

React基础

1. 课程目标

1. 入门React，了解常规用法；
2. 掌握面试中React的基础问题；
3. 掌握React学习路线；

2. 课程大纲

- React简介
- JSX模板语法
- props & state
- 生命周期
- 事件处理
- 条件渲染
- 列表

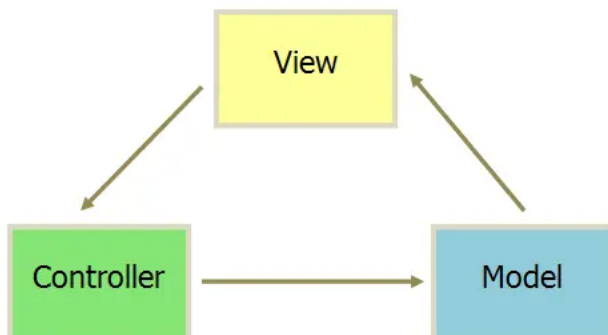
3. 主要内容

3.1. React简介

React 是一个**声明式**，高效且灵活的用于构建用户界面的 JavaScript 库。使用 React 可以将一些简短、独立的代码片段组合成复杂的 UI 界面，这些代码片段被称作“**组件**”。

ui = render (data) -> 单向数据流

- MVC



```
1 // model
2 var myapp = {}; // 创建这个应用对象
3
4 myapp.Model = function() {
5     var val = 0;
6
7     this.add = function(v) {
8         if (val < 100) val += v;
9     };
10
11     this.sub = function(v) {
12         if (val > 0) val -= v;
13     };
14
15     this.getVal = function() {
16         return val;
17     };
18
19     /* 观察者模式 */
20     var self = this,
21         views = [];
22
23     this.register = function(view) {
24         views.push(view);
25     };
26
27     this.notify = function() {
28         for(var i = 0; i < views.length; i++) {
29             views[i].render(self);
30         }
31     };
32 };
33
34 // view
35 myapp.View = function(controller) {
36     var $num = $('#num'),
37         $incBtn = $('#increase'),
38         $decBtn = $('#decrease');
39
40     this.render = function(model) {
41         $num.text(model.getVal() + 'rmb');
42     };
43
44     /* 绑定事件 */
45     $incBtn.click(controller.increase);
```

```

46     $decBtn.click(controller.decrease);
47 };
48
49 // controller
50 ▼ myapp.Controller = function() {
51     var model = null,
52         view = null;
53
54 ▼     this.init = function() {
55         /* 初始化Model和View */
56         model = new myapp.Model();
57         view = new myapp.View(this);
58
59         /* View向Model注册, 当Model更新就会去通知View啦 */
60         model.register(view);
61         model.notify();
62     };
63
64     /* 让Model更新数值并通知View更新视图 */
65 ▼     this.increase = function() {
66         model.add(1);
67         model.notify();
68     };
69
70 ▼     this.decrease = function() {
71         model.sub(1);
72         model.notify();
73     };
74 };
75
76 // init
77 ▼ (function() {
78     var controller = new myapp.Controller();
79     controller.init();
80 })();

```

- MVVM



```
1  // model
2  var data = {
3    val: 0
4  };
5
6  // view
7  <div id="myapp">
8    <div>
9      <span>{{ val }}rmb</span>
10    </div>
11    <div>
12      <button v-on:click="sub(1)">-</button>
13      <button v-on:click="add(1)">+</button>
14    </div>
15  </div>
16
17  // controller
18  new Vue({
19    el: '#myapp',
20    data: data,
21    methods: {
22      add(v) {
23        if(this.val < 100) {
24          this.val += v;
25        }
26      },
27      sub(v) {
28        if(this.val > 0) {
29          this.val -= v;
30        }
31      }
32    }
33  });
34
35  // Vue是不是MVVM? React呢?
36  // 严格来讲都不是
37  // React: ui = render (data) 单向数据流
38  // Vue:   ref 直接操作DOM, 跳过了ViewModel
```

3.2. JSX模板语法

JSX称为JS的语法扩展，将UI与逻辑层耦合在组件里，用{}标识

因为 JSX 语法上更接近 JS 而不是 HTML，所以使用 camelCase（小驼峰命名）来定义属性的名称；JSX 里的 class 变成了 `className`，而 tabindex 则变为 `tabIndex`。

3.2.1. JSX支持表达式

支持JS表达式，变量，方法名

JSX | 复制代码

```
1  // 变量
2  const name = 'Josh Perez';
3  const element = <h1>Hello, {name}</h1>;
4
5  function formatName(user) {
6    return user.firstName + ' ' + user.lastName;
7  }
8
9  // 方法
10 const user = {
11   firstName: 'Harper',
12   lastName: 'Perez'
13 };
14
15 const element = (
16   <h1>
17     Hello, {formatName(user)}!
18   </h1>
19 );
20
21 function getGreeting(user) {
22   if (user) {
23     return <h1>Hello, {formatName(user)}!</h1>;
24   }
25   return <h1>Hello, Stranger.</h1>;
26 }
```

