.setState({  **age**:**this**.**state**.**age**-1  })  }  render() {  **let** {name,age} = **this**.**state  return** (  <**div**>  <**h3**>姓名:{name}</**h3**>  <**h3**>年龄:{age}</**h3**>  <**button onClick=**{()=>{  **this**.setState({  **name**:**'Jennifer'** })  }}>修改姓名</**button**>  <**button onClick=**{**this**.*increment*}>增加年龄</**button**>  <**button onClick=**{**this**.*decrement*}>减小年龄</**button**>  </**div**>  );  } }  **export default** ClassState; |

### 运行效果：现在才会点击一次修改2次

|  |
| --- |
|  |

### 我们把代码再修改一下，让一个事件处理函数调用两次setState但是设置state的不同属性

|  |
| --- |
| **import *React***, {Component} **from 'react'**;  **class** ClassState **extends** Component {  **state** = {  **name**:**'Jessie'**,  **age**:20  }  *increment*= () =>{  **this**.setState({  **age**:**this**.**state**.**age**+1  },()=>{  ***console***.log(**this**.**state**.**age**)  })  **this**.setState({  **name**:**'Jade'** })  }   *decrement*=()=>{  **this**.setState({  **age**:**this**.**state**.**age**-1  })  }  render() {  **let** {name,age} = **this**.**state  return** (  <**div**>  <**h3**>姓名:{name}</**h3**>  <**h3**>年龄:{age}</**h3**>  <**button onClick=**{()=>{  **this**.setState({  **name**:**'Jennifer'** })  }}>修改姓名</**button**>  <**button onClick=**{**this**.*increment*}>增加年龄</**button**>  <**button onClick=**{**this**.*decrement*}>减小年龄</**button**>  </**div**>  );  } }  **export default** ClassState; |

### 效果：我们发现这样子的两次setState都会执行，并不会合并

|  |
| --- |
|  |

# 类组件的计算属性

## 我们在src下面新建一个文件夹04class-computed-property,然后在这个文件夹里面创建一个ClassComputed.jsx文件

|  |
| --- |
| **import *React***, {Component} **from 'react'**;  **class** ClassComputed **extends** Component {  render() {  **return** (  <**div**>    </**div**>  );  } }  **export default** ClassComputed; |

## 然后在App.jsx在引用这个组件

|  |
| --- |
| **import** { *useState* } **from 'react'****import** *FuncComponent* **from "./01-compoment-basic/FuncComponent"**; **import** ClassComponent **from "./01-compoment-basic/ClassComponent"**; **import** ClassEvent **from "./02-class-component-events/ClassEvent"**; **import** ClassState **from "./03class-component-state/ClassState"**; **import** ClassComputed **from "./04class-computed-property/ClassComputed" function** *App*() {   **return** (  <>  <**h1**>React demo3</**h1**>  {*/\*<FuncComponent/>\*/*}  {*/\*<ClassComponent/>\*/*}  {*/\*<ClassEvent/>\*/*}  {*/\* <ClassState></ClassState>\*/*}  <**ClassComputed**></**ClassComputed**>   </>  ) }  **export default** *App* |

## 其实在react中根本没有计算属性的概念，但是我们可以结合class的语法，使用get来模拟计算属性的效果，我们在ClassComputed.jsx文件中编写如下代码

|  |
| --- |
| **import *React***, {Component} **from 'react'**;  **class** ClassComputed **extends** Component {  **state**={  **num1**:10,  **num2**:20  }  **get** sum(){  **return this**.**state**.**num1** + **this**.**state**.**num2** }   render() {  **return** (  <**div**>  <**h3**>总数是：{**this**.sum}</**h3**>  </**div**>  );  } }  **export default** ClassComputed; |

### 这个sum本来是一个方法，他是可以作为事件方法来用，但是在他前面加了get关键字，它就变成一个属性了此时就不能作为函数来调用了，效果如下

|  |
| --- |
|  |

### 虽然能够模拟出来但是他和vue里面的计算属性是不同的，它没有缓存，而vue的计算属性有缓存，唯一的好处是可以用来把计算属性和普通的方法做一个区分

# 类组件的生命周期

## 下图是react组的生命周期图

|  |
| --- |
|  |

### 组件的生命周期可分成三个状态：

* Mounting(挂载)：已插入真实 DOM
* Updating(更新)：正在被重新渲染
* Unmounting(卸载)：已移出真实 DOM

## 挂载

当组件实例被创建并插入 DOM 中时，其生命周期调用顺序如下：

* [constructor()](https://www.runoob.com/react/react-ref-constructor.html): 在 React 组件挂载之前，会调用它的构造函数。
* [getDerivedStateFromProps()](https://www.runoob.com/react/react-ref-getderivedstatefromprops.html): 在调用 render 方法之前调用，并且在初始挂载及后续更新时都会被调用。
* [render()](https://www.runoob.com/react/react-ref-render.html): render() 方法是 class 组件中唯一必须实现的方法。
* [componentDidMount()](https://www.runoob.com/react/react-ref-componentdidmount.html): 在组件挂载后（插入 DOM 树中）立即调用。

