在实现一个React组件时可以选择两种编码方式，第一种是使用JSX编写

**class** **Hello** **extends** **Component** {

render() {

**return** <div>Hello ConardLi</div>;

}

}

第二种是直接使用React.createElement编写：

**class** **Hello** **extends** **Component** {

render() {

**return** React.createElement('div', null, `Hello ConardLi`);

}

}

实际上，上面两种写法是等价的，JSX只是为 React.createElement(component, props, ...children)方法提供的语法糖。也就是说所有的JSX代码最后都会转换成React.createElement(...)，Babel帮助我们完成了这个转换的过程。

注意，babel在编译时会判断JSX中组件的首字母，当首字母为小写时，其被认定为原生DOM标签，createElement的第一个变量被编译为字符串；当首字母为大写时，其被认定为自定义组件，createElement的第一个变量被编译为对象；

另外，由于JSX提前要被Babel编译，所以JSX是不能在运行时动态选择类型的，

**React组件的渲染流程**

* 使用React.createElement或JSX编写React组件，实际上所有的JSX代码最后都会转换成React.createElement(...)，Babel帮助我们完成了这个转换的过程。
* createElement函数对key和ref等特殊的props进行处理，并获取defaultProps对默认props进行赋值，并且对传入的孩子节点进行处理，最终构造成一个ReactElement对象（所谓的虚拟DOM）。
* ReactDOM.render将生成好的虚拟DOM渲染到指定容器上，其中采用了批处理、事务等机制并且对特定浏览器进行了性能优化，最终转换为真实DOM。

### 虚拟DOM的组成

即ReactElement element对象，我们的组件最终会被渲染成下面的结构：

* type：元素的类型，可以是原生html类型（字符串），或者自定义组件（函数或class）
* key：组件的唯一标识，用于Diff算法，下面会详细介绍
* ref：用于访问原生dom节点
* props：传入组件的props，children是props中的一个属性，它存储了当前组件的孩子节点，可以是数组（多个孩子节点）或对象（只有一个孩子节点）
* owner：当前正在构建的Component所属的Component
* self：（非生产环境）指定当前位于哪个组件实例
* \_source：（非生产环境）指定调试代码来自的文件(fileName)和代码行数(lineNumber)

**防止XSS**

ReactElement对象还有一个$$typeof属性，它是一个Symbol类型的变量Symbol.for('react.element')，当环境不支持Symbol时，$$typeof被赋值为0xeac7。

这个变量可以防止XSS。如果你的服务器有一个漏洞，允许用户存储任意JSON对象， 而客户端代码需要一个字符串，这可能为你的应用程序带来风险。JSON中不能存储Symbol类型的变量，而React渲染时会把没有$$typeof标识的组件过滤掉。

**针对性的性能优化**

在IE（8-11）和Edge浏览器中，一个一个插入无子孙的节点，效率要远高于插入一整个序列化完整的节点树。

React通过lazyTree，在IE（8-11）和Edge中进行单个节点依次渲染节点，而在其他浏览器中则首先将整个大的DOM结构构建好，然后再整体插入容器。

并且，在单独渲染节点时，React还考虑了fragment等特殊节点，这些节点则不会一个一个插入渲染。

**虚拟DOM事件机制**

React自己实现了一套事件机制，其将所有绑定在虚拟DOM上的事件映射到真正的DOM事件，并将所有的事件都代理到document上，自己模拟了事件冒泡和捕获的过程，并且进行统一的事件分发。

React自己构造了合成事件对象SyntheticEvent，这是一个跨浏览器原生事件包装器。 它具有与浏览器原生事件相同的接口，包括stopPropagation()和preventDefault()等等，在所有浏览器中他们工作方式都相同。这抹平了各个浏览器的事件兼容性问题。

# react事件和原生事件可以混用吗？

react事件和原生事件最好不要混用。

原生事件中如果执行了stopPropagation方法，则会导致其他react事件失效。因为所有元素的事件将无法冒泡到document上。

由上面的执行机制不难得出，所有的react事件都将无法被注册。

**为什么有时连续两次setState只有一次生效？**

1.调用setState不会立即更新

2.所有组件使用的是同一套更新机制，当所有组件didmount后，父组件didmount，然后执行更新

3.更新时会把每个组件的更新合并，每个组件只会触发一次更新的生命周期。

* 1.直接传递对象的setstate会被合并成一次
* 2.使用函数传递state不会被合并

不推荐直接在componentDidMount直接调用setState，由上面的分析：componentDidMount本身处于一次更新中，我们又调用了一次setState，就会在未来再进行一次render，造成不必要的性能浪费，大多数情况可以设置初始值来搞定。

当然在componentDidMount我们可以调用接口，再回调中去修改state，这是正确的做法。

当state初始值依赖dom属性时，在componentDidMount中setState是无法避免的。

## componentWillUpdate componentDidUpdate

这两个生命周期中不能调用setState。

由上面的流程图很容易发现，在它们里面调用setState会造成死循环，导致程序崩溃。

## 推荐使用方式

在调用setState时使用函数传递state值，在回调函数中获取最新更新后的state。

**1、redux中间件**

中间件提供第三方插件的模式，自定义拦截 **action -> reducer** 的过程。变为 **action -> middlewares -> reducer** 。这种机制可以让我们改变数据流，实现如异步 action ，action 过滤，日志输出，异常报告等功能。

常见的中间件：

redux-logger：提供日志输出

redux-thunk：处理异步操作

redux-promise：处理异步操作，actionCreator的返回值是promise

**2、redux有什么缺点**

1.一个组件所需要的数据，必须由父组件传过来，而不能像flux中直接从store取。

2.当一个组件相关数据更新时，即使父组件不需要用到这个组件，父组件还是会重新render，可能会有效率影响，或者需要写复杂的shouldComponentUpdate进行判断。

**3、react组件的划分业务组件技术组件？**

根据组件的职责通常把组件分为UI组件和容器组件。

UI 组件负责 UI 的呈现，容器组件负责管理数据和逻辑。

**展示组件嵌套在容器组件中，容器组件通过React-redux提供的connect方法拿到store中的数据**

**4、react生命周期函数**

这个问题要考察的是组件的生命周期

一、初始化阶段：

constructor

componentWillMount：组件即将被装载、渲染到页面上

render:组件在这里生成虚拟的DOM节点

componentDidMount:组件真正在被装载之后

二、运行中状态：

componentWillReceiveProps(nextProps):组件将要收到属性的时候调用(props有变化的时候调用)

shouldComponentUpdate(nextProps,nextState): 组件接受到新属性或者新状态的时候（可以返回false，接收数据后不更新，阻止render调用，后面的函数不会被继续执行了）(props state都会调用)

componentWillUpdate:组件即将更新不能修改属性和状态

render:组件重新描绘

componentDidUpdate:组件已经更新

三、销毁阶段：

componentWillUnmount:组件即将销毁

生命周期的方法有：

* **componentWillMount** 在渲染前调用,在客户端也在服务端。
* **componentDidMount** : 在第一次渲染后调用，只在客户端。之后组件已经生成了对应的DOM结构，可以通过this.getDOMNode()来进行访问。 如果你想和其他JavaScript框架一起使用，可以在这个方法中调用setTimeout, setInterval或者发送AJAX请求等操作(防止异步操作阻塞UI)。
* **componentWillReceiveProps** 在组件接收到一个新的 prop (更新后)时被调用。这个方法在初始化render时不会被调用。
* **shouldComponentUpdate** 返回一个布尔值。在组件接收到新的props或者state时被调用。在初始化时或者使用forceUpdate时不被调用。   
  可以在你确认不需要更新组件时使用。
* **componentWillUpdate**在组件接收到新的props或者state但还没有render时被调用。在初始化时不会被调用。
* **componentDidUpdate** 在组件完成更新后立即调用。在初始化时不会被调用。
* **componentWillUnmount**在组件从 DOM 中移除之前立刻被调用。