3.2.2. JSX指定属性

```
1  const element = <img src={user.avatarUrl}></img>;
2
3  注意: JSX支持防注入(防止XSS攻击)
4  const title = response.potentiallyMaliciousInput; // 此时只是字符串
5  // 直接使用是安全的: const element = <h1>{title}</h1>;
6
7  React 如何预防XSS
8
9  // 反射型 XSS
10
11 https://xxx.com/search?query=userInput
12
13 // 服务器在对此 URL 的响应中回显提供的搜索词: query=123
14 <p>您搜索的是: 123</p>
15
16 // https://xxx.com/search?query=
17 <p>您搜索的是: </p>
18 // 如果有用户请求攻击者的 URL , 则攻击者提供的脚本将在用户的浏览器中执行。
19
20
21 // 存储型 XSS, 存储到目标数据库
22 // 评论输入, 所有访问用户都能看到了
23 ▼ <textarea>
24     
25 </textarea>
26
27 // 部分源码
28 for (index = match.index; index < str.length; index++) {
29 ▼   switch (str.charCodeAt(index)) {
30       case 34: // "
31         escape = '&quot;';
32         break;
33       case 38: // &
34         escape = '&amp;';
35         break;
36       case 39: // '
37         escape = '&#x27;';
38         break;
39       case 60: // <
40         escape = '&lt;';
41         break;
42       case 62: // >
43         escape = '&gt;';
44         break;
```

```

45     default:
46         continue;
47     }
48 }
49
50 // 一段恶意代码
51 
52 // React 在渲染到浏览器前进行的转义，可以看到对浏览器有特殊含义的字符都被转义了，恶意
    代码在渲染到 HTML 前都被转成了字符串
53 &lt;img src=&quot;empty.png&quot; onerror
    =&quot;alert(&#x27;xss&#x27;)&quot;&gt;
54
55 // JSX
56 const element = (
57   <h1 className="greeting">
58     Hello, world!
59   </h1>
60 );
61
62 // 通过 babel 编译后的代码
63 const element = React.createElement(
64   'h1',
65   {className: 'greeting'},
66   'Hello, world!'
67 );
68
69 // React.createElement() 方法返回的 ReactElement
70 const element = {
71   $$typeof: Symbol('react.element'),
72   type: 'h1',
73   key: null,
74   props: {
75     children: 'Hello, world!',
76     className: 'greeting'
77   }
78   ...
79 }
80
81 // 如何模拟一个Children会如何?
82 const storedData = `{
83   "ref":null,
84   "type":"body",
85   "props":{
86     "dangerouslySetInnerHTML":{
87       "__html":"<img src=\"empty.png\" onerror =\"alert('xss')\"/>"
88     }
89   }
90 }`;

```

```
91 // 转成 JSON
92 const parsedData = JSON.parse(storedData);
93 // 将数据渲染到页面
94 render () {
95     return <span> {parsedData} </span>;
96 }
97
98 // $$typeof 是用来标记一个ReactElement的, JSON化后Symbol会丢失, React会报错
```

3.2.3. JSX表示对象

JSX | 复制代码

```
1  const element = (
2    <h1 className="greeting">
3      Hello, world!
4    </h1>
5  );
6
7  // 等同于React.createElement
8  const element = React.createElement(
9    'h1',
10    {className: 'greeting'},
11    'Hello, world!'
12  );
13
14  const element = {
15    type: 'h1',
16    props: {
17      className: 'greeting',
18      children: 'Hello, world!'
19    }
20  };
```

3.2.4. 将JSX渲染为DOM


```
1 // 使用ReactDOM.render
2 const element = <h1>Hello, world</h1>;
3 ReactDOM.render(element, document.getElementById('root'));
4
5 // render只能代表当前时刻的状态
6 // 更新元素 只能再次 ReactDOM.render
7 function tick() {
8   const element = (
9     <div>
10       <h1>Hello, world!</h1>
11       <h2>It is {new Date().toLocaleTimeString()}.</h2>
12     </div>
13   );
14   ReactDOM.render(element, document.getElementById('root'));
15 }
16
17 setInterval(tick, 1000); // 不建议多次render
```

3.2.5. JSX转JS

JSX可以当做语法糖，可以在babel官网中尝试，<https://babeljs.io/repl>

可以使用官网提供的create-react-app npm run eject 来看babelrc中的配置，主要使用

<https://www.babeljs.cn/docs/babel-preset-react>

```
1 // 安装babel 及react 的依赖
2 npm install core-js @babel/core @babel/preset-env @babel/preset-react
  @babel/register babel-loader @babel/plugin-transform-runtime --save-dev
3
4 .babelrc
5 {
6   "presets" : [
7     "@babel/preset-env" ,
8     "@babel/preset-es2015",
9     "@babel/preset-react"
10  ],
11  "plugins" : [
12    "@babel/plugin-transform-runtime"
13  ]
14 }
```

