

# react入门笔记

## react入门笔记

1. 创建react组件
  - 1.1. 无状态函数式组件
  - 1.2. React.Component(推荐使用)
2. react生命周期
  - 2.1. react生命周期过程
  - 2.2. setState需要注意的点
3. 事件系统
  - 3.1. 通过bind在构造函数中改变this指向
  - 3.2. 通过箭头函数进行this指向改变
4. 容器类组件
5. 组件之间消息传递(重点)
  - 5.1. 父子组件传递信息
  - 5.2. 兄弟组件之间传递信息
  - 5.3. 父组件和所有后代组件之间传递信息

## 1. 创建react组件

### 1.1. 无状态函数式组件

组件的表现形式为：带有一个 `render()` 方法，通过ES6 `arrow function` 创建或者通过普通函数创建

```
1 function HelloComponent(props) {  
2   return <div> Hello {props.name}</div>  
3 }  
4 const HelloComponent = props => <div> hello {props.name} </div>
```

```
function HelloComponent(props) {  
  return <div> Hello {props.name}</div>  
}  
  
const HelloComponent = props => <div> hello {props.name} </div>
```

这里不是this.props.name,无状态函数式组件无法访问this对象

特点：

- 可读性好，减少代码冗余，精简至一个 `render()` 方法
- 组件只能访问输入的 `props`，无法访问组件中的 `this` 对象（没有实例化过程）
- 组件无法访问生命周期的方法

官方指出：在大部分 `react` 组件中，大部分组件被写成无状态组件，通过简单的组合可以构成其它组件，这种通过简单组件然后合并成一个大应用的设计模式被提倡

## 1.2. React.Component(推荐使用)

es6的继承

```
1  // 使用class定义构造函数
2  class Person {
3      // 定义对象的属性
4      // 传入成员变量，可以给属性初始值或默认值
5      constructor (name,age=25) {
6          this.name = name;
7          this.age = age;
8      };
9      // 定义对象的方法
10     showName () {
11         return this.name;
12     };
13     showAge () {
14         return this.age;
15     }
16 }
17
18 const per = new Person('hello',28);
19 // console.log(per.name,per.age); // hello, 28
20
21 // 子类继承父类
22 class Worker extends Person {
23     constructor(name,age,job="砍柴"){
24         // 把原来构造函数的参数传入
25         // 必须调用super方法，相当于call/apply改变子类this的指向
26         // 否则子类得不到this对象
27         super(name,age);
28         this.job = job;
29     };
26 }
```

```

30     showJob () {
31         return this.job;
32     }
33 }
34
35 const wor = new Worker('ff',75);
36 // console.log(wor.showJob()); //砍柴

```

利用ES6的继承书写 **React** 组件，配置propTypes和defaultProps是作为组件的类的静态属性

```

1  class Contacts extends React.Component {
2      constructor(props) {
3          super(props)
4          // 设置初始化state
5          this.state = {
6              age: 18,
7          }
8      }
9      static propTypes = { // 类的静态属性
10         name: React.PropTypes.string
11     }
12     static defaultProps = { // 类的静态属性
13         name: '',
14     }
15 }

```

总结：

1. 只要有可能，尽量使用无状态组件形式
2. 要用到state，生命周期方法等，使用 **React.Component** 这种形式创建组件

## 2. **react** 生命周期

### 2.1. react生命周期过程

- 组件挂载阶段：
  - constructor
  - componentWillMount
  - render
  - componentDidMount
- 组件更新阶段：

