



第一节

ReactJs 简介

React Js 起源

React 起源于 Facebook 的内部项目，因为该公司对市场上所有 JavaScript MVC 框架，都不满意，就决定自己写一套，用来架设Instagram 的网站。做出来以后，发现这套东西很好用，就在2013年5月开源了。

由于 React的设计思想极其独特，属于革命性创新，性能出众，代码逻辑却非常简单。所以，越来越多的人开始关注和使用，认为它可能是将来 Web 开发的主流工具。



LEARNING FACEBOOK'S
 React.js

ReactJS官网地址: <https://reactjs.org/>

Github: <https://github.com/facebook/react>

ReactJS中文文档: <https://react.docschina.org/>

- React不是一个完整的MVC框架，最多可以认为是MVC中的V（View），甚至React并不非常认可MVC开发模式；

React Js 引入

- 引入js文件

```
<script src="../react.development.js"></script>
<script src="../react-dom.development.js"></script>
<script src="../build/browser.min.js"></script>
.
```

- type=“text/babel”

```
| <script type="text/babel">
```

第二节

ReactJs 入门

Hello world

- Hello World

```
ReactDOM.render(  
  <h1>Hello, world!</h1>,  
  document.getElementById('root')  
)
```

- **JSX** 一种 **JavaScript** 的语法扩展。 我们推荐在 **React** 中使用 **JSX** 来描述用户界面。
 - 可以任意地在 **JSX** 当中使用**JavaScript** 表达式，在 **JSX** 当中的表达式要包含在大括号里
- **ReactDOM.render()** 将**React**元素渲染到**DOM**节点中

```
const element = (
  <div>
    <h1>Hello, world!</h1>
    <h2>It is . . .</h2>
  </div>
);
ReactDOM.render(element, document.getElementById('root'));
```

模块与组件

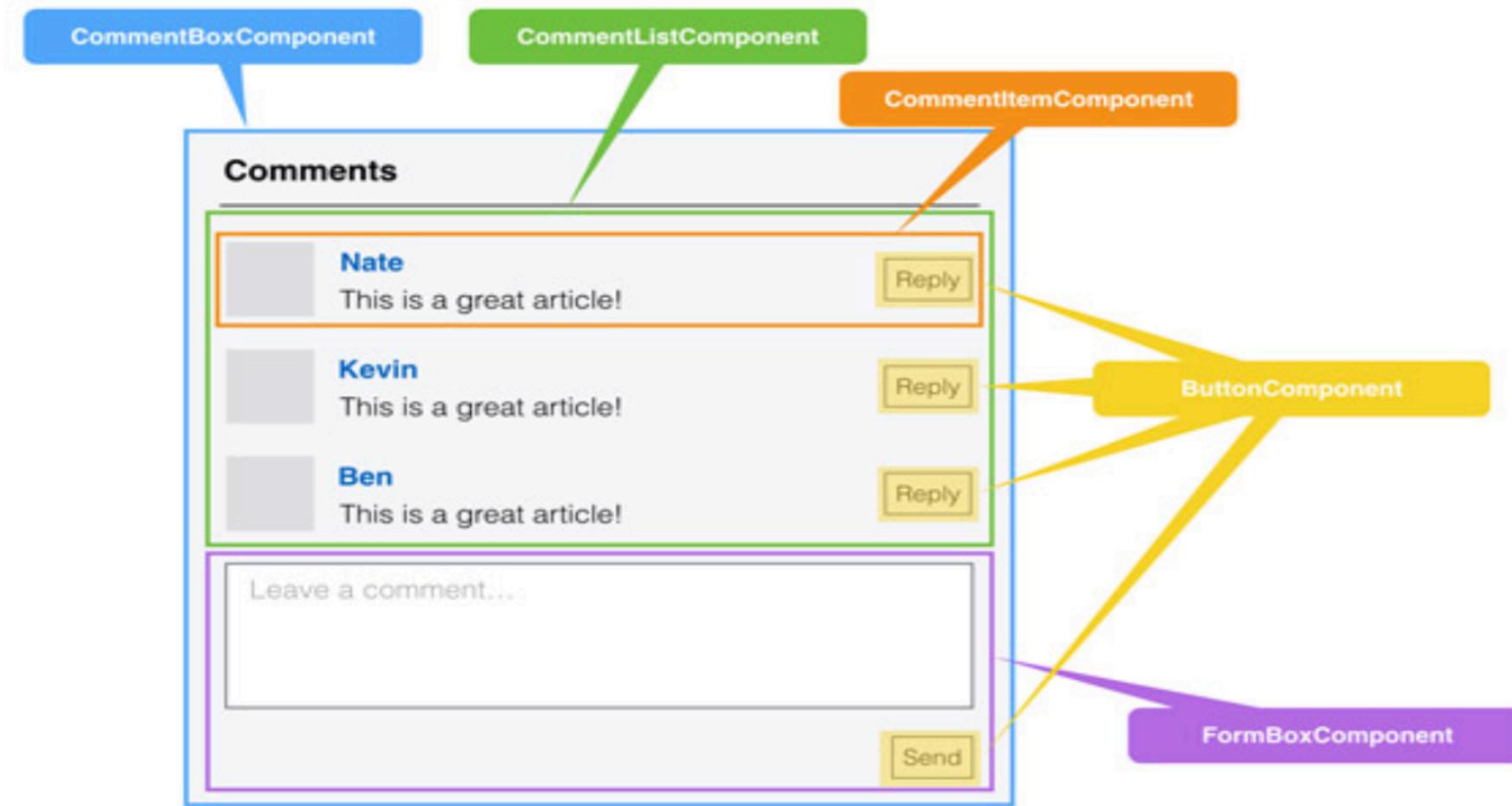
- **模块**
 - 理解: 向外提供特定功能的js程序, 一般就是一个js文件
 - 作用: 复用js, 简化js的编写, 提高js运行效率
- **组件**
 - 理解: 用来实现特定(局部)功能效果的代码集合(html/css/js)
 - 作用: 复用编码, 简化项目编码, 提高运行效率
- **模块化**
 - 当应用的js都以模块来编写的, 这个应用就是一个模块化的应用
- **组件化**
 - 当应用是以多组件的方式实现, 这个应用就是一个组件化的应用