render() 方法是 class 组件中唯一必须实现的方法，其他方法可以根据自己的需要来实现。

这些方法的详细说明，可以参考[官方文档](https://zh-hans.reactjs.org/docs/react-component.html#reference)。

## 更新

每当组件的 state 或 props 发生变化时，组件就会更新。

当组件的 props 或 state 发生变化时会触发更新。组件更新的生命周期调用顺序如下：

* [getDerivedStateFromProps()](https://www.runoob.com/react/react-ref-getderivedstatefromprops.html): 在调用 render 方法之前调用，并且在初始挂载及后续更新时都会被调用。根据 shouldComponentUpdate() 的返回值，判断 React 组件的输出是否受当前 state 或 props 更改的影响。
* [shouldComponentUpdate()](https://www.runoob.com/react/react-ref-shouldcomponentupdate.html):当 props 或 state 发生变化时，shouldComponentUpdate() 会在渲染执行之前被调用。
* [render()](https://www.runoob.com/react/react-ref-render.html): render() 方法是 class 组件中唯一必须实现的方法。
* [getSnapshotBeforeUpdate()](https://www.runoob.com/react/react-ref-getsnapshotbeforeupdate.html): 在最近一次渲染输出（提交到 DOM 节点）之前调用。
* [componentDidUpdate()](https://www.runoob.com/react/react-ref-componentdidupdate.html): 在更新后会被立即调用。

render() 方法是 class 组件中唯一必须实现的方法，其他方法可以根据自己的需要来实现。

这些方法的详细说明，可以参考[官方文档](https://zh-hans.reactjs.org/docs/react-component.html#reference)。

## 卸载

当组件从 DOM 中移除时会调用如下方法：

* [componentWillUnmount()](https://www.runoob.com/react/react-ref-componentwillunmount.html): 在组件卸载及销毁之前直接调用。

## 从图中我们可以看到整个生命周期一共有8给函数，其中getDerivedStatFromProps和render这两个函数非常重要。每一个类组件都必须实现render函数

## 我们通过实例来学习一下生命周期函数，还是上面的项目，在src下面新建一个文件夹05class-comp-life-cycle，然后新建一个ClassLifeCycle.jsx文件，使用rcc+回车快捷键创建一个空组件

|  |
| --- |
| **import *React***, {Component} **from 'react'**;  **class** ClassLifeCycle **extends** Component {  render() {  **return** (  <**div**>    </**div**>  );  } }  **export default** ClassLifeCycle; |

## 然后在App.jsx中引用这个组件

|  |
| --- |
| **import** { *useState* } **from 'react'** *// import reactLogo from './assets/react.svg' // import viteLogo from '/vite.svg' // import './App.css'* **import** *FuncComponent* **from "./01-compoment-basic/FuncComponent"**; **import** ClassComponent **from "./01-compoment-basic/ClassComponent"**; **import** ClassEvent **from "./02-class-component-events/ClassEvent"**; **import** ClassState **from "./03class-component-state/ClassState"**; **import** ClassComputed **from "./04class-computed-property/ClassComputed" import** ClassLifeCycle **from "./05class-comp-life-cycle/ClassLifeCycle"**; **function** *App*() {   **return** (  <>  <**h1**>React demo3</**h1**>  {*/\*<FuncComponent/>\*/*}  {*/\*<ClassComponent/>\*/*}  {*/\*<ClassEvent/>\*/*}  {*/\* <ClassState></ClassState>\*/*}  {*/\* <ClassComputed></ClassComputed>\*/*}  <**ClassLifeCycle**></**ClassLifeCycle**>  </>  ) }  **export default** *App* |

## 我们在ClassLifeCycle.jsx中添加代码来测试一下constructor函数，为了便于对比我们在render函数里面也添加一条输出语句

|  |
| --- |
| **import *React***, {Component} **from 'react'**;  **class** ClassLifeCycle **extends** Component {  constructor() {  **super**();  ***console***.log(**'constructor()'**)  }  render() {  ***console***.log(**'render()'**)  **return** (  <**div**>  <**h4**>ClassLifeCycle</**h4**>  </**div**>  );  } }  **export default** ClassLifeCycle; |

