

React

React概述

1.1 什么是React

React是一个用于构建用户界面的JavaScript库。

用户界面：HTML页面（前端）

React主要用来写HTML页面，或构建Web应用

如果从MVC的角度来看，React仅仅是视图层（V），也就是只负责视图的渲染，而非提供了完整的M和C的功能。

React起源于Facebook的内部项目，后来又用来架设Instagram的网站，并于2013年5月开源

1.2 React的特点

- 声明式

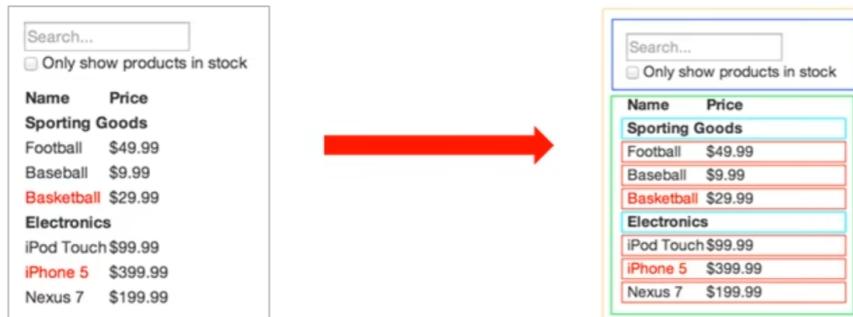
只需要描述UI（HTML）看起来是什么样，就跟写HTML一样

React负责渲染UI，并在数据变化时更新UI

```
const jsx = <div className="app">
  <h1>Hello React! 动态变化数据: {count}</h1>
</div>
```

- 基于组件

- 组件是React最重要的内容
- 组件表示页面中的部分内容
- 组合、复用多个组件，可以实现完整的页面功能



- 学习一次，随处使用

- 使用React可以开发Web应用
- 使用React可以开发移动端原生应用（react-native）
- 使用React可以开发VR（虚拟现实）应用（react360）

React的基本使用

2.1 React的安装

安装命令：npm i react react-dom

- react包是核心，提供.createElement等方法
- react-dom包提供ReactDOM.render等方法

2.2 React的使用

1. 引入react和react-dom两个js文件

```
<script src="./node_modules/react/umd/react.development.js"></script>
<script src="./node_modules/react-dom/umd/react-dom.development.js"></script>
```

2. 创建React元素

3. 渲染React元素到页面中

```
<div id="root"></div>
<script>
  const title = React.createElement('h1', null, 'Hello React')#参数1: 元素名称, 参数2: 元素属性, 参数3: 元素的子节点
  ReactDOM.render(title, document.getElementById('root'))#参数1: 要渲染的react元素, 参数2: 挂载点
</script>
```

React脚手架的使用

3.1 React脚手架意义

1. 脚手架是开发现代Web应用的必备
2. 充分利用Webpack, Babel, ESLint等工具辅助项目开发
3. 零配置，无需动手配置繁琐的工具即可使用
4. 关注业务，而不是工具配置

3.2 使用React脚手架初始化项目

```
# 初始化项目命令 npx 脚手架名称 项目名称  
npx create-react-app my-app  
  
# 启动项目，在项目跟目录执行命令  
npm start
```

npx命令介绍

- npm v5.2.0引入的一条命令
- 目的：提升包内提供的命令行工具的使用体验
- 原来：先安装脚手架包，再使用这个包中提供的命令
- 现在：**无需安装脚手架包**，就可以直接使用这个包提供的命令

补充说明

推荐使用：npx create-react-app my-app

其他：npm init react-app my-app

yarn create react-app my-app

yarn是Facebook发布的包管理器，可以看作是npm的替代品，功能与npm相同

yarn具有快速，可靠和安全的特点

初始化新项目：yarn init

安装包：yarn add 包名称

安装项目依赖项：yarn

其他命令，参考yarn文档

3.3 在脚手架中使用React

1. 导入react和react-dom两个包

```
import React from 'react'  
import ReactDOM from 'react-dom'#用于Web应用，手机/VR的应用需导其他包
```

2. 调用React.createElement()方法创建react元素。
3. 调用ReactDOM.render()方法渲染react元素到页面中。

JSX

JSX的基本使用

1.1 createElement()的问题

- 繁琐不简洁
- 不直观，无法一眼看出所描述的结构

- 不优雅，用户体验不爽



1.2 JSX简介

JSX是JavaScript XML的简写，表示JavaScript代码中写XML（HTML）格式的代码。

优势：声明式语法更加直观、与HTML结构相同，降低了学习成本，提升开发效率

JSX是React的核心内容

1.3 使用步骤

使用JSX语法创建react元素

```
#使用JSX语法，创建react元素
const title = <h1>Hello JSX</h1>
```

使用ReactDOM.render()方法渲染react元素到页面中

```
#渲染创建好的React元素
ReactDOM.render(title, document.getElementById('root'))
```

为什么脚手架中可以使用JSX语法

JSX不是标准的ECMAScript语法，它是ECMAScript的语法扩展。

需要使用babel编译处理后，才能在浏览器环境中使用。

create-react-app脚手架中已经默认有该配置，无需手动配置

编译JSX语法的包为：@babel/preset-react

1.4 注意点

React元素的属性名使用驼峰命名法

特殊属性名：class -> **className**、for -> htmlFor、tabindex -> tabIndex

没有子节点的React元素可以用 /> 结束

推荐：使用小括号包裹JSX，从而避免JS中的自动插入分号陷阱。

```
// 使用小括号包裹 JSX
const dv = (
  <div>Hello JSX</div>
)
```

JSX中使用JavaScript表达式

嵌入Js表达式

- 数据存储在JS中
- 语法：**{JavaScript表达式}**
- 注意：语法中是**单大括号**，不是双大括号！

```
const name = 'Jack'
const dv = (
  <div>你好，我叫：? ? ? </div>
)
```

```
const name = 'Jack'
const dv = (
  <div>你好，我叫：{name}</div>
)
```

注意点

- **单大括号**中可以使用任意的JavaScript表达式
- JSX自身也是JS表达式
- 注意：JS中的对象是一个例外，一般只会出现在style属性中

- 注意：不能在{}中出现语句（比如：if/for等）

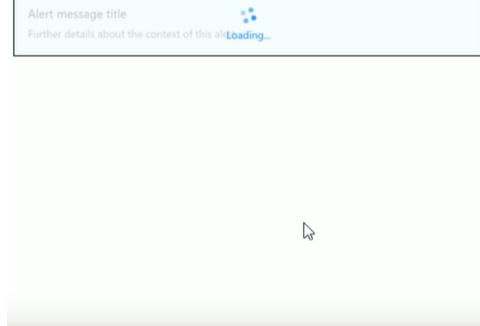
```
const h1 = <h1>我是JSX</h1>

const dv = (
  <div>嵌入表达式: {h1}</div>
)
```

```
//函数调用表达式
const sayHi = () => 'Hi~'
const title1 = (
  <h1>
    Hello JSX
    <p>{1}</p>
    <p>{'a'}</p>
    <p>{1 + 7}</p>
    <p>{3 > 5 ? '大于' : '小于等于'}</p>
    <p>{sayHi()}</p>
    {/*错误演示 */}
    {/*<p>{ a:'6' }</p> */}
    {/*{ if (true) {} } */}
    {/*{ for (var i = 0; i < 10; i++) {} } */}
  </h1>
)
ReactDOM.render(title1, document.getElementById('root-1'))
```