**5、react性能优化是哪个周期函数？**

shouldComponentUpdate 这个方法用来判断是否需要调用render方法重新描绘dom。因为dom的描绘非常消耗性能，如果我们能在shouldComponentUpdate方法中能够写出更优化的dom diff算法，可以极大的提高性能。

**6、为什么虚拟dom会提高性能?**

虚拟dom相当于在js和真实dom中间加了一个缓存，利用dom diff算法避免了没有必要的dom操作，从而提高性能。

具体实现步骤如下：

用 JavaScript 对象结构表示 DOM 树的结构；然后用这个树构建一个真正的 DOM 树，插到文档当中

当状态变更的时候，重新构造一棵新的对象树。然后用新的树和旧的树进行比较，记录两棵树差异

把2所记录的差异应用到步骤1所构建的真正的DOM树上，视图就更新了。

**7、diff算法?**

把树形结构按照层级分解，只比较同级元素。

给列表结构的每个单元添加唯一的key属性，方便比较。

dom树根元素tag变化直接销毁替换新的dom树，如果tag没变化就比较属性，最后递归比较子节点。

合并操作，调用 component 的 setState 方法的时候, React 将其标记为 dirty.到每一个事件循环结束, React 检查所有标记 dirty 的 component 重新绘制.

选择性子树渲染。开发人员可以重写shouldComponentUpdate提高diff的性能。

**8、react性能优化方案**

（1）重写shouldComponentUpdate来避免不必要的dom操作。

（2）使用 production 版本的react.js。

（3）使用key来帮助React识别列表中所有子组件的最小变化。

**9、简述flux 思想**

Flux 的最大特点，就是数据的"单向流动"。

1.用户访问 View

2.View 发出用户的 Action

3.Dispatcher 收到 Action，要求 Store 进行相应的更新

4.Store 更新后，发出一个"change"事件

5.View 收到"change"事件后，更新页面

#### 1 React 全部都是组件化的

React 是围绕可重用组件的概念设计的。你定义小组件并将它们组合在一起形成更大的组件。

无论大小，所有组件都是可重用的，甚至在不同的项目中也是如此。

在 React 顶级 API 中，createElement 函数是主函数。这是你需要学习的 7 个 API 中的 1 个。React 的 API 就是这么小。

JSX，可以单独使用，不仅仅适用于 React。

在 JSX 中，你可以在一对花括号内使用任何 JavaScript 表达式。

这是 JSX 内唯一的约束：只有表达式。例如，你不能使用 if 语句，但三元表达式是可以的。

JavaScript 变量也是表达式

JavaScript 对象也是表达式。

我们也可以在 JSX 中使用所有的 JavaScript 的集合方法（map、reduce 、filter、 concat等）。因为他们返回的也是表达式：

React 认为 {true}、 {false}、{undefined} 和 {null} 是有效元素，不呈现任何内容。

#### 4 你可以使用 JavaScript 类写 React 组件

简单的函数组件非常适合简单的需求，但是有的时候我们需要的更多。React 也支持通过使用 [JavaScript 类](https://link.juejin.im?target=https%3A%2F%2Fdeveloper.mozilla.org%2Fen-US%2Fdocs%2FWeb%2FJavaScript%2FReference%2FClasses)来创建组件。这里 Button 组件（在例 1 中）就是使用类的语法编写的。

class Button extends React.Component {

render() {

return <button>{this.props.label}</button>;

}

}

// 使用（相同的语法）

ReactDOM.render(<Button label="Save" />, mountNode);

例 9：使用 JavaScript 类创建组件

类的语法是非常简单的：定义一个扩展的 React.Component 类（另一个你需要学习的 React 的顶级 API）。该类定义了一个单一的实例函数 —— render()，并使函数返回虚拟 DOM 对象。每一次我们使用基于类的 Button 组件（例如，通过 <Button ... />）,React 将从这个基于类的组件中实例化对象，并在 DOM 树中使用该对象。

这就是为什么上面的例子中我们可以在 JSX 中使用 this.props.label 渲染输出的原因，因为每一个组件都有一个特殊的称为 props 的 **实例** 属性，这让所有的值传递给该组件时被实例化。

#### 5 React 中的事件：两个重要的区别

当处理 React 元素中的事件时，我们与 DOM API 的处理方式有两个非常重要的区别：

* 所有 React 元素属性（包括事件）都使用 **camelCase** 命名，而不是 **lowercase**。例如是 onClick 而不是 onclick。
* 我们将实际的 JavaScript 函数引用传递给事件处理程序，而不是字符串。例如是 onClick={handleClick} 而不是 onClick="handleClick"。

#### 6 每一个 React 组件都有一个故事：第 1 部分

以下仅适用于类组件（扩展 React.Component）。函数组件有一个稍微不同的故事。

1. 首先，我们定义了一个模板来创建组件中的元素。
2. 然后，我们在某处使用 React。例如，在 render 内部调用其他的组件，或者直接使用 ReactDOM.render。
3. 然后，React 实例化一个对象然后给它设置 **props** 然后我们可以通过 this.props 访问。这些属性都是我们在第 2 步传入的。
4. 因为这些全部都是 JavaScript，constructor 方法将会被调用（如果定义的话）。这是我们称之为的第一个：**组件生命周期方法**。
5. 接下来 React 计算渲染之后的输出方法（虚拟 DOM 节点）。
6. 因为这是 React 第一次渲染元素，React 将会与浏览器连通（代表我们使用 DOM API）来显示元素。这整个过程称为 **mounting**。
7. 接下来 React 调用另一个生命周期函数，称为 componentDidMount。我们可以这样使用这个方法，例如：在 DOM 上做一些我们现在知道的在浏览器中存在的东西。在此生命周期方法之前，我们使用的 DOM 都是虚拟的。
8. 一些组件的故事到此结束，其他组件得到卸载浏览器 DOM 中的各种原因。在后一种情况发生时，会调用另一个生命周期的方法，componentWillUnmount。
9. 任何 mounted 的元素的**状态**都可能会改变。该元素的父级可能会重新渲染。无论哪种情况，mounted 的元素都可能接收到不同的属性集。React 的魔力就是这儿，我们实际上需要的正是 React 的这一点！在这一点之前，说实话，我们并不需要 React。
10. 组价的故事还在继续，但是在此之前，我们需要理解我所说的这种**状态**。