componentWillReceiveProps: 在挂载后的组件接收到新属性之前调用

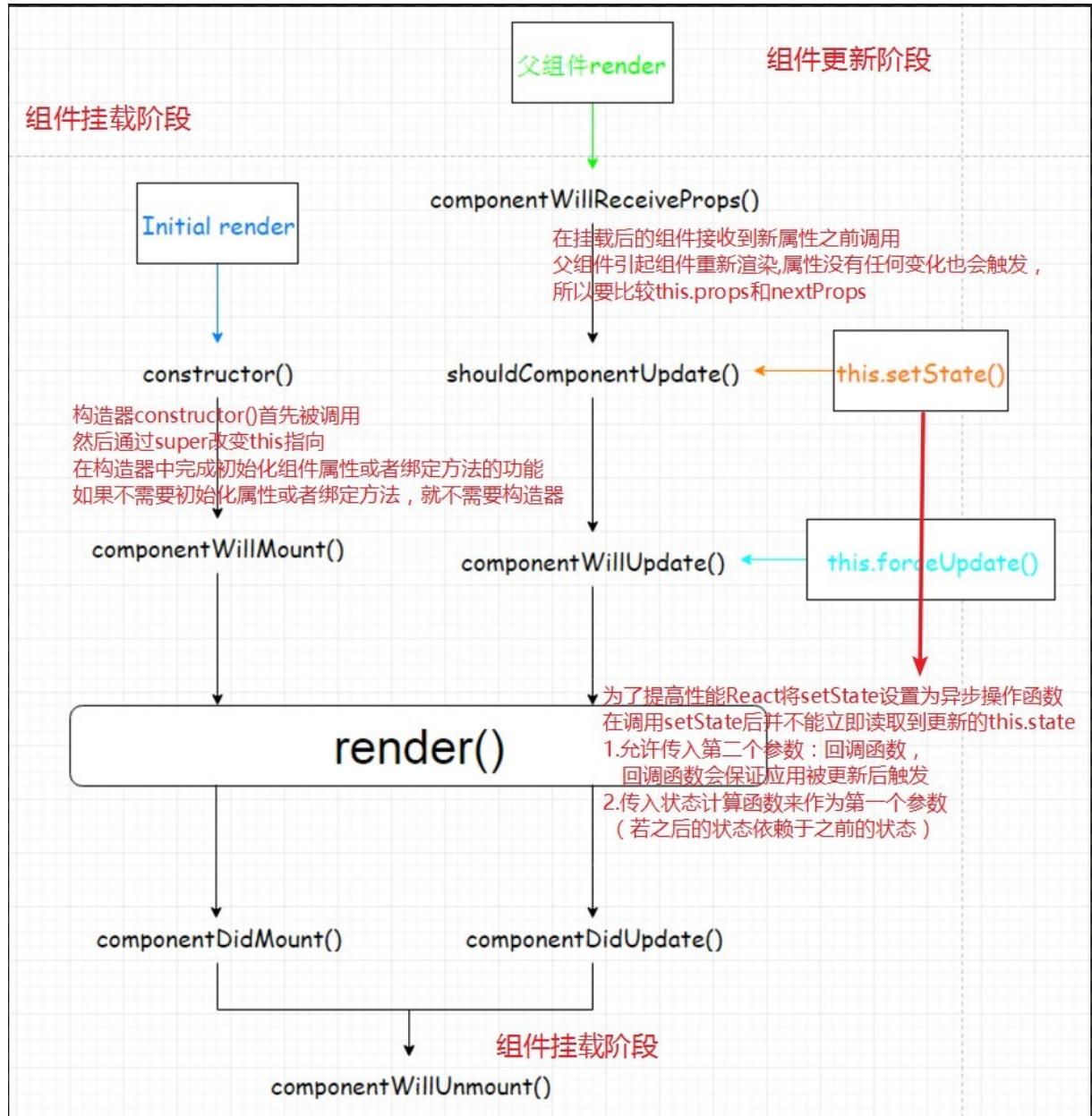
shouldComponentUpdate

render

componentDidUpdate

- 组件卸载阶段

componentWillUnmount



## 2.2. setState需要注意的点

初始的 state

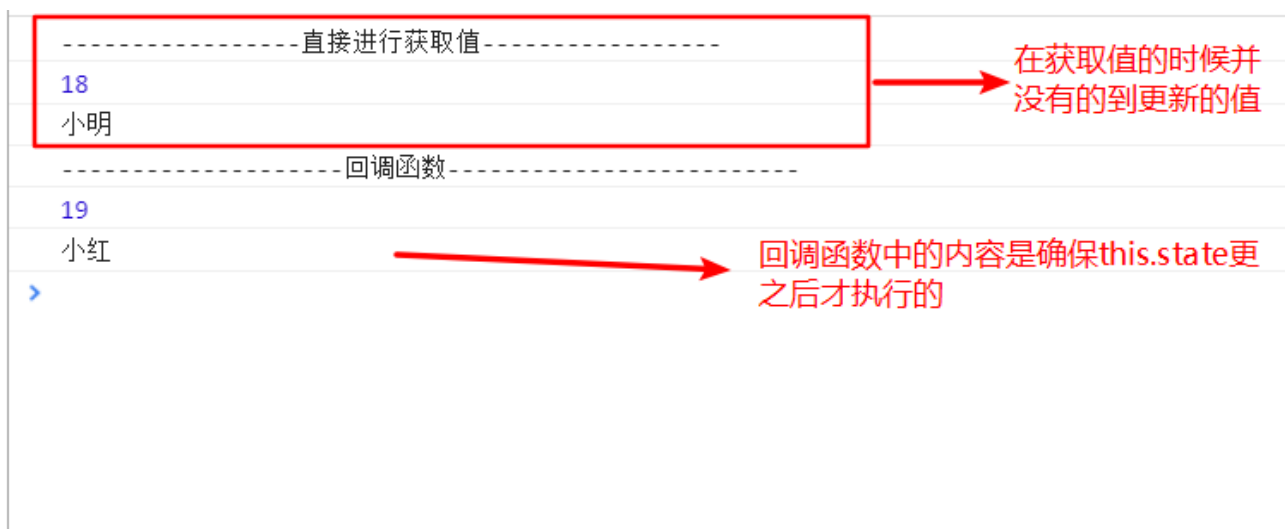
```
1  this.state = {  
2    age: 18,  
3    name: '小明',  
4  }
```

为了提高性能 `react` 将 `setState` 设置为异步操作函数，在调用 `setState` 后并不能立即读取到更新的 `this.state`

解决办法：

- 允许传入第二个参数：回调函数，回调函数会保证应用更新后触发

```
1 handleClick() {
2   this.setState(
3     {
4       age: this.state.age+1,
5       name: '小红',
6     },
7     () => {
8       console.log('-----回调函数-----
-----');
9       console.log(this.state.age);
10      console.log(this.state.name);
11    }
12  )
13  // 先执行后边的代码
14  console.log('-----直接进行获取值-----');
15  console.log(this.state.age);
16  console.log(this.state.name);
17 }
```



- 传入状态计算函数来作为第一个参数（若之后的状态依赖于之前的状态）

```
1 handleClick() {
2   // 相当于object.assign({}, {age:19}, {age:19}):会将相同的属性进行合并，并
   // 不会每次点击+2
3   this.setState({age: this.state.age+1});
```

```
4   this.setState({age: this.state.age+1});
5 }
```

并不是我们想要的效果

21



官方推荐写法

```
1 // 官方推荐写法: (保证数据的同步性, 之后的状态要依赖于之前的状态)
2 // 这样更改之后会立即生效