JSX的条件渲染

- 场景：loading效果
- 条件渲染：根据条件渲染特定的JSX结构
- 可以使用if/else或三元运算符或逻辑与运算符来实现



```
const loadData = () => {
  if (isLoading) {
    return <div>数据加载中, 请稍后...</div>
  }
  return (
    <div>数据加载完成, 此处显示加载后的数据</div>
  )
}
const dv = (
  <div>
    {loadData()}
  </div>
)
```

JSX的列表渲染

- 如果要渲染一组数据，应该使用数组的map()方法
- 注意：渲染列表时应该添加key属性，key属性的值要保证唯一
- 原则：map()遍历谁，就给谁添加key属性
- 注意：尽量避免使用索引号作为key

```
const songs = [
  {id: 1, name: '痴心绝对'},
  {id: 2, name: '像我这样的人'},
  {id: 3, name: '南山南'},
]

const list = (
  <ul>
    {songs.map(item => <li key={item.id}>{item.name}</li>)}
  </ul>
)
```

JSX的样式处理

1. 行内样式——style

```
<h1 style={{ color: 'red', backgroundColor: 'skyblue' }}>
  JSX的样式处理
</h1>
```

2. 类名——className (推荐)

```
<h1 className="title">  
  JSX的样式处理  
</h1>
```

总结

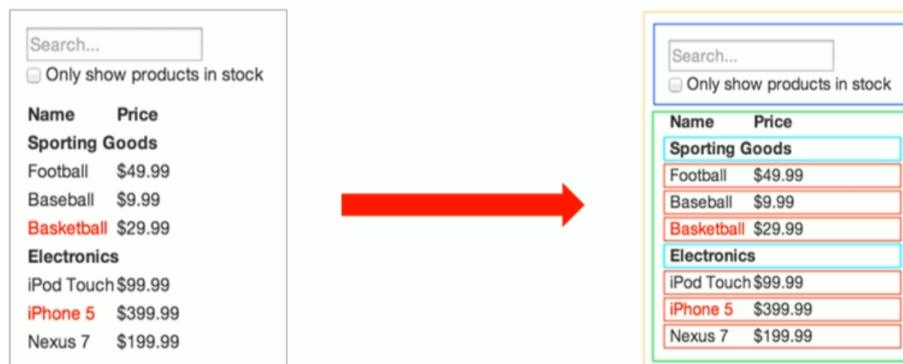
JSX

1. JSX是React的核心内容。
2. JSX表示在JS代码中写HTML结构，是React声明式的体现
3. 使用JSX配合嵌入式的JS表达式，条件渲染，列表渲染，可以描述任意UI结构
4. 推荐使用className的方式给JSX添加样式
5. React完全利用JS语言自身的能力来编写UI，而不是造轮子增强HTML功能

React组件基础

➤ React组件介绍

- 组件是React的一等公民，使用React就是在用组件
- 组件表示页面中的部分功能
- 组合多个组件实现完整的页面功能
- 特点：可复用、独立、可组合



➤ React组件的两种创建方式

1. 使用函数创建组件

- 使用JS中的函数创建的组件叫做：函数组件
- 函数组件必须有返回值
- 组件名称必须以大写字母开头，React据此区分 组件 和 普通的React元素
- 使用函数名作为组件标签名
- 如果返回值为null，表示不渲染任何内容

```
function Hello() {  
  return (  
    <div>这是我的第一个函数组件！</div>  
  )  
}  
  
ReactDOM.render(<Hello />, root)
```

2. 使用类创建组件

- 类组件：使用ES6的class创建的组件
- 约定1：类名称必须以大写字母开头
- 约定2：类组件应该继承React.Component父类，从而可以使用父类中提供的方法或属性
- 约定3：类组件必须提供render()方法
- 约定4：render()方法必须有返回值，表示该组件的结构

```
class Hello extends React.Component {  
  render() {  
    return <div>Hello Class Component！</div>  
  }  
}  
  
ReactDOM.render(<Hello />, root)
```

3. 抽离为独立JS文件

1. 创建Hello.js
2. 在Hello.js中导入React
3. 创建组件（函数或类）
4. 在Hello.js中导出该组件
5. 在index.js中导入Hello组件
6. 渲染组件

```
// Hello.js
import React from 'react'
class Hello extends React.Component {
  render() {
    return <div>Hello Class Component!</div>
  }
}
// 导出Hello组件
export default Hello
```

```
// index.js
import Hello from './Hello'
// 渲染导入的Hello组件
ReactDOM.render(<Hello />, root)
```

➤ React事件处理

1. 事件绑定

React事件绑定语法与DOM事件语法相似

语法: **on + 事件名称 = {事件处理程序}**, 比如: `onClick={() => {}}`

注意: **React事件采用驼峰命名法**, 比如: `onMouseEnter`、`onFocus`

```
class App extends React.Component {
  handleClick() {
    console.log('单击事件触发了')
  }
  render() {
    return (
      <button onClick={this.handleClick}></button>
    )
  }
}
```

通过函数组件绑定事件:

```
function App() {
  function handleClick() {
    console.log('单击事件触发了')
  }

  return (
    <button onClick={handleClick}>点我</button>
  )
}
```

2. 事件对象

可以通过事件处理程序的参数获取到事件对象

React中的事件对象叫做：合成事件（对象）

合成事件：兼容所有浏览器，无需担心跨浏览器兼容性问题

```
function handleClick(e) {
  e.preventDefault()
  console.log('事件对象', e)
}

<a onClick={handleClick}>点我，不会跳转页面</a>
```

➤有状态组件和无状态组件

- 函数组件又叫做无状态组件，类组件又叫做有状态组件
- 状态（state）即数据
- 函数组件没有自己的状态，只负责数据展示（静）
- 类组件有自己的状态，负责更新UI，让页面“动”起来

比如计数器案例中，点击按钮让数值加1。0和1就是不同时刻的状态，而由0变为1就表示状态发生了变化。状态变化后，UI也要相应的更新。React中想要实现该功能，就要使用有状态组件来完成。



➤组件中的state和setState()

1. state的基本使用

- 状态（state）即数据，是组件内部的私有数据，只能在组件内部使用

- state的值是对象，表示一个组件可以有多个数据

```
class Hello extends React.Component {
  constructor() {
    super()
    // 初始化state
    this.state = {
      count: 0
    }
  }
  render() {
    return (
      <div>有状态组件</div>
    )
  }
}
```