#### 7 React 组件可以具有私有状态

以下只适用于类组件。

状态类是任何 React 类组件中的一个特殊字段。React 检测每一个组件状态的变化，但是为了 React 更加有效，我们必须通过 React 的另一个 API 改变状态字段，这就是我们要学习的另一个 API —— this.setState：

setState 实际上是一个异步方法

通过传入一个函数然后返回一个对象。

通过传入一个对象

**将渲染函数的输入视为两种：**

**通过父元素传入的属性 props**

**以及可以随时更新的内部私有状态 state**

**当渲染函数的输入改变时，输出可能也会改变。**

**React 保存了渲染的历史记录，当它看到一个渲染与前一个不同时，它将计算它们之间的差异，并将其有效地转换为在 DOM 中执行的实际 DOM 操作。**

#### 10 每一个 React 组件都有一个故事：第 2 部分

现在我们知道了一个组件的状态，当该状态发生变化的时候，我们来了解一下关于这个过程的最后几个概念。

1. 当组件的状态被更新时，或者它的父进程决定更改它传递给组件的属性时，组件可能需要重新渲染。
2. 如果后者发生，React 会调用另一个生命周期方法：componentWillReceiveProps。
3. 如果状态对象或传递的属性改变了，React 有一个重要的决定要做：组件是否应该在 DOM 中更新？这就是为什么它调用另一个重要的生命周期方法 shouldComponentUpdate 的原因 。此方法是一个实际问题，因此，如果需要自行定制或优化渲染过程，则必须通过返回 true 或 false 来回答这个问题。
4. 如果没有自定义 shouldComponentUpdate，React 的默认事件在大多数情况下都能处理的很好。
5. 首先，这个时候会调用另一生命周期的方法：componentWillUpdate。然后，React 将计算新渲染过的输出，并将其与最后渲染的输出进行对比。
6. 如果渲染过的输出和之前的相同，React 不进行处理（不需要和浏览器先生对话）。
7. 如果有不同的地方，React 将不同传达给浏览器，像我们之前看到的那样。
8. **在任何情况下，一旦一个更新程序发生了，无论以何种方式（即使有相同的输出），React 会调用最后的生命周期方法：componentDidUpdate。**

生命周期方法是逃生舱口。如果你没有做什么特别的事情，你可以在没有它们的情况下创建完整的应用程序。它们非常方便地分析应用程序中正在发生的事情，并进一步优化 React 更新的性能。

状态改变 -> 构建新的 DOM 元素更新页面

这里要提出的一种解决方案：一旦状态发生改变，就重新调用 render 方法，构建一个新的 DOM 元素。这样做的好处是什么呢？好处就是你可以在 render 方法里面使用最新的 this.state 来构造不同 HTML 结构的字符串，并且通过这个字符串构造不同的 DOM 元素。

这样的结果就是，用户每次点击，changeLikeText 都会调用改变组件状态然后调用 setState ；setState 会调用 render ，render 方法会根据 state 的不同重新构建不同的 DOM 元素。

也就是说，你**只要调用 setState，组件就会重新渲染**。我们顺利地消除了手动的 DOM 操作。

**组件化可以帮助我们解决前端结构的复用性问题，整个页面可以由这样的不同的组件组合、嵌套构成。**

一个组件有自己的显示形态（上面的 HTML 结构和内容）行为，组件的显示形态和行为可以由数据状态（state）和配置参数（props）共同决定。数据状态和配置参数的改变都会影响到这个组件的显示形态。

当数据变化的时候，组件的显示需要更新。所以如果组件化的模式能提供一种高效的方式自动化地帮助我们更新页面，那也就可以大大地降低我们代码的复杂度，带来更好的可维护性。

import React, { Component } from 'react'

我们在文件头部从 react 的包当中引入了 React 和 React.js 的组件父类 Component。记住，只要你要写 React.js 组件，那么就必须要引入这两个东西。

React.createElement 会构建一个 JavaScript 对象来描述你 HTML 结构的信息，包括标签名、属性、还有子元素等。这样的代码就是合法的 JavaScript 代码了。所以使用 React 和 JSX 的时候一定要经过编译的过程。

所谓的 JSX 其实就是 JavaScript 对象。

ReactDOM.render(

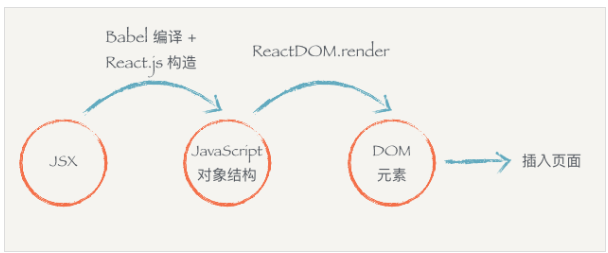
<Header />,

document.getElementById('root')

)

ReactDOM.render 功能就是把组件渲染并且构造 DOM 树，然后插入到页面上某个特定的元素上（在这里是 id 为 root 的 div 元素）。

可以总结一下从 JSX 到页面到底经过了什么样的过程



有些同学可能会问，为什么不直接从 JSX 直接渲染构造 DOM 结构，而是要经过中间这么一层呢？

第一个原因是，当我们拿到一个表示 UI 的结构和信息的对象以后，不一定会把元素渲染到浏览器的普通页面上，我们有可能把这个结构渲染到 canvas 上，或者是手机 App 上。所以这也是为什么会要把 react-dom 单独抽离出来的原因，可以想象有一个叫 react-canvas 可以帮我们把 UI 渲染到 canvas 上，或者是有一个叫 react-app 可以帮我们把它转换成原生的 App（实际上这玩意叫 ReactNative）。

第二个原因是，有了这样一个对象。当数据变化，需要更新组件的时候，就可以用比较快的算法操作这个 JavaScript 对象，而不用直接操作页面上的 DOM，这样可以尽量少的减少浏览器重排，极大地优化性能。

要记住几个点：

1)JSX 是 JavaScript 语言的一种语法扩展，长得像 HTML，但并不是 HTML。

2)React.js 可以用 JSX 来描述你的组件长什么样的。

3)JSX 在编译的时候会变成相应的 JavaScript 对象描述。

4)react-dom 负责把这个用来描述 UI 信息的 JavaScript 对象变成 DOM 元素，并且渲染到页面上。

一个组件类必须要实现一个 render 方法，这个 render 方法必须要返回一个 JSX 元素。但这里要注意的是，必须要用一个外层的 JSX 元素把所有内容包裹起来。返回并列多个 JSX 元素是不合法的，

在 JSX 当中你可以插入 JavaScript 的表达式，表达式返回的结果会相应地渲染到页面上。表达式用 {} 包裹。

简而言之，{} 内可以放任何 JavaScript 的代码，包括变量、表达式计算、函数执行等等。 render 会把这些代码返回的内容如实地渲染到页面上，非常的灵活。

表达式插入不仅仅可以用在标签内部，也可以用在标签的属性上

因为 class 是 JavaScript 的关键字，所以 React.js 中定义了一种新的方式：className 来帮助我们给元素添加类名。

还有一个特例就是 for 属性，例如 <label for='male'>Male</label>，因为 for 也是 JavaScript 的关键字，所以在 JSX 用 htmlFor 替代，即 <label htmlFor='male'>Male</label>。

{} 上面说了，JSX 可以放置任何表达式内容。所以也可以放 JSX，实际上，我们可以在 render 函数内部根据不同条件返回不同的 JSX。

JSX 元素变量

同样的，如果你能理解 JSX 元素就是 JavaScript 对象。那么你就可以联想到，JSX 元素其实可以像 JavaScript 对象那样自由地赋值给变量，或者作为函数参数传递、或者作为函数的返回值。

**自定义的组件都必须要用大写字母开头，普通的 HTML 标签都用小写字母开头**。

组件可以和组件组合在一起，组件内部可以使用别的组件。就像普通的 HTML 标签一样使用就可以。这样的组合嵌套，最后构成一个所谓的组件树，

在 React.js 不需要手动调用浏览器原生的 addEventListener 进行事件监听。React.js 帮我们封装好了一系列的 on\* 的属性，当你需要为某个元素监听某个事件的时候，只需要简单地给它加上 on\* 就可以了。而且你不需要考虑不同浏览器兼容性的问题，React.js 都帮我们封装好这些细节了