3.3. props及state

组件，从概念上类似于 JavaScript 函数。它接受任意的入参（即“props”），并返回用于描述页面展示内容的 React 元素。

3.3.1. 组件

- 函数式组件
- Class类组件

```
1 function Welcome(props) {
2   return <h1>Hello, {props.name}</h1>;
3 }
4
5 class Welcome extends React.Component {
6   render() {
7     return <h1>Hello, {this.props.name}</h1>;
8   }
9 }
```

3.3.1.1. 渲染组件

```
1 ▾ function Welcome(props) {  
2   return <h1>Hello, {props.name}</h1>;  
3 }  
4  
5 const element = <Welcome name="Sara" />;  
6 ReactDOM.render(  
7   element,  
8   document.getElementById('root')  
9 );  
10  
11 // 自定义组件使用大写字母开头  
12 import React from 'react';  
13  
14 // 正确! 组件需要以大写字母开头:  
15 ▾ function Hello(props) {  
16   // 正确! 这种 <div> 的使用是合法的, 因为 div 是一个有效的 HTML 标签:  
17   return <div>Hello {props.toWhat}</div>;  
18 }  
19  
20 ▾ function HelloWorld() {  
21   // 正确! React 知道 <Hello /> 是一个组件, 因为它是大写字母开头的:  
22   return <Hello toWhat="World" />;  
23 }
```

3.3.1.2. 组件的组合与拆分

```
1 // 页面内多次引用
2 <div>
3   <Welcome name="Sara" />
4   <Welcome name="Cahal" />
5   <Welcome name="Edite" />
6 </div>
7
8 function Comment(props) {
9   return (
10     <div className="Comment">
11       <div className="UserInfo">
12         <img className="Avatar"
13           src={props.author.avatarUrl}
14           alt={props.author.name}
15         />
16       <div className="UserInfo-name">
17         {props.author.name}
18       </div>
19     </div>
20     <div className="Comment-text">
21       {props.text}
22     </div>
23     <div className="Comment-date">
24       {formatDate(props.date)}
25     </div>
26   </div>
27 );
28 }
29
30 // 拆分后为
31 function Comment(props) {
32   return (
33     <div className="Comment">
34       <UserInfo user={props.author} />
35       <div className="Comment-text">
36         {props.text}
37       </div>
38       <div className="Comment-date">
39         {formatDate(props.date)}
40       </div>
41     </div>
42   );
43 }
```

3.3.2. props



JSX

复制代码

```
1  所有 React 组件都必须像纯函数一样保护它们的 props 不被更改。
2
3  // 错误, 要像纯函数一样幂等
4  function withdraw(account, amount) {
5    account.total -= amount;
6  }
```