2. setState()修改状态

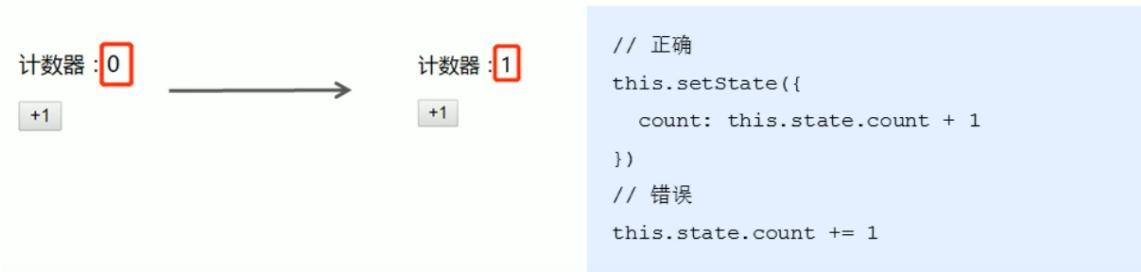
状态是可变的

语法: this.setState({要修改的数据})

注意: 不要直接修改state中的值，这是错误的！！！

setState()作用: 1.修改state 2.更新UI

思想: 数据驱动视图



从JSX中抽离事件处理程序

JSX中掺杂过多JS逻辑代码，会显得非常混乱

推荐: 将逻辑抽离到单独的方法中，保证JSX结构清晰

```
class App extends React.Component {
  //简化语法初始化state
  state = {
    count: 0
  }
  //事件处理程序
  onIncrement() {
    this.setState({
      count: this.state.count + 1
    })
  }
  render() {
    return (
      <div>
        <h1>计数器: {this.state.count}</h1>
        <button onClick={this.onIncrement}>+1</button>
      </div>
    )
  }
}
```

#由于this的指向问题，该写法会报错，this未定义

➤事件绑定this指向

1. 箭头函数

- 利用箭头函数自身不绑定this的特点
- render()方法中的this为组件实例，可以获取到setState()

```

class Hello extends React.Component {
  onIncrement() {
    this.setState({ ... })
  }
  render() {
    // 箭头函数中的this指向外部环境，此处为：render()方法
    return (
      <button onClick={() => this.onIncrement()}></button>
    )
  }
}

```

2. Function.prototype.bind()

利用ES5中的bind方法，将事件处理程序中的this与组件绑定到一起

```

class Hello extends React.Component {
  constructor() {
    super()
    this.onIncrement = this.onIncrement.bind(this)
  }
  // ...省略 onIncrement
  render() {
    return (
      <button onClick={this.onIncrement}></button>
    )
  }
}

```

3. class的实例方法

利用箭头函数形式的class实例方法

注意：该语法是实验性语法，但是，由于babel的存在可以直接使用

```

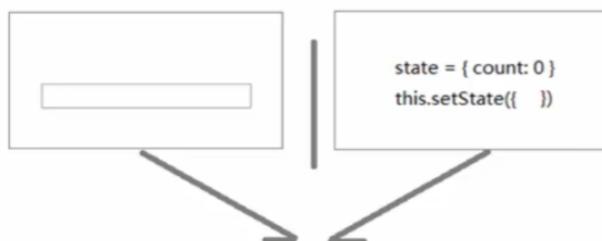
class Hello extends React.Component {
  onIncrement = () => {
    this.setState({ ... })
  }
  render() {
    return (
      <button onClick={this.onIncrement}></button>
    )
  }
}

```

➤表单处理

1. 受控组件

- HTML中的表单元素是可输入的，也就是有自己的可变状态
- 而，React中可变状态通常保存在state中，并且只能通过setState()方法来修改
- React将state与表单元素值value绑定到一起，由state的值来控制表单元素的值
- 受控组件：其值受到React控制的表单元素



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

步骤：

1. 在state中添加一个状态，作为表单元素的value值（控制表单元素值的来源）
2. 给表单元素绑定change事件，将表单元素的值设置为state的值（控制表单元素值的变化）

```

state = { txt: '' }

<input type="text" value={this.state.txt}
      onChange={e => this.setState({ txt: e.target.value })}>
/>

```

示例：

```

import React from 'react';
import ReactDOM from 'react-dom';

class Form extends React.Component {
  state = {
    txt: '',
    content: '',
    city: 'sh',
    isChecked: false
  }
  //处理文本框的变化
  handleChange = e => {
    this.setState({
      txt:e.target.value
    })
  }
  //处理富文本框的变化
  handleContent = e => {
    this.setState({
      content:e.target.value
    })
  }
  //处理下拉框的变化
  handleCity = e => {
    this.setState({
      city:e.target.value
    })
  }
  //处理复选框的变化
  handleChecked = e => {
    this.setState({
      isChecked:e.target.checked
    })
  }
  render() {
    return (
      <div>
        {/* 文本框 */}
        <input type="text" value={this.state.txt} onChange={this.handleChange} />
        <br/>
        {/* 富文本框 */}
        <textarea value={this.state.content} onChange={this.handleContent} />
        <br/>
        {/* 下拉框 */}
        <select value={this.state.city} onChange={this.handleCity}>
          <option value="sh">上海</option>
          <option value="bj">北京</option>
          <option value="gz">广州</option>
        </select>
        <br/>
        {/* 复选框 */}
        <input type="checkbox" checked={this.state.isChecked} onChange={this.handleChecked} />
      </div>
    )
  }
}

ReactDOM.render(<Form />, document.getElementById('root'))

```

多表单元素优化：

- 问题：每个表单元素都有一个单独的事件处理程序，处理太繁琐
- 优化：使用一个事件处理程序同时处理多个表单元素

多表单元素优化步骤：

- 给表单元素添加name属性，名称与state相同
- 根据表单元素类型获取对应值
- 在change事件处理程序中通过[name]来修改对应的state

```
<input  
  type="text"  
  name="txt"  
  value={this.state.txt}  
  onChange={this.handleForm}  
/>  
  
// 根据表单元素类型获取值  
const value = target.type === 'checkbox'  
  ? target.checked  
  : target.value  
  
// 根据name设置对应state  
this.setState({  
  [name]: value  
})
```

1. 非受控组件 (DOM方式)

- 说明：借助于ref，使用原生DOM方式来获取表单元素值
- ref的作用：获取DOM或组件

使用步骤：

- 调用React.createRef()方法创建一个ref对象

```
constructor() {  
  super()  
  this.txtRef = React.createRef()  
}
```

- 将创建好的ref对象添加到文本框中

```
<input type="text" ref={this.txtRef} />
```

- 通过ref对象获取到文本框的值

```
Console.log(this.txtRef.current.value)
```

案例

- 渲染评论列表（列表渲染）
- 没有评论数据时渲染：暂无评论（条件渲染）
- 获取评论信息，包括评论人和评论内容（受控组件）
- 发表评论，更新评论列表（setState()）