组件化

- 所谓组件，即封装起来的具有独立功能的UI部件。React推荐以组件的方式去重新思考UI构成，将UI上每一个功能相对独立的模块定义成组件，然后将小的组件通过组合或者嵌套的方式构成大的组件，最终完成整体UI的构建。例如，Facebook的instagram.com整站都采用了React来开发，整个页面就是一个大的组件，其中包含了嵌套的大量其它组件，大家有兴趣可以看下它背后的代码。
- 如果说MVC的思想让你做到视图-数据-控制器的分离，那么组件化的思考方式则是带来了UI功能模块之间的分离。我们通过一个典型的Blog评论界面来看MVC和组件化开发思路的区别。
- 对于MVC开发模式来说，开发者将三者定义成不同的类，实现了表现，数据，控制的分离。开发者更多的是从技术的角度来对UI进行拆分，实现松耦合。
- 对于React而言，则完全是一个新的思路，开发者从功能的角度出发，将UI分成不同的组件，每个组件都独立封装。

1-5 组件化

- ▶ 在React中，你按照界面模块自然划分的方式来组织和编写你的代码，对于评论界面而言，整个UI是一个通过小组件构成的大组件，每个组件只关心自己部分的逻辑，彼此独立。



- 特征

- (1) 可组合 (**Composeable**)：一个组件易于和其它组件一起使用，或者嵌套在另一个组件内部。如果一个组件内部创建了另一个组件，那么说父组件拥有 (`own`) 它创建的子组件，通过这个特性，一个复杂的UI可以拆分成多个简单的UI组件；
- (2) 可重用 (**Reusable**)：每个组件都是具有独立功能的，它可以被使用在多个UI场景；
- (3) 可维护 (**Maintainable**)：每个小的组件仅仅包含自身的逻辑，更容易被理解和维护；

- 优点

- 提高代码复用率：组件将数据和逻辑封装，类似面向对象中类；
- 降低测试难度：组件高内聚低耦合，很容易对单个组件进行测试；
- 降低代码复杂度：直观的语法提高代码可读性。