3.3.3. state

```
1 // 使用props形式
2 function Clock(props) {
3   return (
4     <div>
5       <h1>Hello, world!</h1>
6       <h2>It is {props.date.toLocaleTimeString()}</h2>
7     </div>
8   );
9 }
10
11 function tick() {
12   ReactDOM.render(
13     <Clock date={new Date()} />,
14     document.getElementById('root')
15   );
16 }
17
18 setInterval(tick, 1000);
19
20 // 如何避免多次React.DOM render?
21
22 // 引用生命周期，根组件保留一个
23 class Clock extends React.Component {
24   constructor(props) {
25     super(props);
26     this.state = {date: new Date()};
27   }
28
29   componentDidMount() {
30     this.timerID = setInterval(
31       () => this.tick(),
32       1000
33     );
34   }
35
36   componentWillUnmount() {
37     clearInterval(this.timerID);
38   }
39
40   tick() {
41     this.setState({
42       date: new Date()
43     });
44   }
45 }
```

```
46 ▼ render() {
47     return (
48         <div>
49 ▼         <h1>Hello, world!</h1>
50 ▼         <h2>It is {this.state.date.toLocaleTimeString()}</h2>
51         </div>
52     );
53 }
54 }
55
56 ReactDOM.render(
57     <Clock />,
58     document.getElementById('root')
59 );
```

```
1  1. setState
2  构造函数是唯一可以给state赋值的地方
3  this.setState({comment: 'Hello'});
4
5  2. state更新可能是异步的
6  // Wrong
7  this.setState({
8    counter: this.state.counter + this.props.increment,
9  });
10 // Correct
11 this.setState(function(state, props) {
12   return {
13     counter: state.counter + props.increment
14   };
15 });
16
17 3. state更新会合并
18 constructor(props) {
19   super(props);
20   this.state = {
21     posts: [],
22     comments: []
23   };
24 }
25
26 componentDidMount() {
27   fetchPosts().then(response => {
28     // 相当于{post: response.posts, ...otherState}
29     this.setState({
30       posts: response.posts
31     });
32   });
33
34   fetchComments().then(response => {
35     this.setState({
36       comments: response.comments
37     });
38   });
39 }
40
41 4. 单向数据流
42 state 只在当前的组件里生效，属于组件内的属性，重复实例化相同的组件，内部的内存地址也是不一样的；
43 例如Clock中计时器都是独立的
```



```
1 // setState 异步
2 // 异步目的: batch 处理, 性能优化
3 1. 合成事件
4 class App extends Component {
5
6     state = { val: 0 }
7
8     increment = () => {
9         this.setState({ val: this.state.val + 1 })
10        console.log(this.state.val) // 输出的是更新前的val --> 0
11    }
12
13    render() {
14        return (
15            <div onClick={this.increment}>
16                {`Counter is: ${this.state.val}`}
17            </div>
18        )
19    }
20 }
21
22 2. 生命周期
23 class App extends Component {
24
25     state = { val: 0 }
26
27     componentDidMount() {
28         this.setState({ val: this.state.val + 1 })
29         console.log(this.state.val) // 输出的还是更新前的值 --> 0
30     }
31
32     render() {
33         return (
34             <div>
35                 {`Counter is: ${this.state.val}`}
36             </div>
37         )
38     }
39 }
40
41 3. 原生事件
42 class App extends Component {
43
44     state = { val: 0 }
45
46     changeValue = () => {
```

```

46     this.setState({ val: this.state.val + 1 })
47     console.log(this.state.val) // 输出的是更新后的值 --> 1
48 }
49
50 componentDidMount() {
51     document.body.addEventListener('click', this.changeValue, false)
52 }
53
54 render() {
55     return (
56         <div>
57             {`Counter is: ${this.state.val}`}
58         </div>
59     )
60 }
61 }
62
63 4. setTimeout
64 class App extends Component {
65
66     state = { val: 0 }
67
68     componentDidMount() {
69         setTimeout(_ => {
70             this.setState({ val: this.state.val + 1 })
71             console.log(this.state.val) // 输出更新后的值 --> 1
72         }, 0)
73     }
74
75     render() {
76         return (
77             <div>
78                 {`Counter is: ${this.state.val}`}
79             </div>
80         )
81     }
82 }
83
84 5. 批处理
85 class App extends Component {
86
87     state = { val: 0 }
88
89     batchUpdates = () => {
90         this.setState({ val: this.state.val + 1 })
91         this.setState({ val: this.state.val + 1 })
92         this.setState({ val: this.state.val + 1 })
93     }

```

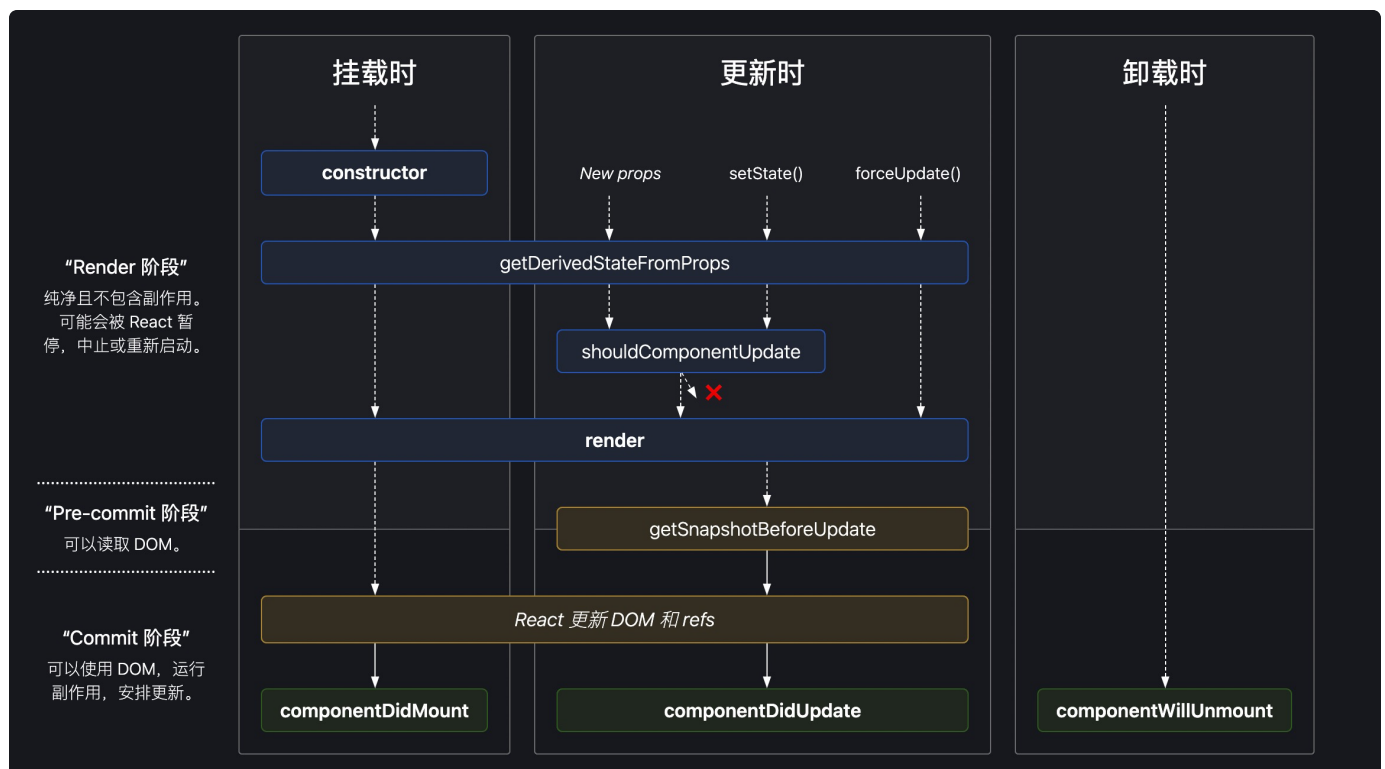
```

94
95   render() {
96     return (
97       <div onClick={this.batchUpdates}>
98         `Counter is ${this.state.val}` // 1
99       </div>
100     )
101   }
102 }