3 this.setState((prevState, props) => {
4   // prevState: 之前的this.state
5   // props: 父组件传来的属性组成的对象
6   // 必须要将修改的内容return, 否则不会生效
7   return {age: prevState.age+1}
8 })
9 this.setState((prevState, props) => {
10   return {age: prevState.age+1}
11 })
```



### 3. 事件系统

绑定事件时函数的调用方法并不是以对象的方法来调用的，如果不进行this指向更改的话，`this` 指向 `null` 或者 `undefined`

#### 3.1. 通过 `bind` 在构造函数中改变this指向

```
1 constructor() {
```

```

2     super()
3     // 通过bind在构造函数中改变this的指向
4     this.handleClick = this.handleClick.bind(this);
5     this.handleParams = this.handleParams.bind(this, 5);
6 }

```

通过 `bind` 方式绑定事件对象以及更多的参数被隐式的传递

```

1 handleClick() {
2     // 通过bind方式绑定事件对象以及更多的参数被隐式的传递
3     // 比如event，并不需要进行参数的传递，就可以直接在函数中使用
4     console.log(event);
5 }

```

```

▼ Event {isTrusted: false, type: "react-
  click", target: react, currentTarget: react, eventPhase: 2, ...} ⓘ
  bubbles: false
  cancelBubble: false
  cancelable: false
  composed: false
  currentTarget: null
  defaultPrevented: false
  eventPhase: 0
  isTrusted: false
  ▶ path: [react]
  returnValue: true
  srcElement: null
  target: null
  timeStamp: 13156.79999999702
  type: "react-click"
  ▶ __proto__: Event