### 效果

|  |
| --- |
|  |

## 其实严格意义上来讲，state应该在constructor里面定义的，但是react底层帮我们简化了，所以我可以直接是render函数外面定义state。

## 我们来学习一下componentDidMount()函数，他是当页面加载完成后用来发送网络请求的

|  |
| --- |
| **import *React***, {Component} **from 'react'**;  **class** ClassLifeCycle **extends** Component {  *// constructor() { //这是一个常用的生命周期函数，但是react帮我们简化了，我们不需要经常写  // super();  // console.log('constructor()')  // }* componentDidMount() {  ***console***.log(**'组件挂载完成，可以发送网络请求'**)  }   render() {  ***console***.log(**'render()'**)  **return** (  <**div**>  <**h4**>ClassLifeCycle</**h4**>  </**div**>  );  } }  **export default** ClassLifeCycle; |

### 效果

|  |
| --- |
|  |

## 可见这个函数虽然写在render函数之前，但是他是在render函数调用之后执行的

## 我们来修改一下代码看看更新的效果，我们在render函数里面添加一条输出语句

|  |
| --- |
| **import *React***, {Component} **from 'react'**;  **class** ClassLifeCycle **extends** Component {  *// constructor() { //这是一个常用的生命周期函数，但是react帮我们简化了，我们不需要经常写  // super();  // console.log('constructor()')  // }* **state** = {  **num**:1  }  componentDidMount() {  ***console***.log(**'组件挂载完成，可以发送网络请求'**)  }   componentDidUpdate(prevProps, prevState, snapshot) {  ***console***.log(**'componentDidUpdate called '**)  ***console***.log(prevProps)  ***console***.log(prevState)  }   render() {  ***console***.log(**'render()'**)  **return** (  <**div**>  <**h4**>ClassLifeCycle</**h4**>  <**h5**>{**this**.**state**.**num**}</**h5**>  <**button onClick=**{()=>{  **this**.setState({  **num**:**this**.**state**.**num**+1  })  }}>改变数字</**button**>  </**div**>  );  } }  **export default** ClassLifeCycle; |

### 效果，

|  |
| --- |
|  |

### 我们发现当数据更新时，会先调用render然后调用componentDidUpdate函数，其实componentDidUpdate函数用的很少。

## 在组件卸载前会调用componentWillUnmount函数，在这里主要做一下清理工作，如停止计算器setInterval的运行（这个方法即使你把组件销毁了，它还会继续运行），

|  |
| --- |
| **import *React***, {Component} **from 'react'**;  **class** ClassLifeCycle **extends** Component {  *// constructor() { //这是一个常用的生命周期函数，但是react帮我们简化了，我们不需要经常写  // super();  // console.log('constructor()')  // }* **state** = {  **num**:1  }  componentDidMount() {  ***console***.log(**'组件挂载完成，可以发送网络请求'**)  }   componentDidUpdate(prevProps, prevState, snapshot) {  ***console***.log(**'componentDidUpdate called '**)  ***console***.log(prevProps)  ***console***.log(prevState)  }  componentWillUnmount() {  ***console***.log(**'即将卸载组件'**)  }   render() {  ***console***.log(**'render()'**)//在这里这句话的输出看不到  **return** (  <**div**>  <**h4**>ClassLifeCycle</**h4**>  <**h5**>{**this**.**state**.**num**}</**h5**>  <**button onClick=**{()=>{  **this**.setState({  **num**:**this**.**state**.**num**+1  })  }}>改变数字</**button**>  </**div**>  );  } }  **export default** ClassLifeCycle; |

## 在本例中无法看到这个函数的效果，以后可能会用到

## 此外还有一个不常用但是比较重要的，就是

## [shouldComponentUpdate](https://www.runoob.com/react/react-ref-shouldcomponentupdate.html)方法，我们添加运行代码看看效果

|  |
| --- |
| **import *React***, {Component} **from 'react'**;  **class** ClassLifeCycle **extends** Component {  *// constructor() { //这是一个常用的生命周期函数，但是react帮我们简化了，我们不需要经常写  // super();  // console.log('constructor()')  // }* **state** = {  **num**:1  }  componentDidMount() {  ***console***.log(**'组件挂载完成，可以发送网络请求'**)  }   componentDidUpdate(prevProps, prevState, snapshot) {  ***console***.log(**'componentDidUpdate called '**)  ***console***.log(prevProps)  ***console***.log(prevState)  }  componentWillUnmount() {  ***console***.log(**'即将卸载组件'**)  }  shouldComponentUpdate(nextProps, nextState, nextContext) {  ***console***.log(**'inside shouldComponentUpdate'**)  ***console***.log(nextProps)  ***console***.log(nextState)  ***console***.log(nextContext)  **return true** }   render() {  ***console***.log(**'render()'**)  **return** (  <**div**>  <**h4**>ClassLifeCycle</**h4**>  <**h5**>{**this**.**state**.**num**}</**h5**>  <**button onClick=**{()=>{  **this**.setState({  **num**:**this**.**state**.**num**+1  })  }}>改变数字</**button**>  </**div**>  );  } }  **export default** ClassLifeCycle; |