```

1. `setState` 只在合成事件和生命周期中是“异步”的，在原生事件和 `setTimeout` 中都是同步的；
2. `setState` 的“异步”并不是说内部由异步代码实现，其实本身执行的过程和代码都是同步的，只是合成事件和钩子函数的调用顺序在更新之前，导致在合成事件和钩子函数中没法立马拿到更新后的值，形式了所谓的“异步”，当然可以通过第二个参数 `setState(partialState, callback)` 中的 `callback` 拿到更新后的结果。
3. `setState` 的批量更新优化也是建立在“异步”（合成事件、钩子函数）之上的，在原生事件和 `setTimeout` 中不会批量更新，在“异步”中如果对同一个值进行多次 `setState`，`setState` 的批量更新策略会对其进行覆盖，取最后一次的执行，如果是同时 `setState` 多个不同的值，在更新时会对其进行合并批量更新。

3.4. 生命周期



3.4.1. render

是class组件必需的方法

获取最新的 props 和 state

在不修改组件 state 的情况下，每次调用时都返回相同的结果

3.4.2. constructor

如果不初始化 state 或不进行方法绑定，则不需要为 React 组件实现构造函数。

- 通过给 this.state 赋值对象来初始化内部 state。
- 为事件处理函数绑定实例

JSX | 复制代码

```
1  constructor(props) {  
2    super(props);  
3    // 不要在这里调用 this.setState()  
4    this.state = { counter: 0 };  
5    this.handleClick = this.handleClick.bind(this);  
6  }  
7  
8  1. 不要调用 setState()  
9  2. 避免将 props 的值复制给 state  
10 this.state = { color: props.color }; // wrong
```

3.4.3. componentDidMount

会在组件挂载后（插入 DOM 树中）立即调用

依赖于 DOM 节点的初始化应该放在这里，如需通过网络请求获取数据；

可以在此生命周期里加 setState，但发生在浏览器更新屏幕之前，会导致性能问题；

有更新在render阶段的 constructor 中 init State，但有更新可以在此方法时 setState

3.4.4. componentDidUpdate

JSX | 复制代码

```
1  componentDidUpdate(prevProps, prevState, snapshot)
```

会在更新后会被立即调用。首次渲染不会执行此方法。

```

1  componentDidUpdate(prevProps) {
2    // 典型用法（不要忘记比较 props）：加条件判断，不然死循环
3    if (this.props.userID !== prevProps.userID) {
4      this.fetchData(this.props.userID);
5    }
6  }
7  如果组件实现了 getSnapshotBeforeUpdate() 生命周期，
8  则它的返回值将作为 componentDidUpdate() 的第三个参数 “snapshot” 参数传递。否则此
   参数将为 undefined。

```

如果 shouldComponentUpdate() 返回值为 false，则不会调用 componentDidUpdate()。

3.4.5. componentWillUnmount

componentWillUnmount() 会在组件卸载及销毁之前直接调用。例如，清除 timer，取消网络请求；componentWillUnmount() 中不应调用 setState()，因为该组件将永远不会重新渲染；

3.4.6. shouldComponentUpdate

(不常用)

```

1  shouldComponentUpdate(nextProps, nextState)

```

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。默认行为是 state 每次发生变化组件都会重新渲染。

作为性能优化使用，返回 false 可以跳过 re-render

shouldComponentUpdate() 返回 false，不会调用 UNSAFE_componentWillUpdate(), render() 和 componentDidUpdate()。

3.4.7. getDerivedStateFromProps

(不常用)