组件定义

组件从概念上看就像是函数，它可以接收任意的输入值（称之为“**props**”），并返回一个需要在页面上展示的**React**元素

1. 定义一个组件最简单的方式是使用JavaScript函数(无状态组件):

```
const Welcome = () => {
  return <h1>Hello, 唯创 ~ </h1>;
}

const element = <Welcome/>;
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

2. ES6 class 来定义一个组件(有状态组件):

```
class Welcome extends React.Component{
  render(){
    return <h3>hello, 唯创 ~ </h3>
  }
}
ReactDOM.render(
  <Welcome />,
  document.getElementById('root')
);
```

3. 注意：

- 1) 组件名必须首字母大写
- 2) 虚拟DOM元素只能有一个根元素
- 3) 虚拟DOM元素必须有结束标签

- **state** 是组件对象最重要的属性, 值是对象(可以包含多个数据), 用于改变组件内容状态的值 (动态的)
- 组件被称为“状态机”, 通过更新组件的**state**来更新对应的页面显示(重新渲染组件)

1) 初始化状态

```
class Number extends React.Component{  
    constructor(props){  
        super(props);  
        this.state = {  
            count:1  
        };  
    }  
}
```

2) 读取某个状态值 **this.state.count**

3) 更新状态---->组件界面更新

```
//this.state.count++; 不能直接修改count值  
this.setState({  
    count:this.state.count+1  
});
```

- React-组件通信(属性传值 – props)

- 类的写法:

```
class Welcome extends React.Component{  
    render(){  
        return <h3>hello, {this.props.name}</h3>  
    }  
}  
  
ReactDOM.render(  
    <Welcome name="haha"/>,  
    document.getElementById('root')  
,
```

- 函数写法:

```
const Welcome = (props) => {  
    return <h1>Hello, {props.name} ~ </h1>;  
}
```

- **props.children** 取到组件中首尾标签中间的值{props.children}

```
ReactDOM.render(  
    <Welcome name="唯创">你好同学!</Welcome>,  
    document.getElementById('root')
```

props VS state

state 和 **props** 主要的区别

在于 **props** 是不可变的，而 **state** 可以根据与用户交互来改变。

- **props** 用于组件通信进行传值
- **state** 用于改变组件内容状态的值（动态的）

组件事件

- 组件绑定事件

- React事件绑定属性的命名采用驼峰式写法
- bind改变this指向
- 形参event可以取到事件
- ref

```
class Number extends React.Component{  
  constructor(props){  
    super(props);  
    this.state = {  
      count:1  
    };  
    add(){  
      //this.state.count++; 不能直接修改count值  
      this.setState({  
        count:this.state.count+1  
      });  
    }  
    render(props){  
      return(  
        <div>  
          <button onClick = {this.add.bind(this)}>+</button>  
          <h3>{this.state.count}</h3>  
        </div>  
      )  
    }  
}
```

- 事件传递参数
 - 箭头函数
 - **bind()**
- 子组件向父组件传值（传事件）

双向数据绑定

- **onChange()**事件
 - 修改**state**中的值为表单元素的**value**

```
class Number extends React.Component{  
  constructor(props){  
    super(props);  
    this.state = {  
      count:1  
    };  
  }  
  change(event){  
    this.setState({  
      count:event.target.value  
    })  
  }  
  render(){  
    return(  
      <div>  
        <h1>{this.state.count}</h1>  
        <input type="text" onChange = {this.change.bind(this)} />  
      </div>  
    )  
  }  
}
```

- **className**

```
return(  
  <div>  
    |   <h1 className = "a" >{this.state.count}</h1>  
  </div>  
)
```

- **style** 使用**render**中声明的样式