The screenshot shows a user interface for managing comments. On the right, there is a list of comments with a header '评论列表' (Comment List). The comments are listed as follows:

- 评论人 : jack
评论内容 : 沙发！！！
- 评论人 : rose
评论内容 : 板凳~
- 评论人 : tom
评论内容 : 楼主好人

On the left, there is a form for adding a new comment. It consists of two input fields: '请输入评论人' (Please enter commentator) and '请输入评论内容' (Please enter comment content), and a '发表评论' (Post comment) button.

```
import React from 'react';
```

```
import ReactDOM from 'react-dom';
import './index.css';
class App extends React.Component {
  //初始化状态
  state = {
    comments: [
      {id: 1, name: 'jack', content: '沙发！！！'},
      {id: 2, name: 'rose', content: '板凳~'},
      {id: 3, name: 'tom', content: '楼主好人'}
    ],
    //评论人
    userName: '',
    //评论内容
    userContent: ''
  }
  //渲染评论列表
  renderList() {
    if (this.state.comments.length === 0) {
      return <div className='no-comment'>暂无评论，快去评论吧~</div>
    }
    return (
      <ul>
        {this.state.comments.map(item => (
          <li key={item.id}>
            <h3>评论人: {item.name}</h3>
            <p>评论内容: {item.content}</p>
          </li>
        ))}
      </ul>
    )
    // return this.state.comments.length === 0 ? (
    //   <div className='no-comment'>暂无评论，快去评论吧~</div>
    // ) : (
    //   <ul>
    //     {this.state.comments.map(item => (
    //       <li key={item.id}>
    //         <h3>评论人: {item.name}</h3>
    //         <p>评论内容: {item.content}</p>
    //       </li>
    //     )))
    //   </ul>
    // )
  }
  //处理表单元素值
  handleForm = (e) => {
    const {name, value} = e.target
    this.setState({
      [name]: value
    })
  }
  //发表评论
  addComment = (e) => {
    const { comments,userName, userContent} = this.state

    //非空校验
    if (userName.trim() === '' || userContent.trim() === ''){
      alert('请输入评论人和评论内容')
      return
    }
    //将评论信息添加到state中
    const newComments = [
      {
        id: Math.random(),
        name: userName,
        content: userContent
      },
      ...comments
    ]
    //文本框的值如何清空？要清空文本框只需要将其对应的state清空即可
    this.setState({
      comments: newComments,
      userName: '',
      userContent: ''
    })
  }
  render() {
    const { userName, userContent} = this.state
```

```

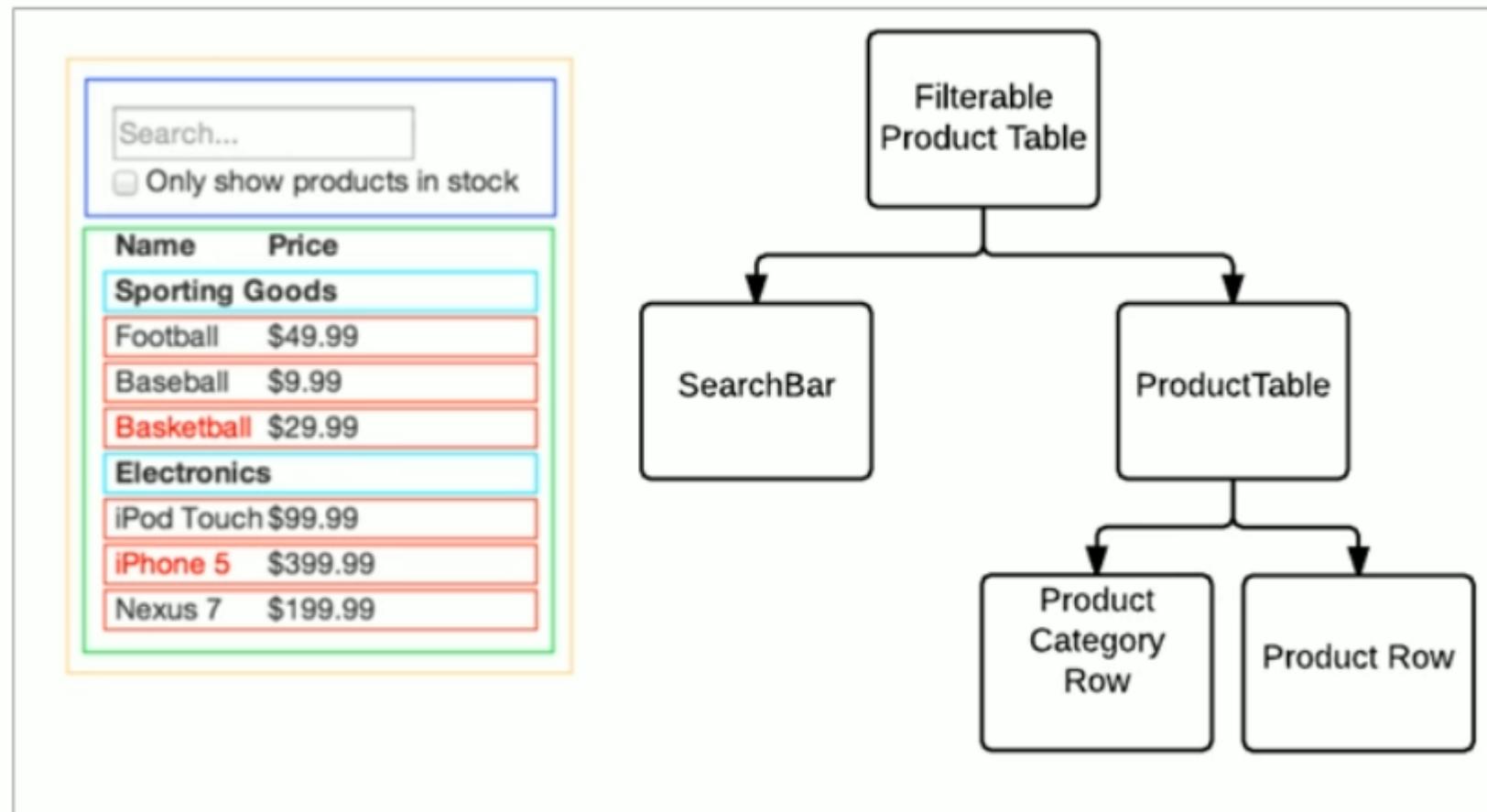
return (
  <div className='app'>
    <div>
      <input className='user' type='text' placeholder='请输入评论人' value={userName} name='userName'
        onChange={this.handleForm}></input>
      <br />
      <textarea
        className='content'
        cols='30'
        rows='10'
        placeholder='请输入评论内容'
        value={userContent}
        name='userContent'
        onChange={this.handleForm}>
      />
      <br />
      <button onClick={this.addComment}>发表评论</button>
    </div>
    {/* 通过条件渲染决定渲染什么内容： */}
    {this.renderList()}
  </div>
)
}
ReactDOM.render(<App />, document.getElementById('root'))