是为了取代 componentWillReceiveProps 和 componentWillUpdate 设置的

根据 props 的变化改变 state，它应返回一个对象来更新 state，如果返回 null 则不更新任何内容。

- 在使用此生命周期时，要注意把传入的 prop 值和之前传入的 prop 进行比较；
- 因为这个生命周期是静态方法，同时要保持它是纯函数，不要产生副作用；

```

1  static getDerivedStateFromProps(nextProps, prevState) {
2      const {type} = nextProps;
3      // 当传入的type发生变化的时候, 更新state
4      if (type !== prevState.type) {
5          return {
6              type,
7          };
8      }
9      // 否则, 对于state不进行任何操作
10     return null;
11 }
12
13 Class ColorPicker extends React.Component {
14     state = {
15         color: '#000000'
16     }
17     static getDerivedStateFromProps (props, state) {
18         if (props.color !== state.color) {
19             return {
20                 color: props.color
21             }
22         }
23         return null
24     }
25     ... // 选择颜色方法
26     render () {
27         .... // 显示颜色和选择颜色操作, setState({color: XXX})
28     }
29 }
30
31 Class ColorPicker extends React.Component {
32     state = {
33         color: '#000000',
34         prevPropColor: '' // setState 和 forceUpdate也会触发此生命周期, 会覆盖
35     }
36     static getDerivedStateFromProps (props, state) {
37         if (props.color !== state.prevPropColor) {
38             return {
39                 color: props.color,
40                 prevPropColor: props.color
41             }
42         }
43         return null
44     }
45     ... // 选择颜色方法

```


```
46   render () {  
47     .... // 显示颜色和选择颜色操作  
48   }  
49 }
```

3.4.8. `getSnapshotBeforeUpdate`

(不常用)



JSX

 复制代码

```
1  getSnapshotBeforeUpdate(prevProps, prevState)
```

`getSnapshotBeforeUpdate()` 在最近一次渲染输出（提交到 DOM 节点）之前调用；此生命周期方法的任何返回值将作为参数传递给 `componentDidUpdate()`。

```
1 class ScrollingList extends React.Component {
2   constructor(props) {
3     super(props);
4     this.listRef = React.createRef();
5   }
6
7   getSnapshotBeforeUpdate(prevProps, prevState) {
8     // 我们是否在 list 中添加新的 items ?
9     // 捕获滚动位置以便我们稍后调整滚动位置。
10    if (prevProps.list.length < this.props.list.length) {
11      const list = this.listRef.current;
12      return list.scrollHeight - list.scrollTop;
13    }
14    return null;
15  }
16
17  componentDidUpdate(prevProps, prevState, snapshot) {
18    // 如果我们 snapshot 有值, 说明我们刚刚添加了新的 items,
19    // 调整滚动位置使得这些新 items 不会将旧的 items 推出视图。
20    // (这里的 snapshot 是 getSnapshotBeforeUpdate 的返回值)
21    if (snapshot !== null) {
22      const list = this.listRef.current;
23      list.scrollTop = list.scrollHeight - snapshot;
24    }
25  }
26
27  render() {
28    return (
29      <div ref={this.listRef}>{/* ...contents... */}</div>
30    );
31  }
32 }
```

3.4.9. static getDerivedStateFromError

(不常用)

配合Error boundaries使用

此生命周期会在后代组件抛出错误后被调用。它将抛出的错误作为参数，并返回一个值以更新 state；

3.4.10. componentDidCatch

(不常用)

`componentDidCatch()` 会在“提交”阶段被调用，因此允许执行副作用。它应该用于记录错误之类的情况；

JSX | 复制代码

```
1  componentDidCatch(error, info)
2
3  class ErrorBoundary extends React.Component {
4    constructor(props) {
5      super(props);
6      this.state = { hasError: false };
7    }
8
9    static getDerivedStateFromError(error) {
10     // 更新 state 使下一次渲染可以显示降级 UI
11     return { hasError: true };
12   }
13
14   componentDidCatch(error, info) {
15     // "组件堆栈" 例子:
16     //   in ComponentThatThrows (created by App)
17     //   in ErrorBoundary (created by App)
18     //   in div (created by App)
19     //   in App
20     logComponentStackToMyService(info.componentStack);
21   }
22
23   render() {
24     if (this.state.hasError) {
25       // 你可以渲染任何自定义的降级 UI
26       return <h1>Something went wrong.</h1>;
27     }
28
29     return this.props.children;
30   }
31 }
```

3.4.11. UNSAFE_componentWillMount

(不建议使用)

`UNSAFE_componentWillMount()` 在挂载之前被调用；

它在 `render()` 之前调用，因此在此方法中同步调用 `setState()` 不会生效；

需要的话用 `componentDidMount` 替代。

3.4.12. UNSAFE_componentWillReceiveProps

(不建议使用)

UNSAFE_componentWillReceiveProps() 会在已挂载的组件接收新的 props 之前被调用；

如果你需要更新状态以响应 prop 更改（例如，重置它），你可以比较 this.props 和 nextProps 并在此方法中使用 this.setState() 执行 state 转换。

3.4.13. UNSAFE_componentWillUpdate

(不建议使用)