没有经过特殊处理的话，这些 on\* 的事件监听只能用在普通的 HTML 的标签上，而不能用在组件标签上。也就是说，<Header onClick={…} /> 这样的写法不会有什么效果的。这一点要注意，但是有办法可以做到这样的绑定，以后我们会提及。现在只要记住一点就可以了：这些 on\* 的事件监听只能用在普通的 HTML 的标签上，而不能用在组件标签上。

为 React 的组件添加事件监听是很简单的事情，你只需要使用 React.js 提供了一系列的 on\* 方法即可。

React.js 会给每个事件监听传入一个 event 对象，这个对象提供的功能和浏览器提供的功能一致，而且它是兼容所有浏览器的。

React.js 的事件监听方法需要手动 bind 到当前实例，这种模式在 React.js 中非常常用。

bind 会把实例方法绑定到当前实例上

一个组件的显示形态是可以由它数据状态和配置参数决定的。一个组件可以拥有自己的状态，就像一个点赞按钮，可以有“已点赞”和“未点赞”状态，并且可以在这两种状态之间进行切换。React.js 的 state 就是用来存储这种可变化的状态的。

**setState 方法由父类 Component 所提供。当我们调用这个函数的时候，React.js 会更新组件的状态 state ，并且重新调用 render 方法，然后再把 render 方法所渲染的最新的内容显示到页面上。**

注意，当我们要改变组件的状态的时候，不能直接用 this.state = xxx 这种方式来修改，如果这样做 React.js 就没办法知道你修改了组件的状态，它也就没有办法更新页面。所以，一定要使用 React.js 提供的 setState 方法，它接受一个对象或者函数作为参数。

传入一个对象的时候，这个对象表示该组件的新状态。但你只需要传入需要更新的部分就可以了，而不需要传入整个对象。

setState 接受函数参数

这里还有要注意的是，当你调用 setState 的时候，React.js 并不会马上修改 state。而是把这个对象放到一个更新队列里面，稍后才会从队列当中把新的状态提取出来合并到 state 当中，然后再触发组件更新。

setState 的第二种使用方式，可以接受一个函数作为参数。React.js 会把上一个 setState 的结果传入这个函数，你就可以使用该结果进行运算、操作，然后返回一个对象作为更新 state 的对象

setState 合并

这是因为在 React.js 内部会把 JavaScript 事件循环中的消息队列的同一个消息中的 setState 都进行合并以后再重新渲染组件。

深层的原理并不需要过多纠结，你只需要记住的是：在使用 React.js 的时候，并不需要担心多次进行 setState 会带来性能问题。

React.js 的 props 就可以帮助我们达到这个效果。每个组件都可以接受一个 props 参数，它是一个对象，包含了所有你对这个组件的配置。

从 render 函数可以看出来，组件内部是通过 this.props 的方式获取到组件的参数的，如果 this.props 里面有需要的属性我们就采用相应的属性，没有的话就用默认的属性。

一个组件的行为、显示形态都可以用 props 来控制，就可以达到很好的可配置性。

默认配置 defaultProps

props 不可变

props 一旦传入进来就不能改变。

1)为了使得组件的可定制性更强，在使用组件的时候，可以在标签上加属性来传入配置参数。

2)组件可以在内部通过 this.props 获取到配置参数，组件可以根据 props 的不同来确定自己的显示形态，达到可配置的效果。

3)可以通过给组件添加类属性 defaultProps 来配置默认参数。

4)props 一旦传入，你就不可以在组件内部对它进行修改。但是你可以通过父组件主动重新渲染的方式来传入新的 props，从而达到更新的效果。

**state 的主要作用是用于组件保存、控制、修改自己的可变状态。state 在组件内部初始化，可以被组件自身修改，而外部不能访问也不能修改。你可以认为 state 是一个局部的、只能被组件自身控制的数据源。state 中状态可以通过 this.setState 方法进行更新，setState 会导致组件的重新渲染。**

**props 的主要作用是让使用该组件的父组件可以传入参数来配置该组件。它是外部传进来的配置参数，组件内部无法控制也无法修改。除非外部组件主动传入新的 props，否则组件的 props 永远保持不变。**

***state* 是让组件控制自己的状态，*props* 是让外部对组件自己进行配置**。

请记住一个简单的规则：尽量少地用 state，尽量多地用 props。

没有 state 的组件叫无状态组件（stateless component），设置了 state 的叫做有状态组件（stateful component）。因为状态会带来管理的复杂性，我们尽量多地写无状态组件，尽量少地写有状态的组件。这样会降低代码维护的难度，也会在一定程度上增强组件的可复用性。

**如果你往 *{}* 放一个数组，React.js 会帮你把数组里面一个个元素罗列并且渲染出来**。

使用 map 渲染列表数据

对于用表达式套数组罗列到页面上的元素，都要为每个元素加上 key 属性，这个 key 必须是每个元素唯一的标识。

所以应该怎么做才能把用户内容输入更新到输入框当中呢？在 React.js 当中必须要用 setState 才能更新组件的内容，所以我们需要做的就是：监听输入框的 onChange 事件，然后获取到用户输入的内容，再通过 setState 的方式更新 state 中的 username，这样 input 的内容才会更新。

类似于 <input />、<select />、<textarea> 这些元素的 value 值被 React.js 所控制、渲染的组件，在 React.js 当中被称为受控组件（Controlled Component）。对于用户可输入的控件，一般都可以让它们成为受控组件，这是 React.js 所推崇的做法。另外还有非受控组件，这里暂时不提及。

**向父组件传递数据**

CommentInput 如何向 CommentApp 传递的数据？父组件 CommentApp 只需要通过 props 给子组件 CommentInput 传入一个回调函数。当用户点击发布按钮的时候，CommentInput 调用 props 中的回调函数并且将 state 传入该函数即可。



受控组件的概念，React.js 中的 <input /> 、<textarea />、<select /> 等元素的 value 值如果是受到 React.js 的控制，那么就是受控组件。

遇到这种情况，我们**将这种组件之间共享的状态交给组件最近的公共父节点保管**，然后通过 props 把状态传递给子组件，这样就可以在组件之间共享数据了。

当某个状态被多个组件依赖或者影响的时候，就把该状态提升到这些组件的最近公共父组件中去管理，用 props 传递数据或者函数来管理这种依赖或着影响的行为。

**React.js 将组件渲染，并且构造 DOM 元素然后塞入页面的过程称为组件的挂载**

-> constructor()

-> componentWillMount()

-> render()

// 然后构造 DOM 元素插入页面

-> componentDidMount()

// ...

// 即将从页面中删除

-> componentWillUnmount()

// 从页面中删除

componentWillMount：组件挂载开始之前，也就是在组件调用 render 方法之前调用。

componentDidMount：组件挂载完成以后，也就是 DOM 元素已经插入页面后调用。

componentWillUnmount：组件对应的 DOM 元素从页面中删除之前调用。

所有关于组件自身的状态的初始化工作都会放在 constructor 里面去做

一些组件启动的动作，包括像 Ajax 数据的拉取操作、一些定时器的启动等，就可以放在 componentWillMount 里面进行

组件从页面上销毁的时候，有时候需要一些数据的清理，例如定时器的清理，就会放在 componentWillUnmount 里面去做。

componentDidMount 。一般来说，有些组件的启动工作是依赖 DOM 的，例如动画的启动，而 componentWillMount 的时候组件还没挂载完成，所以没法进行这些启动工作，这时候就可以把这些操作放在 componentDidMount 当中。

除了挂载阶段，还有一种“更新阶段”。说白了就是 setState 导致 React.js 重新渲染组件并且把组件的变化应用到 DOM 元素上的过程，**这是一个组件的变化过程**。而 React.js 也提供了一系列的生命周期函数可以让我们在这个组件更新的过程执行一些操作。