```

如果要传入形参以及事件对象，事件对象必须要在最后传入

```

1 handleParams(id, e) {
2     // e这里必须要在id的后面传入
3     console.log(e);
4     // 我们传入的e.target为事件对象的一些属性和方法
5 }

```

```

▼ Proxy {dispatchConfig: {...}, _targetInst: FiberNode, isDefaultPrevented: f, isPropagationStopped: f, _dispatchListeners: f, ...} ⓘ
  ▶ [[Handler]]: Object
  ▶ [[Target]]: SyntheticMouseEvent
  [[IsRevoked]]: false

```

```
[[Target]]: SyntheticMouseEvent
  altKey: (...)
  bubbles: (...)
  button: (...)
  buttons: (...)
  cancelable: (...)
  clientX: (...)
  clientY: (...)
  ctrlKey: (...)
  currentTarget: (...)
  defaultPrevented: (...)
  detail: (...)
  dispatchConfig: null
  eventPhase: (...)
  getModifierState: (...)
  isDefaultPrevented: null
  isPropagationStopped: null
  isTrusted: (...)
  metaKey: (...)
  nativeEvent: (...)
  pageX: (...)
  pageY: (...)
  relatedTarget: (...)
  screenX: (...)
  screenY: (...)
  shiftKey: (...)
  target: (...)
  timeStamp: (...)
  type: (...)
  view: (...)
  _dispatchInstances: null
  _dispatchListeners: null
  _targetInst: null
  preventDefault: (...)
```

`e.target`

也可以在行内通过 `bind` 方法更改 `this` 指向（不推荐）

```
1 <h1 onClick={this.handleClick.bind(this)}>Dog</h1>
2 {/* 行内绑定的缺点:
3  每次调用的时候都会通过bind来进行this指向改变
4  在构造函数中绑定只需要一次
5  */}
```

### 3.2. 通过箭头函数进行 `this` 指向改变

通过箭头函数进行绑定事件，事件参数必须进行显式的传递

```
1 {/* 传入事件对象 */}
2 <h2 onClick={(e) => this.handleClickExtra(e)}>Cat</h2>
```

## 4. 容器类组件



```
render(
  <BlackBorderContainer>
    <div className="name">MyName: Lucy</div>
    <p className="age">
      MyAge: <span>12</span>
    </p>
  </BlackBorderContainer>,
  document.getElementById('box')
)
```

组件中嵌套的内容通过 `this.props.children` 进行在组件内展示

也可以通过类似 `this.props.children[0]` 将某个单独的组件内容选出来进行展示

循环渲染组件嵌套内容

```
1 class BlackBorderContainer extends Component {
2   constructor(props) {
3     super(props)
4   }
5   render() {
6     return (
7       <div>
8         <h2>BlackBorderContainer</h2>
9         {
10           this.props.children.map(
11             // 这里要将显示的内容return
12             // 为每一个嵌套内容添加边框
13             (children,index) => <div key={index}
14               className="border">{children}</div>
15           )
16         }
17       </div>
18     )
19   }
20 }
```

## 5. 组件之间消息传递(重点)

### 5.1. 父子组件传递信息

父->子：通过为子组件添加行内属性，子组件通过 `props` 进行接收

子->父：父组件通过 `callback` 作为 `props` 的属性传递到子组件，子组件调用 `callback`

定义2个子组件

```
1 // 在不使用state和生命周期的时候最好使用无状态函数式组件
2 const Child = props => <div><input onChange={props.change} /></div> ;
```

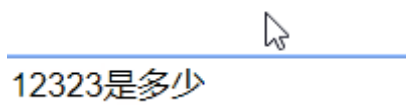
```
3  const Child1 = props => <div><h1>{props.value}</h1></div>;
```

定义父组件，将子组件的`input`中输入的值，通过子组件1展示出来

```
1  class Parent extends React.Component {
2      constructor() {
3          super()
4          this.state = {
5              value: ''
6          }
7          this.handleChange = this.handleChange.bind(this);
8      }
9      handleChange(e) {
10         // 这里的e.target.value是通过子组件传来的
11         // 这里的e.target.value是通过子组件传来的
12         this.setState({value: e.target.value});
13     }
14     render() {
15         return (
16             <div>
17                 <Child1 value={this.state.value} />
18                 <Child change={this.handleChange} />
19             </div>
20         );
21     }
22 }
```