- 当组件收到新的 props 或 state 时，会在渲染之前调用 UNSAFE_componentWillUpdate()；
- 使用此作为在更新发生之前执行准备更新的机会；
- 初始渲染不会调用此方法；

如果 shouldComponentUpdate() 返回 false，则不会调用 UNSAFE_componentWillUpdate()；

3.5. 事件处理

3.5.1. 语法格式

1. 在JSX元素上添加事件,通过on*EventType这种内联方式添加,命名采用小驼峰式(camelCase)的形式,而不是纯小写(原生HTML中对DOM元素绑定事件,事件类型是小写的)；
2. 无需调用addEventListener进行事件监听，也无需考虑兼容性，React已经封装好了一些的事件类型属性；
3. 使用 JSX 语法时你需要传入一个函数作为事件处理函数，而不是一个字符串；
4. 不能通过返回 false 的方式阻止默认行为。你必须显式的使用 preventDefault；

```
1 // DOM
2 <button onClick="activateLasers()">
3   Activate Lasers
4 </button>
5
6 // React
7 <button onClick={activateLasers}>
8   Activate Lasers
9 </button>
10
11 // JS
12 <form onSubmit="console.log('You clicked submit.');" return false">
13   <button type="submit">Submit</button>
14 </form>
15
16 // React
17 一般不需要使用 addEventListener 为已创建的 DOM 元素添加监听器;
18 function Form() {
19   function handleSubmit(e) {
20     e.preventDefault();
21     console.log('You clicked submit.');
```

```

1 class Toggle extends React.Component {
2   constructor(props) {
3     super(props);
4     this.state = {isToggleOn: true};
5
6     // 为了在回调中使用 `this`，这个绑定是不可避免的
7     this.handleClick = this.handleClick.bind(this);
8   }
9
10  handleClick() {
11    this.setState(prevState => ({
12      isToggleOn: !prevState.isToggleOn
13    }));
14  }
15
16  render() {
17    return (
18      // class 的方法默认不会绑定 this。如果没有绑定 this.handleClick 并把它传入
      // 了 onClick，
19      // this 的值为 undefined。
20      <button onClick={this.handleClick}>
21        {this.state.isToggleOn ? 'ON' : 'OFF'}
22      </button>
23    );
24  }
25 }
26
27 ReactDOM.render(
28   <Toggle />,
29   document.getElementById('root')
30 );
31
32 // 为什么要绑定this
33 function createElement(dom, params) {
34   var domObj = document.createElement(dom);
35   domObj.onclick = params.onclick;
36   domObj.innerHTML = params.conent;
37   return domObj
38 }
39 // createElement 的onClick函数是绑定到domObj上的，如果this不显式绑定，不会绑定到
// Toggle上
40
41 // 不显式使用bind
42 1. public class fields 语法
43 class LoggingButton extends React.Component {

```

```

44 // 此语法确保 `handleClick` 内的 `this` 已被绑定。
45 // 注意：这是 *实验性* 语法。
46 ▼ handleClick = () => {
47     console.log('this is:', this);
48 }
49
50 ▼ render() {
51     return (
52         <button onClick={this.handleClick}>
53             Click me
54         </button>
55     );
56 }
57 }
58
59 2. 箭头函数，问题： 每次render都会创建不同的回调函数，如果该回调函数作为props传入子组件，每次子组件都要re-render
60 ▼ class LoggingButton extends React.Component {
61     handleClick() {
62         console.log('this is:', this);
63     }
64
65     render() {
66         // 此语法确保 `handleClick` 内的 `this` 已被绑定。
67         return (
68             <button onClick={() => this.handleClick()}>
69             // <button onClick={this.handleClick().bind(this)}>
70                 Click me
71             </button>
72         );
73     }
74 }
75
76 3. createReactClass代替

```

3.5.2. 接收参数

1. 事件对象 e 会被作为第二个参数传递；
2. 通过箭头函数的方式，事件对象必须显式的进行传递；
3. 通过 Function.prototype.bind 的方式，事件对象以及更多的参数将会被隐式的进行传递；

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

3.6. 条件渲染

3.6.1. if else 渲染

```
1 class LoginControl extends React.Component {
2   constructor(props) {
3     super(props);
4     this.handleClick = this.handleClick.bind(this);
5     this.handleLogoutClick = this.handleLogoutClick.bind(this);
6     this.state = {isLoggedIn: false};
7   }
8
9   handleClick() {
10    this.setState({isLoggedIn: true});
11  }
12
13  handleLogoutClick() {
14    this.setState({isLoggedIn: false});
15  }
16
17  render() {
18    const isLoggedIn = this.state.isLoggedIn;
19    let button;
20    if (isLoggedIn) {
21      button = <LogoutButton onClick={this.handleLogoutClick} />;
22    } else {
23      button = <LoginButton onClick={this.handleClick} />;
24    }
25
26    return (
27      <div>
28        <Greeting isLoggedIn={isLoggedIn} />
29        {button}
30      </div>
31    );
32  }
33 }
34
35 ReactDOM.render(
36   <LoginControl />,
37   document.getElementById('root')
38 );
```