```
render(){  
  const style = {  
    color:'#f00'  
  }  
  return(  
    <div>  
      |   <h1 style={style} >{this.state.count}</h1>  
    </div>  
  )  
}
```

- 条件渲染
 - if
 - 三目运算符
- 循环
 - es6 map

有状态组件vs无状态组件

有状态

无状态

`class A extends React.Component`

`const A = (props) => {..... }`

✓ 可以拥有状态

✗ 不可以拥有状态

✓ 拥有生命周期

✗ 不拥有生命周期

可以通过`this`来接受状态和属性

可以通过属性实现数据传递

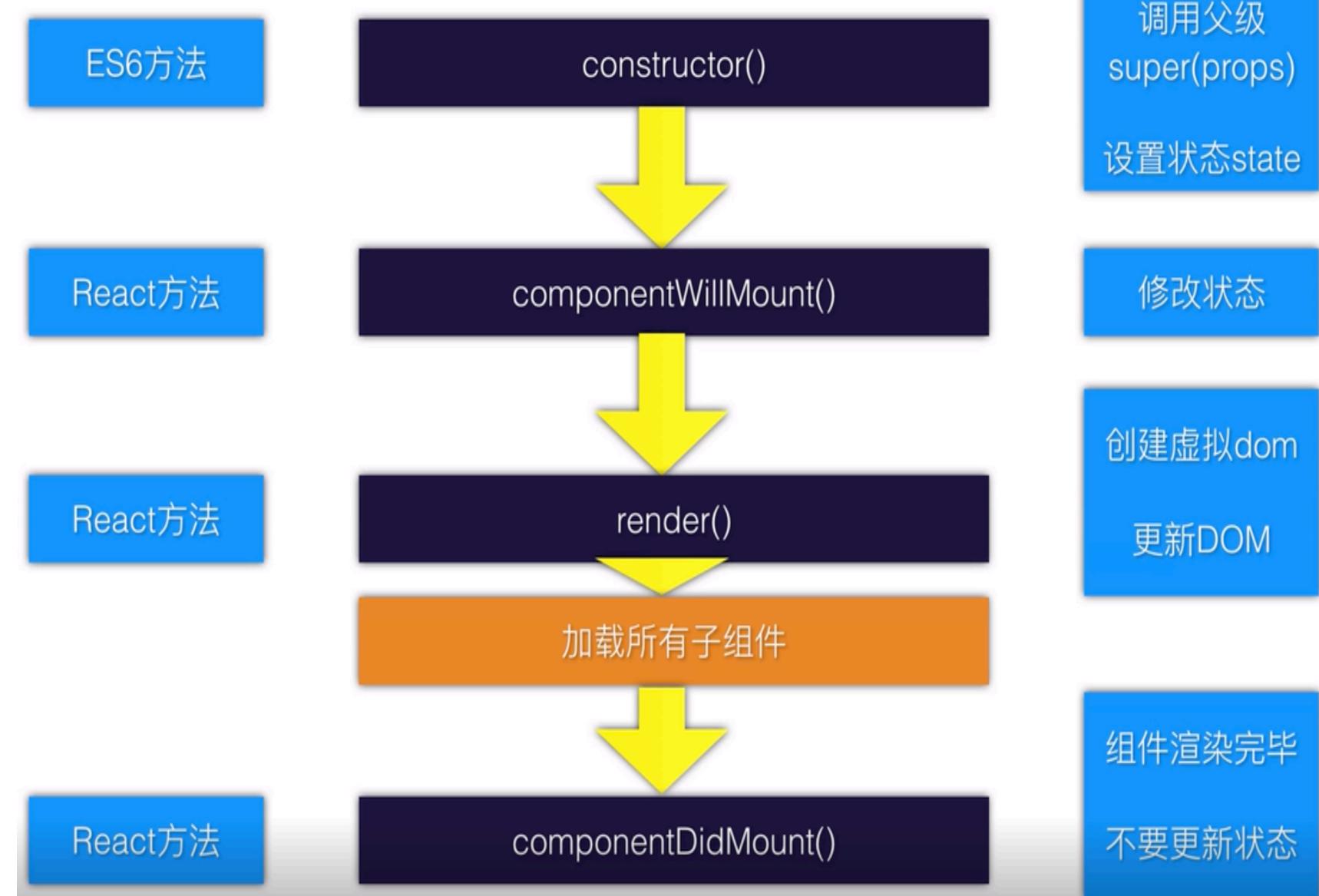