### 效果

|  |
| --- |
|  |

可以看到，更新数据时，会先执行shouldComponentUpdate方法，再执行render方法最后执行componentDidUpdate，这个函数可以

设置组件是否更新，我们把它改为return false，界面就不会更新

|  |
| --- |
| **import *React***, {Component} **from 'react'**;  **class** ClassLifeCycle **extends** Component {  *// constructor() { //这是一个常用的生命周期函数，但是react帮我们简化了，我们不需要经常写  // super();  // console.log('constructor()')  // }* **state** = {  **num**:1  }  componentDidMount() {  ***console***.log(**'组件挂载完成，可以发送网络请求'**)  }   componentDidUpdate(prevProps, prevState, snapshot) {  ***console***.log(**'componentDidUpdate called '**)  ***console***.log(prevProps)  ***console***.log(prevState)  }  componentWillUnmount() {  ***console***.log(**'即将卸载组件'**)  }  shouldComponentUpdate(nextProps, nextState, nextContext) {  ***console***.log(**'inside shouldComponentUpdate'**)  ***console***.log(nextProps)  ***console***.log(nextState)  ***console***.log(nextContext)  *// return true* **return false** }   render() {  ***console***.log(**'render()'**)  **return** (  <**div**>  <**h4**>ClassLifeCycle</**h4**>  <**h5**>{**this**.**state**.**num**}</**h5**>  <**button onClick=**{()=>{  **this**.setState({  **num**:**this**.**state**.**num**+1  })  }}>改变数字</**button**>  </**div**>  );  } }  **export default** ClassLifeCycle; |

## 效果

|  |
| --- |
|  |

## 可以看到内容修改了，但是界面没有更新，说明它默认是返回true的，我们把它改回true

|  |
| --- |
| **import *React***, {Component} **from 'react'**;  **class** ClassLifeCycle **extends** Component {  *// constructor() { //这是一个常用的生命周期函数，但是react帮我们简化了，我们不需要经常写  // super();  // console.log('constructor()')  // }* **state** = {  **num**:1  }  componentDidMount() {  ***console***.log(**'组件挂载完成，可以发送网络请求'**)  }   componentDidUpdate(prevProps, prevState, snapshot) {  ***console***.log(**'componentDidUpdate called '**)  ***console***.log(prevProps)  ***console***.log(prevState)  }  componentWillUnmount() {  ***console***.log(**'即将卸载组件'**)  }  shouldComponentUpdate(nextProps, nextState, nextContext) {  ***console***.log(**'inside shouldComponentUpdate'**)  ***console***.log(nextProps)  ***console***.log(nextState)  ***console***.log(nextContext)  **return true** }   render() {  ***console***.log(**'render()'**)  **return** (  <**div**>  <**h4**>ClassLifeCycle</**h4**>  <**h5**>{**this**.**state**.**num**}</**h5**>  <**button onClick=**{()=>{  **this**.setState({  **num**:**this**.**state**.**num**+1  })  }}>改变数字</**button**>  </**div**>  );  } }  **export default** ClassLifeCycle; |

### 可以看到，界面又正常更新了

|  |
| --- |
|  |

## 那么，react为什么要阻止组件的更新呢？这是用于如果界面没有东西显示，那么其实就不用渲染，这个函数就是在这种情况下可以设置阻止组件更新，避免无效的render函数调用。

## 注意：在react中，类组件是否更新不是取决于数据是否改变，而是看是否调用了setState函数。一旦调用了setState函数，组件就会更新然后调用render函数重新渲染组件。这就产生很多没有必要的render渲染。从而影响程序的性能。shouldComponentUpdate就可以用来解决这个问题，我们把代码修改一下