实现效果

# 12323是多少



## 5.2. 兄弟组件之间传递信息

- 将需要传递的数据挂载到公共的父组件中，通过 `props` 传递给2个子组件，如果某个组件需要改变数据并通知其兄弟组件，则通过父组件传递 `callback` 给子组件来实现
- 利用观察者模式（发布-订阅模式实现组件之间消息的传递），利

用 `eventProxy` 模块实现消息的传递，类似于 `vue` 的 `$emit` 和 `$on`

```
1  const ep = new EventProxy();
2  class Child1 extends React.Component {
3    constructor() {
4      super();
5      this.handleChange = this.handleChange.bind(this);
6    }
7    handleChange(e) {
8      // 通知Child2组件，input的输入内容发生变化，并将输入内容传给Child2组
      件
9      ep.emit('inpVal', e.target.value)
10   }
11   render() {
12     return (
13       <div>
14         <input onChange={this.handleChange}/>
15       </div>
16     )
17   }
18 }
19 class Child2 extends React.Component {
20   constructor() {
21     super();
22     this.state = {
23       msg: '',
24     };
25   }
26   componentWillMount() {
27     // 收到Child1的通知，通过函数接收Child1传来的数据（data）
28     ep.on('inpVal', (data) => {
29       this.setState({
30         msg: data
31       });
32     });
33   }
34   render() {
35     return (
36       <div>
37         <h1>{this.state.msg}</h1>
38       </div>
39     )
40   }
41 }
42 const Parents = () => {
```

```
43     return (
44         <div>
45             <Child1 />
46             <Child2 />
47         </div>
48     )
49 }
50 ReactDOM.render(<Parents />,document.getElementById('app'));
```

### 5.3. 父组件和所有后代组件之间传递信息

传参步骤：

- 定义父组件和后代组件（后代可以是儿子、孙子、重孙子...）
  - 引入prop-types模块
  - 定义父组件的传参函数(getChildContext),将要传递给子组件的参数以对象的形式作为返回值返回
  - 分别定义父组件和孙子组件的传参和接收参数的规则(Grandson.contextTypes和Father.getChildrenTypes)
- 引入依赖模块


```
1 import React from 'react';
2 import ReactDOM from 'react-dom';
3 // 限制传递内容的数据类型
4 import PropTypes from 'prop-types';
```

定义孙子组件

```

class Grandson extends React.Component {
  constructor() {
    super();
    this.state = {
      money: 998,
    };
    this.handleClick = this.handleClick.bind(this);
  };
  handleClick() {
    this.props.toGrandson(this.state.money);
    this.context.testFn();
  }
  render() {
    return (
      <div>
        <h1>Grandson,父亲的年龄 {this.context.age}</h1>
        <button onClick={this.handleClick}>给父亲传值</button>
      </div>
    )
  }
}


```


 this.context是父组件传来的参数组成的对象

```

// 这里的function类型为: func
// 可以去官网查看PropTypes的类型
Grandson.contextTypes = {
  age: PropTypes.number,
  testFn: PropTypes.func
}

```


 不是function

定义父组件并渲染到页面

```

1  class Father extends React.Component {
2    constructor() {
3      super();
4      this.state = {
5        money: '并没有钱',
6        age: 98
7      };
8      this.changeMoney = this.changeMoney.bind(this);
9    }
10   changeMoney(newMoney) {
11     const money = newMoney;
12     this.setState({
13       money: money
14     });
15   }
16   render() {

```

```

17         return (
18             <div>
19                 <h1>我是父组件,money:{this.state.money}</h1>
20                 <Grandson faAge={this.state.age} toGrandson=
{this.changeMoney}></Grandson>
21             </div>
22         );
23     }
24     testFn() {
25         window.alert('我被调用')
26     }
27     // 原型上定义方法，返回要传递给后代的数据
28     getChildContext() {
29         return {
30             age: 18,
31             testFn: () => {
32                 this.testFn();
33             },
34         }
35     }
36 }
37 // 限制传递数据的格式
38 Father.childContextTypes = {
39     age: PropTypes.number,
40     testFn: PropTypes.func
41 };
42 ReactDOM.render(<Father />, document.getElementById('box'));

```

实现效果，alert为调用的函数，money：998为孙子组件传来的参数

## 我是父组件,money:并没有钱

## Grandson,父亲的年龄 18

给父亲传值