`this.state.x & this.props.x`

只有在需要管理状态,或者需要使用生命
周期时在用.

其他时间用这种

生命周期钩子函数（有状态组件）

Vue实例有一个完整的生命周期，也就是从开始创建、初始化数据、编译模板、挂载Dom、渲染→更新→渲染、卸载等一系列过程，我们称这是**Vue**的生命周期。通俗说就是**Vue**实例从创建到销毁的过程，就是生命周期。



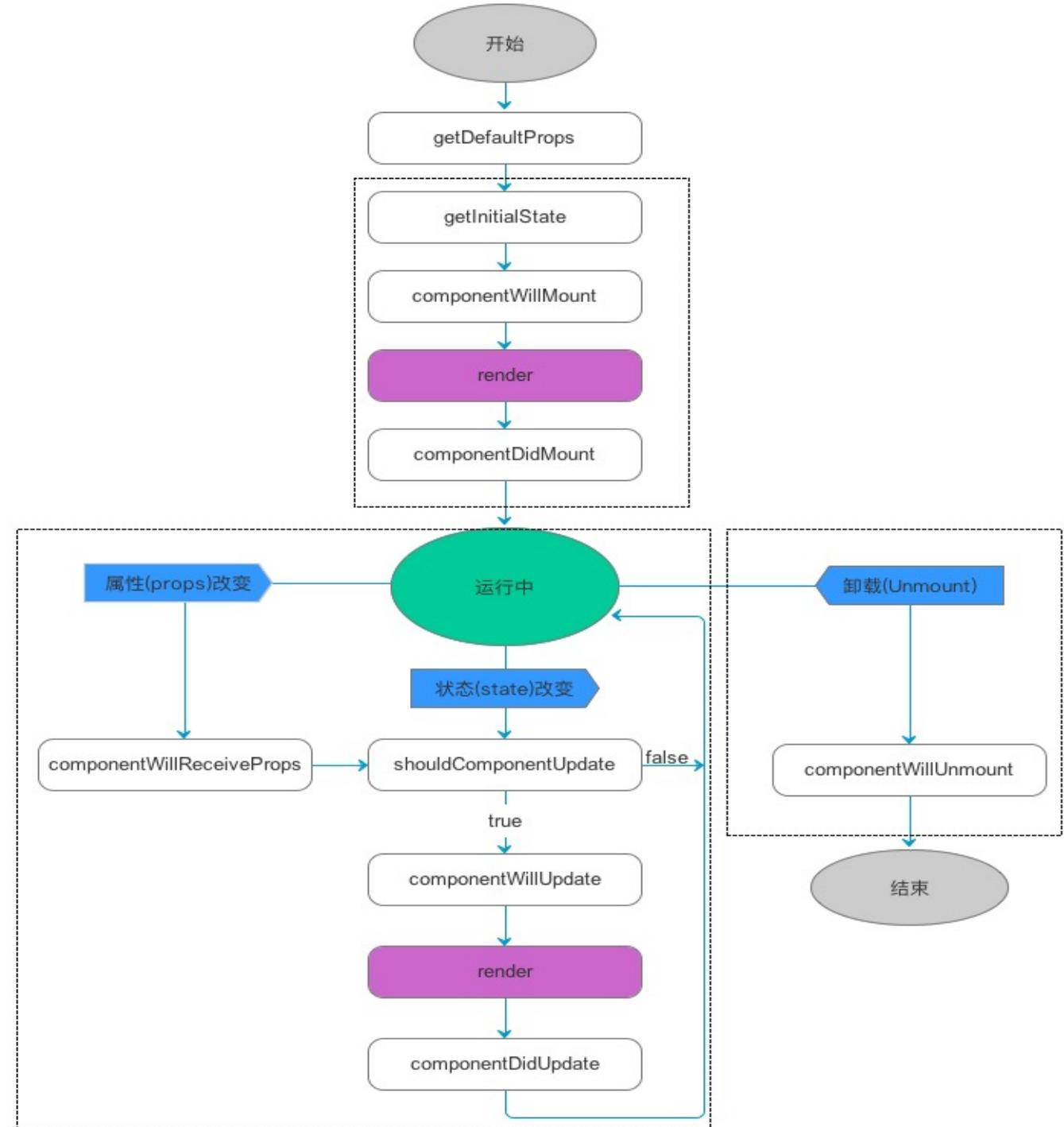
生命周期-更新



生命周期-销毁

- **componentWillUnmount()**
 - 组件将要卸载时调用，一些事件监听和定时器需要在此时清除。

```
ReactDOM.unmountComponentAtNode(  
  document.getElementById('root'));
```





Thank you

谢

谢

观

看