```

React组件进阶

➤ 组件通讯介绍

组件是独立且封闭的单元，默认情况下，只能使用组件自己的数据。在组件化过程中，我们将一个完整功能拆分成多个组件，以更好的完成整个应用的功能。而在过程中，多个组件之间不可避免的要共享某些数据。为了实现这些功能，就需要打破组件的独立封闭性，让其于外界沟通。这个过程就是组件通讯。



➤ 组件的props

- 组件时封闭的，要接收外部数据应该通过props来实现
- **props的作用：**接收传递给组件的数据
- **传递数据：**给组件标签添加属性
- **接收数据：**函数组件通过参数**props**接收数据，类组件通过**this.props**接收数据

```
<Hello name="jack" age={19} />
```

```
function Hello(props) {
  console.log(props)
  return (
    <div>接收到数据: {props.name}</div>
  )
}
```

```
class Hello extends React.Component {
  render() {
    return (
      <div>接收到的数据: {this.props.age}</div>
    )
  }
}
<Hello name="jack" age={19} />
```

特点：

1. 可以给组件传递任意类型的数据
2. **props是只读的**对象，只能读取属性的值，无法修改对象
3. 注意：使用类组件时，如果写了构造函数，**应该将props传递给super()**，否则，无法在构造函数中获取到props！

```
class Hello extends React.Component {
  constructor(props) {
    // 推荐将props传递给父类构造函数
    super(props)
  }
  render() {
    return <div>接收到的数据: {this.props.age}</div>
  }
}
```

➤ 组件通讯的三种方式

组件之间的通讯分为3种

1. 父组件 —> 子组件
2. 子组件 —> 父组件
3. 兄弟组件

父组件 —> 子组件

1. 父组件提供要传递的state数据
2. 给子组件标签添加属性，值为state中的数据
3. 子组件中通过props接收父组件中传递的数据

```
class Parent extends React.Component {
  state = { lastName: '王' }
  render() {
    return (
      <div>
        传递数据给子组件: <Child name={this.state.lastName} />
      </div>
    )
  }
}
```

```
function Child(props) {
  return <div>子组件接收到数据: {props.name}</div>
}
```

子组件 → 父组件

思路：利用回调函数，父组件提供回调，子组件调用，将要传递的数据作为回调函数的参数。

1. 父组件提供一个回调函数（用于接收数据）
2. 将该函数作为属性的值，传递给子组件
3. 子组件通过props调用回调函数
4. 子组件的数据作为参数传递给回调函数

```
class Parent extends React.Component {
  getChildMsg = (msg) => {
    console.log('接收到子组件数据', msg)
  }
  render() {
    return (
      <div>
        子组件: <Child getMsg={this.getChildMsg} />
      </div>
    )
  }
}
```

```
class Child extends React.Component {
  state = { childMsg: 'React' }
  handleClick = () => {
    this.props.getMsg(this.state.childMsg)
  }
  return (
    <button onClick={this.handleClick}>点我，给父组件传递数据</button>
  )
}
```

兄弟组件

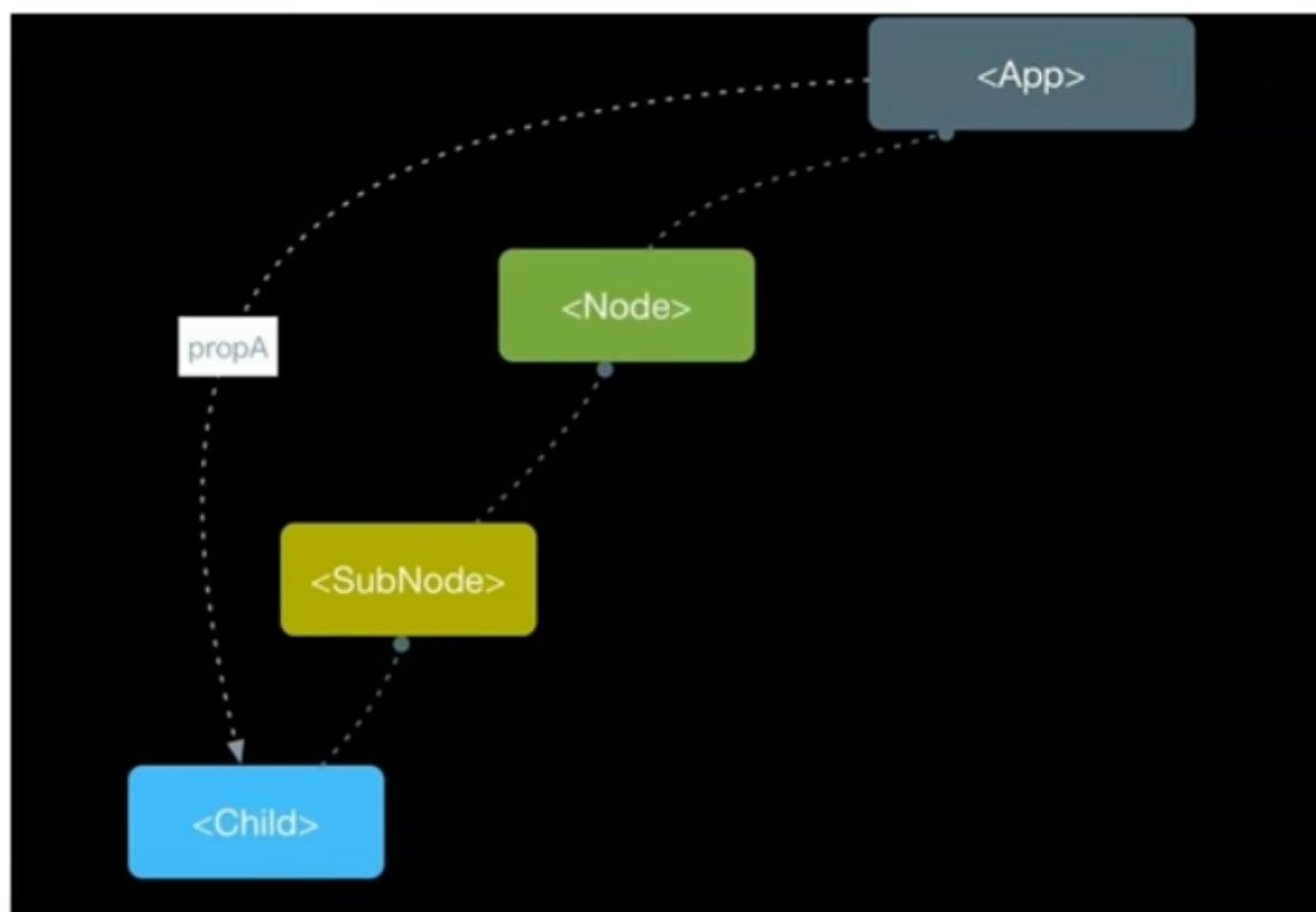
- 将共享状态提升到最近的公共父组件中，有公共父组件管理这个状态
- 思想：状态提升
- 公共父组件职责：1. 提供共享状态 2. 提供操作共享状态的方法
- 要通讯的子组件只需要通过props接收状态或操作状态的方法