**Props变化，会调用componentWillReceiveProps(nextProps)，state变化，不会调用componentWillReceiveProps(nextProps)。**

**props变化 componentWillReceiveProps(nextProps)-> shouldComponentUpdate(nextProps, nextState)-> componentWillUpdate()->render-> componentDidUpdate()**

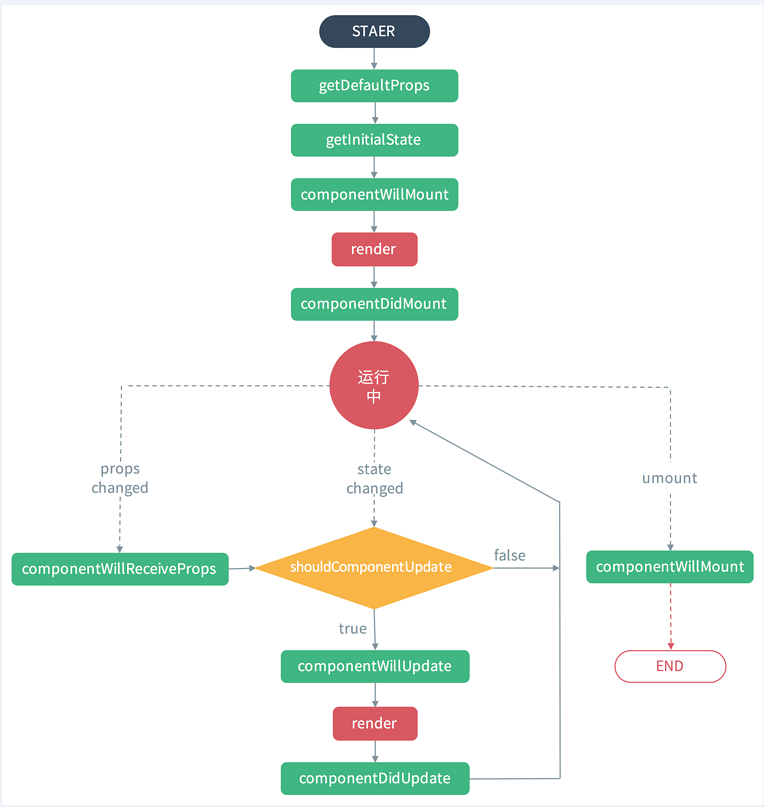
**state变化，shouldComponentUpdate(nextProps, nextState)-> componentWillUpdate()->render-> componentDidUpdate()**

componentWillReceiveProps(nextProps)：组件从父组件接收到新的 props 之前调用。

shouldComponentUpdate(nextProps, nextState)：你可以通过这个方法控制组件是否重新渲染。如果返回 false 组件就不会重新渲染。这个生命周期在 React.js 性能优化上非常有用。

componentWillUpdate()：组件开始重新渲染之前调用。

componentDidUpdate()：组件重新渲染并且把更改变更到真实的 DOM 以后调用。



**16和17的对比：**

React生命周期指的是组件从创建到卸载的整个过程，每个过程都有对应的钩子函数，它主要有以下几个阶段：

1、挂载阶段  -  组件实例被创建和插入Dom树的过程

2、更新阶段  -  组件被重新渲染的过程

3、卸载阶段  -  组件从Dom树中被删除的过程

早在React16.3就开始对生命周期做了一些改动，React16.3新增了两个生命周期函数：

1、static getDerivedStateFromProps(nextProps, prevState)

2、getSnapshotBeforeUpdate(prevProps, prevState)

新增的特性：

1、static getDerivedStateFromError(error)

2、componentDidCatch(error, info)

同时Facebook声明将在React17版本将删除以下生命周期函数：

1、componentWillMount

2、componentWillReceiveProps

3、componentWillUpdate

保留以下生命周期函数：

1、UNSAFE\_componentWillMount

2、UNSAFE\_componentWillReceiveProps

3、UNSAFE\_componentWillUpdate

下面就以17版本做为划分，React16.3新增的两个周期函数归为17版本。

17版本前生命周期

**挂载阶段**

这个阶段主要是做初始化操作，主要有这几个钩子函数：

static defaultProps

    -- 设置props的默认值

static propTypes

    -- props数据类型检查

constructor(props)

    -- 构造函数

componentWillMount()

    -- 组件挂载前钩子

render()

    -- 组件更新钩子

componentDidMount()

    -- 组件挂载成功钩子，该过程组件已经成功挂载到了真实Dom上

或许有小伙伴会问static静态方法是咋肥事，简单来说是它不依赖构造函数，只跟类有关，可优先于构造函数执行。

**更新阶段**

这个阶段主要是做状态更新操作，主要有这几个钩子函数：

componentWillReceiveProps(newProps)

    -- 父组件更新props钩子

shouldComponentUpdate()

     -- 组件是否更新钩子

componentWillUpdate()

    -- 组件更新前钩子

render()

    -- 组件渲染钩子

componentDidUpdate()

    -- 组件更新成功钩子

**卸载阶段**

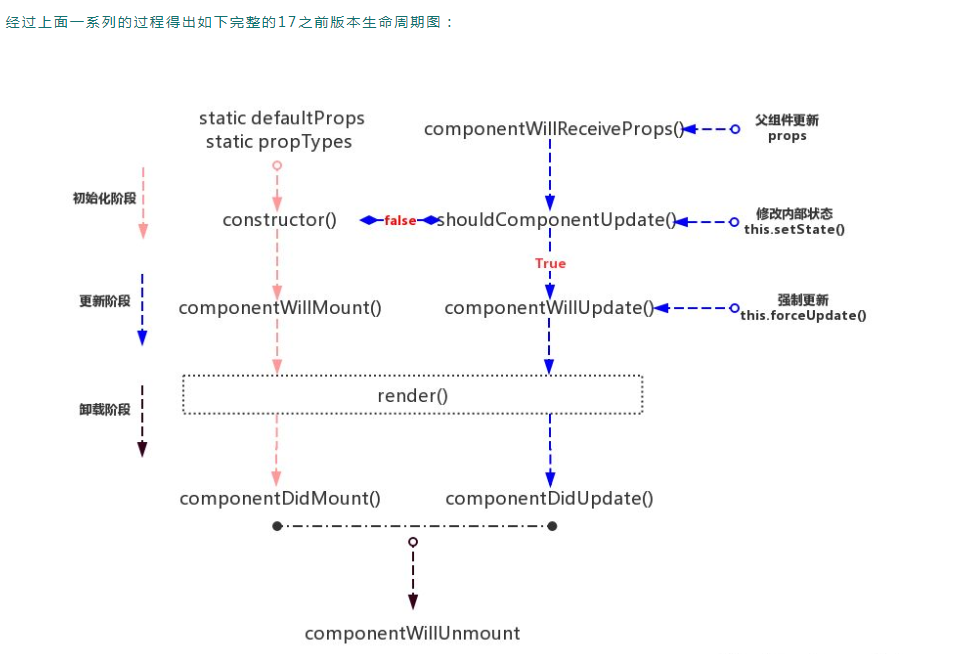
这个阶段主要是从Dom树中删除组件的操作，它的钩子函数：

componentWillUnmount()

-- 组件将要被卸载的时候调用

static defaultProps和static propTypes优先于constructor执行，因为如果父组件不向子组件传递props时，子组件会获取默认props并且进行静态类型检测。

只要父组件更新必然引起子组件的更新，不管子组件的props是否变化。组件是否需要更新取决于shouldComponentUpdate这个钩子函数，它的默认值为return true。false则代表不需要更新。这被称为“render浪潮”，性能上也会有所影响。