3.6.2. 与运算符 &&

```
1  function Mailbox(props) {
2    const unreadMessages = props.unreadMessages;
3    return (
4      <div>
5        <h1>Hello!</h1>
6        {unreadMessages.length > 0 &&
7          <h2>
8            You have {unreadMessages.length} unread messages.
9          </h2>
10       }
11     </div>
12   );
13 }
14
15 const messages = ['React', 'Re: React', 'Re:Re: React'];
16 ReactDOM.render(
17   <Mailbox unreadMessages={messages} />,
18   document.getElementById('root')
19 );
20
21 // 返回false的表达式，会跳过元素，但会返回该表达式
22 render() {
23   const count = 0;
24   return (
25     <div>
26       { count && <h1>Messages: {count}</h1>}
27     </div>
28   );
29 }
```

3.6.3. 三元运算符


```
1  render() {  
2    const isLoggedIn = this.state.isLoggedIn;  
3    return (  
4      <div>  
5        {isLoggedIn  
6          ? <LogoutButton onClick={this.handleLogoutClick} />  
7          : <LoginButton onClick={this.handleLoginClick} />  
8        }  
9      </div>  
10   );  
11 }
```

3.6.4. 如何阻止组件渲染

```
1  function WarningBanner(props) {
2    if (!props.warn) {
3      return null;
4    }
5
6    return (
7      <div className="warning">
8        Warning!
9      </div>
10   );
11 }
12
13 class Page extends React.Component {
14   constructor(props) {
15     super(props);
16     this.state = {showWarning: true};
17     this.handleClick = this.handleClick.bind(this);
18   }
19
20   handleClick() {
21     this.setState(state => ({
22       showWarning: !state.showWarning
23     }));
24   }
25
26   render() {
27     return (
28       <div>
29         <WarningBanner warn={this.state.showWarning} />
30         <button onClick={this.handleClick}>
31           {this.state.showWarning ? 'Hide' : 'Show'}
32         </button>
33       </div>
34     );
35   }
36 }
37
38 ReactDOM.render(
39   <Page />,
40   document.getElementById('root')
41 );
```

3.7. 列表

```
1  function NumberList(props) {  
2    const numbers = props.numbers;  
3    const listItems = numbers.map((number) =>  
4      <li key={number.toString()}>  
5        {number}  
6      </li>  
7    );  
8  
9    return (  
10     <ul>{listItems}</ul>  
11   );  
12 }  
13  
14 const numbers = [1, 2, 3, 4, 5];  
15 ReactDOM.render(  
16   <NumberList numbers={numbers} />,  
17   document.getElementById('root')  
18 );  
19 // 若没有key, 会warning a key should be provided for list items  
20 // key可以帮助react diff, 最好不用index作为key, 会导致性能变差;  
21 // 如果不指定显式的 key 值, 默认使用索引作为列表项目的 key 值;
```

3.7.1. key注意点

```
1 key要保留在map的遍历元素上
2
3 // demo1
4 ▾ function ListItem(props) {
5     // 正确! 这里不需要指定 key:
6     return <li>{props.value}</li>;
7 }
8
9 ▾ function NumberList(props) {
10     const numbers = props.numbers;
11     const listItems = numbers.map((number) =>
12         // 正确! key 应该在数组的上下文中被指定
13         <ListItem key={number.toString()} value={number} />
14     );
15     return (
16         <ul>
17             {listItems}
18         </ul>
19     );
20 }
21
22 const numbers = [1, 2, 3, 4, 5];
23 ReactDOM.render(
24     <NumberList numbers={numbers} />,
25     document.getElementById('root')
26 );
27
28 // demo2
29 ▾ function Blog(props) {
30     const sidebar = (
31         <ul>
32             {props.posts.map((post) =>
33                 <li key={post.id}>
34                     {post.title}
35                 </li>
36             )}
37         </ul>
38     );
39     const content = props.posts.map((post) =>
40         <div key={post.id}>
41 ▾         <h3>{post.title}</h3>
42 ▾         <p>{post.content}</p>
43         </div>
44     );
45     return (
```

```

46     <div>
47       {sidebar}
48     </div>
49     {content}
50   </div>
51 );
52 }
53
54 ▼ const posts = [
55   {id: 1, title: 'Hello World', content: 'Welcome to learning React!'},
56   {id: 2, title: 'Installation', content: 'You can install React from
    npm.'}
57 ];
58 ReactDOM.render(
59   <Blog posts={posts} />,
60   document.getElementById('root')
61 );
62
63 // demo3
64 ▼ function NumberList(props) {
65   const numbers = props.numbers;
66   return (
67     <ul>
68       {numbers.map((number) =>
69         <ListItem key={number.toString()}
70           value={number} />
71       )}
72     </ul>
73   );
74 }

```