>Context

思考：App组件要传递数据给Child组件，该如何处理？

- 更好的姿势：使用Context
- 作用：跨组件传递数据（比如主题，语言等）



1. 使用步骤React.createContext()创建Provider（提供数据）和Consumer（消费数据）两个组件。

```
const { Provider, Consumer } = React.createContext()
```

2. 使用Provider组件作为父节点

```
<Provider>
  <div className="App">
    <Child1 />
  </div>
</Provider>
```

3. 设置value属性，表示要传递的数据

```
<Provider value="pink">
```

4. 调用Consumer组件接收数据

```
<Consumer>
  {data => <span>data参数表示接收到的数据 -- {data}</span>}
</Consumer>
```

总结：

1. 如果两个组件是远方亲戚（比如，嵌套多层）可以使用Context实现组件通讯
2. Context提供了两个组件：Provider 和 Consumer
3. Provider组件：用来提供数据
4. Consumer组件：用来消费数据

➤ props深入

children属性

children属性：表示组件标签的子节点时，props就会有该属性

children属性与普通的props一样，值可以是任意值（文本，React元素，组件，甚至是函数）

```
function Hello(props) {
  return (
    <div>
      组件的子节点：{props.children}
    </div>
  )
}

<Hello>我是子节点</Hello>
```

props校验

- 对于组件来说，props是外来的，无法保证组件使用者传入什么格式的数据
- 如果传入的数据格式不对。可能会导致组件内部报错
- 关键问题：组件的使用者不知道明确的错误原因

```
// 小明创建的组件App
function App(props) {
  const arr = props.colors
  const lis = arr.map((item, index) => <li key={index}>{item.name}</li>)
  return (
    <ul>{lis}</ul>
  )
}

// 小红使用组件App
<App colors={19} />
```

props校验：允许在创建组件的时候，就指定props的类型，格式等

作用：捕获使用组件时因为props导致的错误，给出明确的错误提示，增加组件的健壮性

```
App.propTypes = {
  colors: PropTypes.array
}
```

✖ Warning: Failed prop type: Invalid prop `colors` of type [index.js:1375](#) `number` supplied to `App`, expected `array`.
in App (at [src/index.js:21](#))

使用步骤

1. 安装包prop-types (yarn add prop-types/npm i prop-types)
2. 导入prop-types包
3. 使用组件名.propTypes = {}来给组件的props添加校验规则
4. 校验规则通过PropTypes对象来指定

```
import PropTypes from 'prop-types'
function App(props) {
  return (
    <h1>Hi, {props.colors}</h1>
  )
}
App.propTypes = {
  // 约定colors属性为array类型
  // 如果类型不对，则报出明确错误，便于分析错误原因
  colors: PropTypes.array
}
```

约束规则

常见类型: array, bool, func, number, object, string

React元素类型: element

必填项:.isRequired

特定结构的对象: shape({})

```

// 常见类型
optionalFunc: PropTypes.func,
// 必选
requiredFunc: PropTypes.func.isRequired,
// 特定结构的对象
optionalObjectWithShape: PropTypes.shape({
  color: PropTypes.string,
  fontSize: PropTypes.number
})

const App = props => {
  return (
    <div>
      <h1>props校验: </h1>
    </div>
  )
}

// 添加props校验
// 属性 a 的类型: 数值 (number)
// 属性 fn 的类型: 函数 (func) 并且为必填项
// 属性 tag 的类型: React元素 (element)
// 属性 filter 的类型: 对象 ({area: '上海', price: 1999})
App.propTypes = {
  a: PropTypes.number,
  fn: PropTypes.func.isRequired,
  tag: PropTypes.element,
  filter: PropTypes.shape({
    area: PropTypes.string,
    price: PropTypes.number
  })
}

ReactDOM.render(<App fn={() => {}} />, document.getElementById('root'))

```

props的默认值

场景：分页组件 ---> 每页显示条数

作用：给props设置默认值，在未传入props时生效

```

function App (props) {
  return (
    <div>
      此处展示props的默认值: {props.pageSize}
    </div>
  )
}

// 设置默认值
App.defaultProps = {
  pageSize: 10
}

// 不传入pageSize属性
<App />

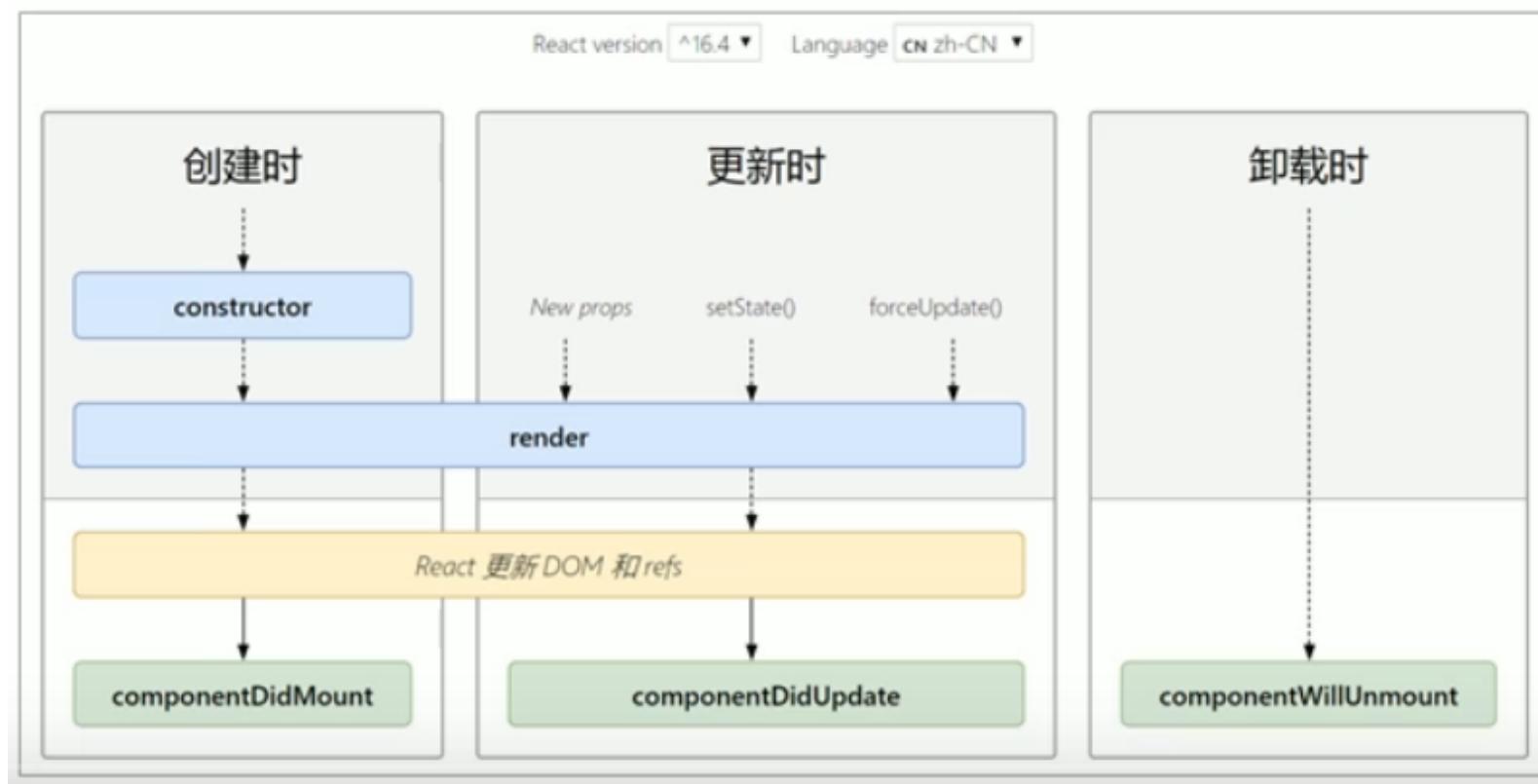
```