17版本生命周期

17版本新增了两个生命周期函数：

1、static getDerivedStateFromProps(nextProps, prevState)

2、getSnapshotBeforeUpdate(prevProps, prevState)

17版本删除了以下生命周期函数：

1、componentWillMount

2、componentWillReceiveProps

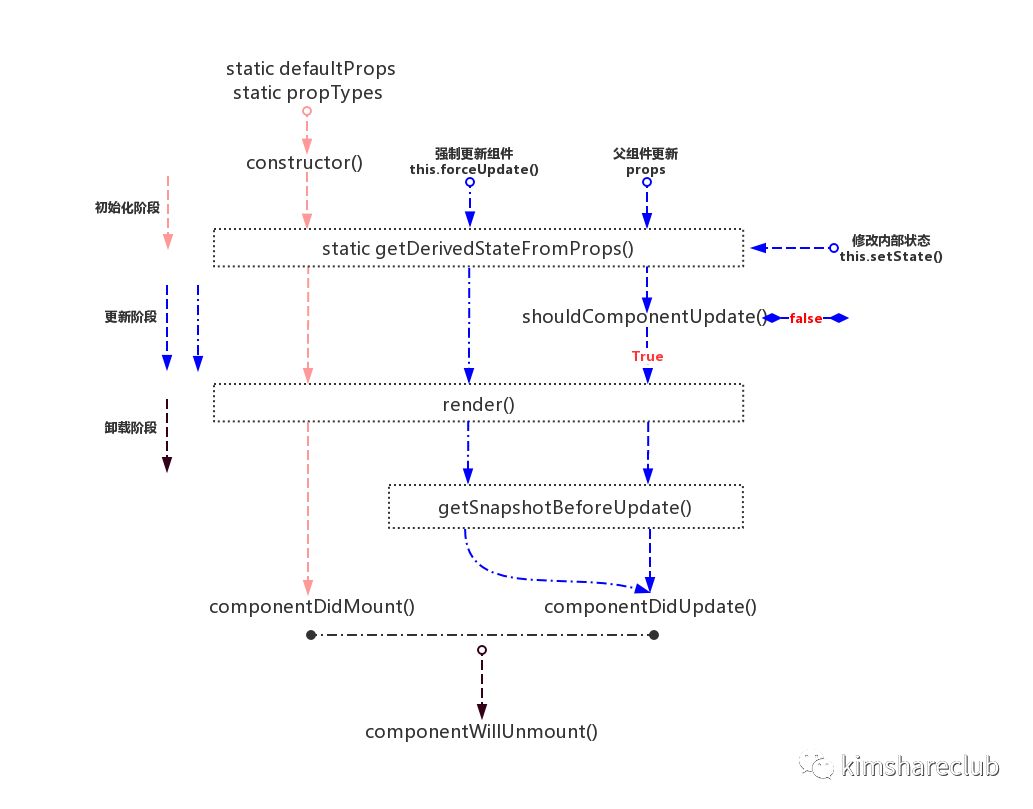
3、componentWillUpdate

所以在16.x版本中使用新生命周期时不可和这三个函数一起使用，否则会报错。

1、static getDerivedStateFormProps(nextProps, prevState)在render前调用，在初始挂载以及后续更新时都会被调用。他应该返回一个对象来更新state。如果返回null则不更新任何内容。它接收两个参数，一个是传进来的nextProps和之前的prevState。

2、getSnapshotBeforeUpdate(prevProps, prevState)在更新阶段render后挂载到真实Dom前进行的操作，它使得组件能在发生更改之前从DOM中捕获一些信息。此组件返回的任何值将作为componentDidUpdate的第三个参数。

3、经过上面一系列的过程得出如下完整的17版本生命周期图：



React.js 当中提供了 ref 属性来帮助我们获取已经挂载的元素的 DOM 节点，你可以给某个 JSX 元素加上 ref属性

可以给任意代表 HTML 元素标签加上 ref 从而获取到它 DOM 元素然后调用 DOM API。但是记住一个原则：能不用 ref 就不用。特别是要避免用 ref 来做 React.js 本来就可以帮助你做到的页面自动更新的操作和事件监听。多余的 DOM 操作其实是代码里面的“噪音”，不利于我们理解和维护。

其实可以给组件标签也加上 ref

所有嵌套在组件中的 JSX 结构都可以在组件内部通过 props.children 获取到

使用自定义组件的时候，可以在其中嵌套 JSX 结构。嵌套的结构在组件内部都可以通过 props.children 获取到，这种组件编写方式在编写容器类型的组件当中非常有用。而在实际的 React.js 项目当中，我们几乎每天都需要用这种方式来编写组件。

出于安全考虑的原因（XSS 攻击），在 React.js 当中所有的表达式插入的内容都会被自动转义，就相当于 jQuery 里面的 text(…) 函数一样，任何的 HTML 格式都会被转义掉：

React.js 提供了一个属性 dangerouslySetInnerHTML，可以让我们设置动态设置元素的 innerHTML

dangerouslySetInnerHTML={{\_\_html: this.state.content}}

需要给 dangerouslySetInnerHTML 传入一个对象，这个对象的 \_\_html 属性值就相当于元素的 innerHTML，这样我们就可以动态渲染元素的 innerHTML 结构了。

React.js 中的元素的 style 属性的用法和 DOM 里面的 style 不大一样，普通的 HTML 中的：

<h1 style='font-size: 12px; color: red;'>React.js 小书</h1>

在 React.js 中你需要把 CSS 属性变成一个对象再传给元素：

<h1 style={{fontSize: '12px', color: 'red'}}>React.js 小书</h1>

style 接受一个对象，这个对象里面是这个元素的 CSS 属性键值对，原来 CSS 属性中带 - 的元素都必须要去掉 - 换成驼峰命名，如 font-size 换成 fontSize，text-align 换成 textAlign。

于是 React.js 就提供了一种机制，让你可以**给组件的配置参数加上类型验证**

通过 PropTypes 给组件的参数做类型限制，可以在帮助我们迅速定位错误，这在构建大型应用程序的时候特别有用；另外，给组件加上 propTypes，也让组件的开发、使用更加规范清晰。

类（class）通过 static 关键字定义静态方法。不能在类的实例上调用静态方法，而应该通过类本身调用。

class ClassWithStaticMethod {

static staticMethod() {

return 'static method has been called.';

}

}

console.log(ClassWithStaticMethod.staticMethod());

组件的私有方法都用 \_ 开头，所有事件监听的方法都用 handle 开头。把事件监听方法传给组件的时候，属性名用 on 开头

另外，组件的内容编写顺序如下：

1. static 开头的类属性，如 defaultProps、propTypes。
2. 构造函数，constructor。
3. getter/setter（还不了解的同学可以暂时忽略）。
4. 组件生命周期。
5. \_ 开头的私有方法。
6. 事件监听方法，handle\*。
7. render\*开头的方法，有时候 render() 方法里面的内容会分开到不同函数里面进行，这些函数都以 render\* 开头。
8. render() 方法。

## 什么是高阶组件

**高阶组件就是一个函数，传给它一个组件，它返回一个新的组件。**

const NewComponent = higherOrderComponent(OldComponent)

重要的事情再重复一次，高阶组件是一个函数（而不是组件），它接受一个组件作为参数，返回一个新的组件。这个新的组件会使用你传给它的组件作为子组件

高阶组件的作用其实不言而喻，**其实就是为了组件之间的代码复用**。组件可能有着某些相同的逻辑，把这些逻辑抽离出来，放到高阶组件中进行复用。**高阶组件内部的包装组件和被包装组件之间通过 *props* 传递数据**。