|  |
| --- |
| **import *React***, {Component} **from 'react'**;  **class** ClassLifeCycle **extends** Component {  *// constructor() { //这是一个常用的生命周期函数，但是react帮我们简化了，我们不需要经常写  // super();  // console.log('constructor()')  // }* **state** = {  **num**:1  }  componentDidMount() {  ***console***.log(**'组件挂载完成，可以发送网络请求'**)  }   componentDidUpdate(prevProps, prevState, snapshot) {  ***console***.log(**'componentDidUpdate called '**)  ***console***.log(prevProps)  ***console***.log(prevState)  }  componentWillUnmount() {  ***console***.log(**'即将卸载组件'**)  }  shouldComponentUpdate(nextProps, nextState, nextContext) {  ***console***.log(**'inside shouldComponentUpdate'**)  *// console.log(nextProps)  // console.log(nextState)  // console.log(nextContext)* **if**(nextState.**num** === **this**.**state**.**num**){  **return false** }   **return true** }   render() {  ***console***.log(**'render()'**)  **return** (  <**div**>  <**h4**>ClassLifeCycle</**h4**>  <**h5**>{**this**.**state**.**num**}</**h5**>  <**button onClick=**{()=>{  **this**.setState({  **num**:**this**.**state**.**num**+1  *// num:1 //*  })  }}>改变数字</**button**>  </**div**>  );  } }  **export default** ClassLifeCycle; |

### 效果

|  |
| --- |
|  |

## 假如我们把state的num写死为1，他是不会更新组件的

|  |
| --- |
| **import *React***, {Component} **from 'react'**;  **class** ClassLifeCycle **extends** Component {  *// constructor() { //这是一个常用的生命周期函数，但是react帮我们简化了，我们不需要经常写  // super();  // console.log('constructor()')  // }* **state** = {  **num**:1  }  componentDidMount() {  ***console***.log(**'组件挂载完成，可以发送网络请求'**)  }   componentDidUpdate(prevProps, prevState, snapshot) {  ***console***.log(**'componentDidUpdate called '**)  ***console***.log(prevProps)  ***console***.log(prevState)  }  componentWillUnmount() {  ***console***.log(**'即将卸载组件'**)  }  shouldComponentUpdate(nextProps, nextState, nextContext) {  ***console***.log(**'inside shouldComponentUpdate'**)  *// console.log(nextProps)  // console.log(nextState)  // console.log(nextContext)* **if**(nextState.**num** === **this**.**state**.**num**){  **return false** }   **return true** }   render() {  ***console***.log(**'render()'**)  **return** (  <**div**>  <**h4**>ClassLifeCycle</**h4**>  <**h5**>{**this**.**state**.**num**}</**h5**>  <**button onClick=**{()=>{  **this**.setState({  **num**:1  })  }}>改变数字</**button**>  </**div**>  );  } }  **export default** ClassLifeCycle; |

### 效果

|  |
| --- |
|  |

### 可以看到，点击按钮没有调用render函数

## 但是用这种方法来优化性能还不是最好的，我们有一个更好的办法，就是使用react的

## PureComponent，我们可以修改一下代码，使得它继承自PureComponent，我们就不需要自己来写shouldComponentUpdate的代码了，因为PureComponent内部已经帮我们做好了

|  |
| --- |
| **import *React***, {Component,PureComponent} **from 'react'**;  **class** ClassLifeCycle **extends** PureComponent {  *// constructor() { //这是一个常用的生命周期函数，但是react帮我们简化了，我们不需要经常写  // super();  // console.log('constructor()')  // }* **state** = {  **num**:1  }  componentDidMount() {  ***console***.log(**'组件挂载完成，可以发送网络请求'**)  }   componentDidUpdate(prevProps, prevState, snapshot) {  ***console***.log(**'componentDidUpdate called '**)  ***console***.log(prevProps)  ***console***.log(prevState)  }  componentWillUnmount() {  ***console***.log(**'即将卸载组件'**)  }  *// shouldComponentUpdate(nextProps, nextState, nextContext) { //继承自PureComponent的组件不需要写这个函数  // console.log('inside shouldComponentUpdate')  // // console.log(nextProps)  // // console.log(nextState)  // // console.log(nextContext)  // if(nextState.num === this.state.num){  // return false  // }  //  // return true  // }* render() {  ***console***.log(**'render()'**)  **return** (  <**div**>  <**h4**>ClassLifeCycle</**h4**>  <**h5**>{**this**.**state**.**num**}</**h5**>  <**button onClick=**{()=>{  **this**.setState({  **num**:**this**.**state**.**num**+1  *// num:1* })  }}>改变数字</**button**>  </**div**>  );  } }  **export default** ClassLifeCycle; |

## 可以测试一下，效果是一样的

## 其实在类组件中只有render函数是必须写的，其他可以根据需要来写。render函数渲染的是虚拟dom而不是真实dom。