➤组件的生命周期

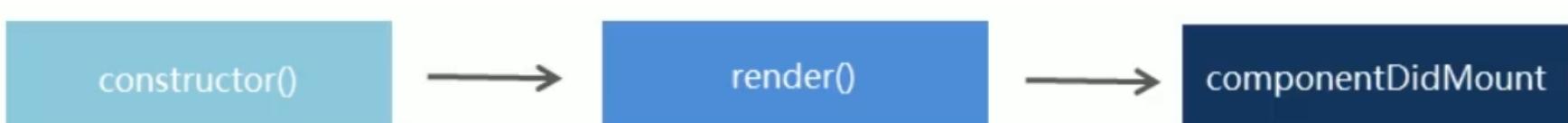
- 意义：组件的生命周期有助于理解组件的运行方式，完成更复杂的组件功能，分析组件错误原因等
- **组件的生命周期**：组件从被创建到挂载到页面中运行，再到组件不用时卸载的过程
- 生命周期的每个阶段总是伴随着一些方法调用，这些方法就是生命周期的**钩子函数**
- 钩子函数的作用：为开发人员在不用阶段操作组件提供了时机
- **只有类组件才有生命周期**

生命周期的三个阶段

1. 每个阶段的执行时机
2. 每个阶段钩子函数的执行顺序
3. 每个阶段钩子函数的作用



1. 创建时 (挂载阶段)
2. 执行时机：组件创建时 (页面加载时)
3. 执行顺序：

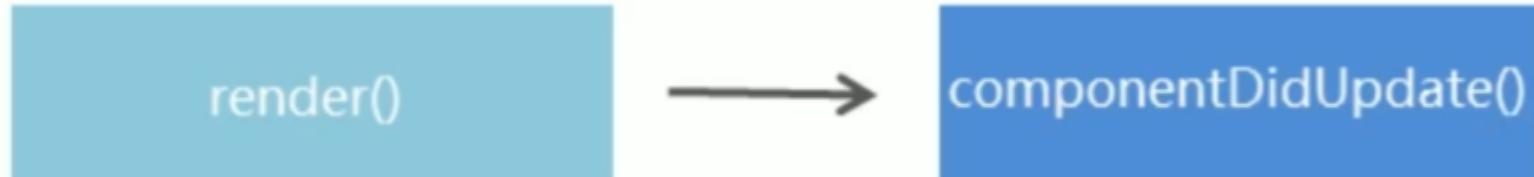


钩子函数	触发时机	作用
------	------	----

constructor	创建组件时，最先执行	1. 初始化state 2. 为事件处理程序绑定this
render	每次组件渲染都会触发	渲染UI (注意: 不能调用setState())
componentDidMount	组件挂载 (完成DOM) 后	1. 发送网络请求 2. DOM操作

更新时 (更新阶段)

- 执行时机: 1.setState() 2.forceUpdate() 3.组件接收到新的props
- 说明: 以上三者任意一种变化, 组件就会重新渲染
- 执行顺序:



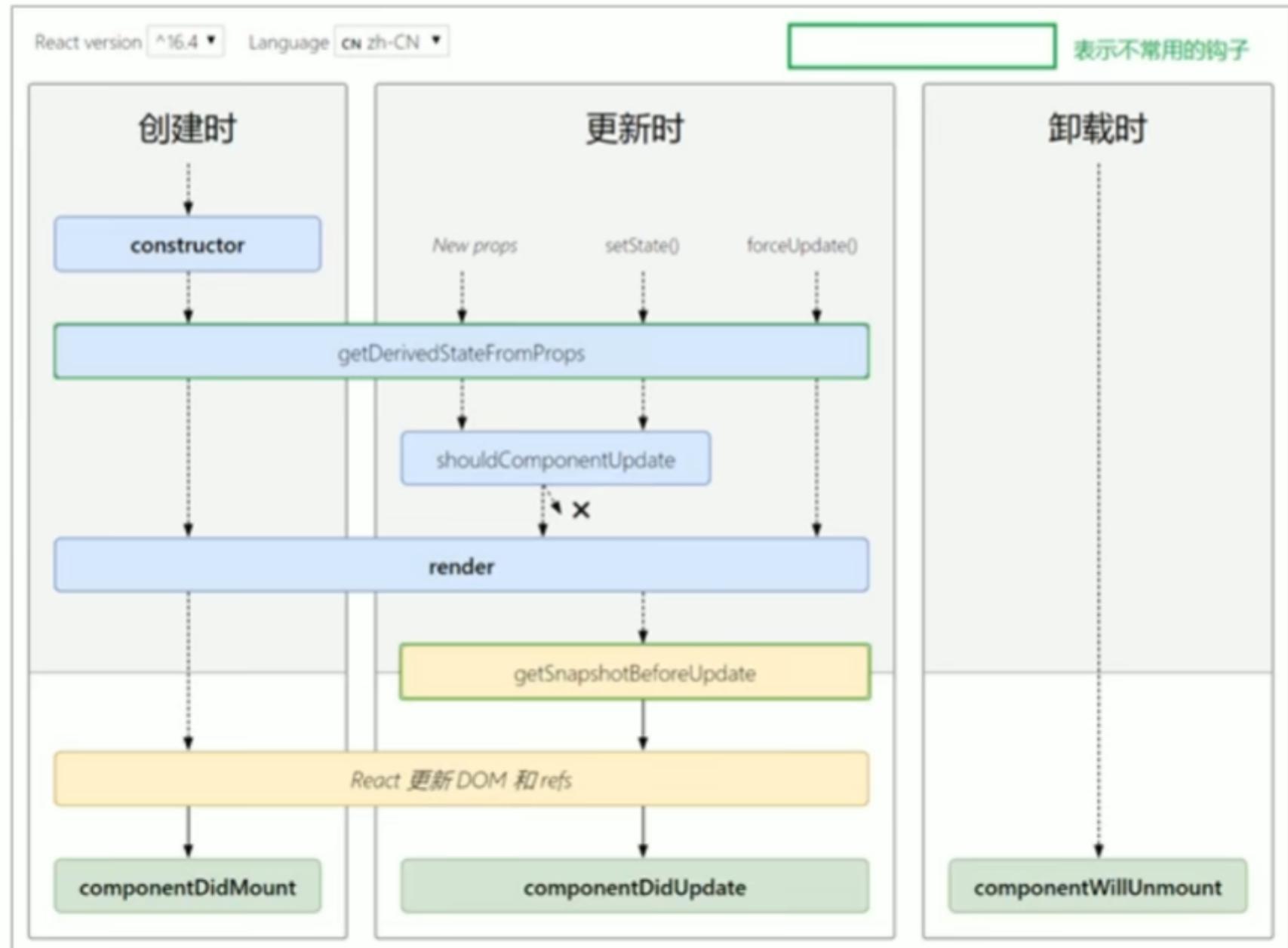
钩子函数	触发时机	作用
render	每次组件渲染都会触发	渲染UI (与挂载阶段是同一个render)
componentDidUpdate	组件更新 (完成DOM渲染) 后	1. 发送网络请求 2. DOM操作 注意: 如果要setState()必须放在一个if条件中

卸载时 (卸载阶段)

执行时机: 组件从页面中消失

钩子函数	触发时机	作用
componentWillUnmount	组件卸载 (从页面中消失)	执行清理工作 (比如: 清理定时器等)

不常用钩子函数



> render-props和高阶组件

React组件复用概述

- 思考：如果两个组件中的部分功能相似或相同，该如何处理？
- 处理方式：**复用**相似的功能（联想函数封装）
- 复用什么？1.**state** 2.**操作state的方法**（组件状态逻辑）
- 两种方式：1.**render props模式** 2.**高阶组件 (HOC)**
- 注意：这两种方式**不是新的API**，而是利用React自身特点的编码技巧，演化而成的固定模式（写法）

render props模式

- 思路：将要复用的state和操作state的方法封装到一个组件中
- 问题1：如何拿到该组件中复用的state
- 在使用组件时，添加一个值为**函数的prop**。通过**函数参数**来获取（需要组件内部实现）
- 问题2：如何渲染任意UI？
- 使用该**函数的返回值**作为要渲染的UI内容（需要组件内部实现）

```

<Mouse render={(mouse) => {}}/>
```
<Mouse render={(mouse) => (
 <p>鼠标当前位置 {mouse.x} , {mouse.y}</p>
)} />

```

使用步骤

- 创建Mouse组件，在组件中提供复用的**状态逻辑**代码（1.状态 2.操作状态的方法）
- 将要**复用的状态**作为props.render(state)方法的参数，暴露到组件外部
- 使用props.render()的**返回值**作为要渲染的内容

```
class Mouse extends React.Component {
 // ... 省略state和操作state的方法
 render() {
 return this.props.render(this.state)
 }
}
```

```
<Mouse render={(mouse) => <p>鼠标当前位置 {mouse.x}, {mouse.y}</p>}/>
```

children代替render属性

- 注意：并不是该模式叫render props就必须使用名为render的prop，实际上可以使用任意名称的prop
- 把prop是一个函数并且告诉组件要渲染什么内容的技术叫做：render props模式
- 推荐：使用children代替render属性

```
<Mouse>
 {({x, y}) => <p>鼠标的位置是 {x}, {y}</p> }
</Mouse>
// 组件内部：
this.props.children(this.state)
```

```
// Context 中的用法：
<Consumer>
 {data => data参数表示接收到的数据 -- {data}}
</Consumer>
```

代码优化

- 推荐：给render props模式添加props校验

```
Mouse.propTypes = {
 children: PropTypes.func.isRequired
}
```

- 应该在组件卸载时解除mousemove事件绑定

```
componentWillUnmount() {
 window.removeEventListener('mousemove', this.handleMouseMove)
}
```

高阶组件

目的：实现状态逻辑复用

采用包装（装饰）模式，比如说：手机壳

手机：获取保护功能

手机壳：提供保护功能

高阶组件就相当于手机壳，通过包装组件，增强组件功能

高阶组件（HOC, Higher-Order Component）是一个函数，接收要包装的组件，返回增强后的组件

高阶组件内部创建一个类组件，在这个类组件中提供复用的状态逻辑代码，通过prop将复用的状态传递给被包装组件 WrappedComponent

```
const EnhancedComponent = withHOC(WrappedComponent)
```

```
// 高阶组件内部创建的类组件:
class Mouse extends React.Component {
 render() {
 return <WrappedComponent {...this.state} />
 }
}
```

## 使用步骤

1. 创建一个函数，名称约定以with开头
2. 指定函数参数，参数应该以大写字母开头（作为要渲染的组件）
3. 在函数内部创建一个类组件，提供复用的状态逻辑代码，并返回
4. 在该组件中，渲染参数组件，同时将状态通过prop传递给参数组件
5. 调用该高阶组件，传入要增强的组件，通过返回值拿到增强后的组件，并将其渲染到页面中

```
function withMouse(WrappedComponent) {
 class Mouse extends React.Component {}
 return Mouse
}
```

```
// Mouse组件的render方法中:
return <WrappedComponent {...this.state} />
```

```
// 创建组件
const MousePosition = withMouse(Position)

// 渲染组件
<MousePosition />
```

## 设置displayName

- 使用高阶组件存在的问题：得到的两个组件名称相同
- 原因：默认情况下，React使用组件名称作为displayName
- 解决方式：为高阶组件设置displayName便于调试时区分不同的组件
- displayName的作用：用于设置调试信息（React Developer Tools信息）
- 设置方式

```
Mouse.displayName = `WithMouse${getDisplayName(WrappedComponent)}`

function getDisplayName(WrappedComponent) {
 return WrappedComponent.displayName || WrappedComponent.name || 'Component'
}
```

## 传递props

- 问题：props丢失
- 原因：高阶组件没有往下传递props

- 解决方法：渲染WrappedComponent时，将state和this.props一起传递给组件
- 传递方式：

```
<WrappedComponent {...this.state} {...this.props} />
```

- 

未完

## React原理

- setState()说明
- JSX语法的转化过程
- 组件更新机制
- 组件性能优化
- 虚拟DOM和Diff算法