代码复用的方法、形式有很多种，你可以用类继承来做到代码复用，也可以分离模块的方式。但是**高阶组件这种方式**很有意思，也很灵活。学过设计模式的同学其实应该能反应过来，它其实就**是设计模式里面的装饰者模式**。**它通过组合的方式达到很高的灵活程度。**

高阶组件就是一个函数，传给它一个组件，它返回一个新的组件。新的组件使用传入的组件作为子组件。

高阶组件的作用是用于代码复用，可以把组件之间可复用的代码、逻辑抽离到高阶组件当中。新的组件和传入的组件通过 props 传递信息。

高阶组件有助于提高我们代码的灵活性，逻辑的复用性。

一个组件可以通过 getChildContext 方法返回一个对象，这个对象就是子树的 context，提供 context 的组件必须提供 childContextTypes 作为 context 的声明和验证。

如果一个组件设置了 context，那么它的子组件都可以直接访问到里面的内容，它就像这个组件为根的子树的全局变量。任意深度的子组件都可以通过 contextTypes 来声明你想要的 context 里面的哪些状态，然后可以通过 this.context 访问到那些状态。

context 打破了组件和组件之间通过 props 传递数据的规范，极大地增强了组件之间的耦合性。而且，就如全局变量一样，context 里面的数据能被随意接触就能被随意修改，每个组件都能够改 context 里面的内容会导致程序的运行不可预料。

Redux 和 React-redux 并不是同一个东西。Redux 是一种架构模式（Flux 架构的一种变种），它不关注你到底用什么库，你可以把它应用到 React 和 Vue，甚至跟 jQuery 结合都没有问题。而 React-redux 就是把 Redux 这种架构模式和 React.js 结合起来的一个库，就是 Redux 架构在 React.js 中的体现。

**Flux 的最大特点，就是数据的"单向流动"。**

**1.用户访问 View**

**2.View 发出用户的 Action**

**3.Dispatcher 收到 Action，要求 Store 进行相应的更新**

**4.Store 更新后，发出一个"change"事件**

**5.View 收到"change"事件后，更新页面**

现在我们有了一个比较通用的 createStore，它可以产生一种我们新定义的数据类型 store，通过 store.getState 我们获取共享状态，而且我们约定只能通过 store.dispatch 修改共享状态。store 也允许我们通过 store.subscribe 监听数据数据状态被修改了，并且进行后续的例如重新渲染页面的操作。

**一个函数的返回结果只依赖于它的参数，并且在执行过程里面没有副作用，我们就把这个函数叫做纯函数**。

纯函数的第一个条件：**一个函数的返回结果只依赖于它的参数**。

函数执行过程没有副作用

为什么要煞费苦心地构建纯函数？因为纯函数非常“靠谱”，执行一个纯函数你不用担心它会干什么坏事，它不会产生不可预料的行为，也不会对外部产生影响。不管何时何地，你给它什么它就会乖乖地吐出什么。如果你的应用程序大多数函数都是由纯函数组成，那么你的程序测试、调试起来会非常方便。

在每个渲染函数执行渲染操作之前先做个判断，判断传入的新数据和旧的数据是不是相同，相同的话就不渲染了。



希望大家都知道这种 ES6 的语法：

const obj = { a: 1, b: 2}

const obj2 = { ...obj } *// => { a: 1, b: 2 }*

const obj2 = { ...obj } 其实就是新建一个对象 obj2，然后把 obj 所有的属性都复制到 obj2 里面，相当于对象的浅复制。上面的 obj 里面的内容和 obj2 是完全一样的，但是却是两个不同的对象。除了浅复制对象，还可以覆盖、拓展对象属性：

const obj = { a: 1, b: 2}

const obj2 = { ...obj, b: 3, c: 4} *// => { a: 1, b: 3, c: 4 }，覆盖了 b，新增了 c*

React 只更新它需要更新的部分

React DOM 会将元素和它的子元素与它们之前的状态进行比较，并只会进行必要的更新来使 DOM 达到预期的状态。

***React 只会更新必要的部分***

*值得注意的是 React DOM 首先会比较元素内容先后的不同，而在渲染过程中只会更新改变了的部分。*

使用 ES6 类写法，用 this.props.属性名 来取值。

JSX 是一个看起来很像 XML 的 JavaScript 语法扩展。

我们不需要一定使用 JSX，但它有以下优点：

1）JSX 执行更快，因为它在编译为 JavaScript 代码后进行了优化。

2）它是类型安全的，在编译过程中就能发现错误。

3）使用 JSX 编写模板更加简单快速。

JSX 是在 JavaScript 内部实现的。

我们知道元素是构成 React 应用的最小单位，JSX 就是用来声明 React 当中的元素。

React 推荐使用内联样式。我们可以使用 camelCase 语法来设置内联样式. React 会在指定元素数字后自动添加 px 。

注释需要写在花括号中。

1、在标签内部的注释需要花括号

2、在标签外部的注释不能使用花括号

ReactDOM.render(

/\*注释 \*/

<h1>孙朝阳 {/\*注释\*/}</h1>,

document.getElementById('example')

);

JSX 允许在模板中插入数组，数组会自动展开所有成员。

组件名不一定是用单标签，也可以是双标签

ReactDOM.render 是 React 的最基本方法用于将模板转为 HTML 语言，并插入指定的 DOM 节点。

ReactDOM.render(template,targetDOM) 方法接收两个参数：

第一个是创建的模板，多个 dom 元素外层需使用一个标签进行包裹，如 <div>；

第二个参数是插入该模板的目标位置。

若要为创建的某个元素增加 class 属性，不能直接定义 class 而要用 className，因为 class 是 javascript 中的保留字。

# React State(状态)

React 把组件看成是一个状态机（State Machines）。通过与用户的交互，实现不同状态，然后渲染 UI，让用户界面和数据保持一致。

React 里，只需更新组件的 state，然后根据新的 state 重新渲染用户界面（不要操作 DOM）。

### 数据自顶向下流动

父组件或子组件都不能知道某个组件是有状态还是无状态，并且它们不应该关心某组件是被定义为一个函数还是一个类。

这就是为什么状态通常被称为局部或封装。 除了拥有并设置它的组件外，其它组件不可访问。

这通常被称为自顶向下或单向数据流。 任何状态始终由某些特定组件所有，并且从该状态导出的任何数据或 UI 只能影响树中下方的组件。

在 React 应用程序中，组件是有状态还是无状态被认为是可能随时间而变化的组件的实现细节。

我们可以在有状态组件中使用无状态组件，也可以在无状态组件中使用有状态组件。

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。这就是为什么有些容器组件需要定义 state 来更新和修改数据。 而子组件只能通过 props 来传递数据。

可以通过组件类的 defaultProps 属性为 props 设置默认值

Props 验证使用 propTypes，它可以保证我们的应用组件被正确使用，React.PropTypes 提供很多验证器 (validator) 来验证传入数据是否有效。当向 props 传入无效数据时，JavaScript 控制台会抛出警告。

React 元素的事件处理和 DOM 元素类似。但是有一点语法上的不同:

React 事件绑定属性的命名采用驼峰式写法，而不是小写。

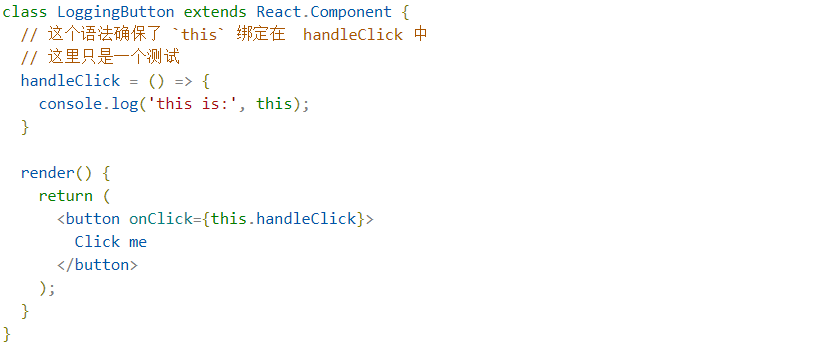
如果采用 JSX 的语法你需要传入一个函数作为事件处理函数，而不是一个字符串(DOM 元素的写法)

在 React 中另一个不同是你不能使用返回 **false** 的方式阻止默认行为， 你必须明确的使用 **preventDefault**。

你必须谨慎对待 JSX 回调函数中的 this，类的方法默认是不会绑定 this 的。如果你忘记绑定 this.handleClick 并把它传入 onClick, 当你调用这个函数的时候 this 的值会是 undefined。

这并不是 React 的特殊行为；它是函数如何在 JavaScript 中运行的一部分。通常情况下，如果你没有在方法后面添加 () ，例如 onClick={this.handleClick}，你应该为这个方法绑定 this。

1. **属性初始化器的方式**：（定义的时候用箭头函数）



2）**回调函数的方式**



**向事件处理程序传递参数**

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

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

参数 e 作为 React 事件对象将会被作为第二个参数进行传递。通过箭头函数的方式，事件对象必须显式的进行传递，但是通过 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。

值得注意的是，通过 bind 方式向监听函数传参，在类组件中定义的监听函数，事件对象 e 要排在所传递参数的后面，例如



事件：**this.handleclick.bind(this，要传的参数)**

函数：**handleclick(传过来的参数，event)**

es6中新加入了箭头函数=>，箭头函数除了方便之外还有而一个特征就是将函数的this绑定到其定义时所在的上下文。

**React 事件中 this 的三种使用方式**

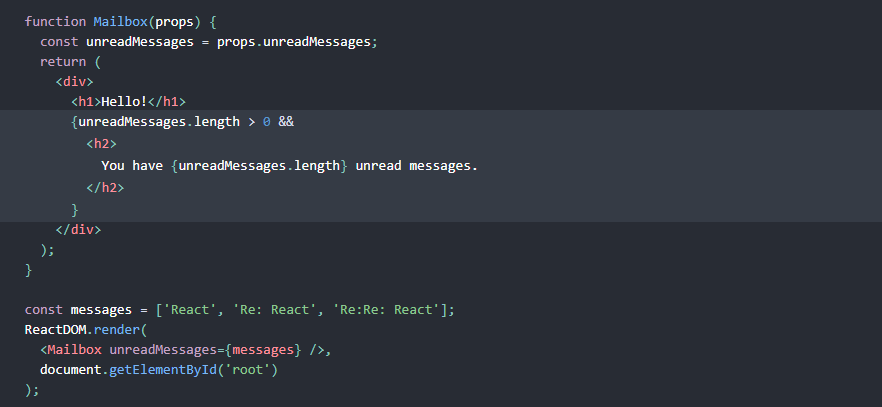
1. **bind绑定的方法**
2. **属性初始化器的方式**
3. **回调函数的方式**

**介绍几种在 JSX 中内联条件渲染的方法**

**1)** **与运算符 &&**

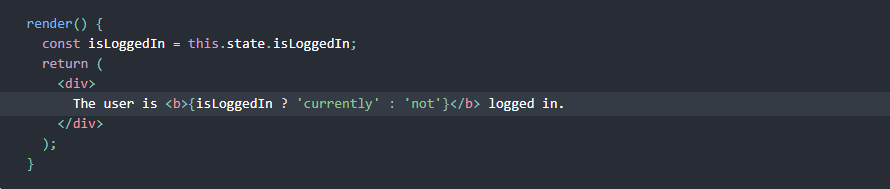
在 JavaScript 中，true && expression 总是返回 expression，而 false && expression 总是返回 false。

因此，如果条件是 true，&& 右侧的元素就会被渲染，如果是 false，React 会忽略并跳过它。



**2)三目运算符**

另一种内联条件渲染的方法是使用 JavaScript 中的三目运算符 condition ? true : false。



**阻止组件渲染**

在极少数情况下，你可能希望隐藏组件，即使它被其他组件渲染。让 render 方法返回 null 而不是它的渲染结果即可实现。

组件的 render 方法返回 null 并不会影响该组件生命周期方法的回调。例如，componentWillUpdate 和 componentDidUpdate 依然可以被调用。

**为什么要进行变量的初始化？**

一个软件所分配到的空间中极可能存在着以前其他软件使用过后的残留数据，这些数据被称之为垃圾数据。所以通常情况下我们为一个变量，为一个数组，分配好存储空间之后都要对该内存空间初始化。

简单来说就是清理残留数据。

setState() 可以接收一个函数，这个函数接受两个参数，第一个参数表示上一个状态值，第二参数表示当前的 props：

this.setState((prevState, props) => ({

//do something here

}));

**Keys**

Keys 可以在 DOM 中的某些元素被增加或删除的时候帮助 React 识别哪些元素发生了变化。因此你应当给数组中的每一个元素赋予一个确定的标识。

一个元素的 key 最好是这个元素在列表中拥有的一个独一无二的字符串。通常，我们使用来自数据的 id 作为元素的 key

元素的 key 只有在它和它的兄弟节点对比时才有意义。

**元素的 key 在他的兄弟元素之间应该唯一**

数组元素中使用的 key 在其兄弟之间应该是独一无二的。然而，它们不需要是全局唯一的。当我们生成两个不同的数组时，我们可以使用相同的键。

可以使用 JavaScript 的 map() 方法来创建列表。

**setState() 这里有三件事情需要知道**

**1、不要直接更新状态**

例如，此代码不会重新渲染组件：

// Wrong

this.state.comment = 'Hello';

应当使用 setState():

// Correct

this.setState({comment: 'Hello'});

**2、状态更新可能是异步的**

React 可以将多个 setState() 调用合并成一个调用来提高性能。

因为 this.props 和 this.state 可能是异步更新的，你不应该依靠它们的值来计算下一个状态。

要修复它，请使用第二种形式的 setState() 来接受一个函数而不是一个对象。 该函数将接收先前的状态作为第一个参数，将此次更新被应用时的props做为第二个参数：

// Correct

this.setState((prevState, props) => ({

counter: prevState.counter + props.increment

}));

**3、状态更新合并**

当你调用 setState() 时，React 将你提供的对象合并到当前状态。

例如，你的状态可能包含一些独立的变量：

constructor(props) {

super(props);

this.state = {

posts: [],

comments: []

};

}

你可以调用 setState() 独立地更新它们：

componentDidMount() {

fetchPosts().then(response => {

this.setState({

posts: response.posts

});

});

fetchComments().then(response => {

this.setState({

comments: response.comments

});

});

}

这里的合并是浅合并，也就是说 this.setState({comments}) 完整保留了 this.state.posts，但完全替换了 this.state.comments。

React 组件的数据可以通过 componentDidMount 方法中的 Ajax 来获取，当从服务端获取数据时可以将数据存储在 state 中，再用 this.setState 方法重新渲染 UI。

当使用异步加载数据时，在组件卸载前使用 componentWillUnmount 来取消未完成的请求。

HTML 表单元素与 React 中的其他 DOM 元素有所不同,因为表单元素生来就保留一些内部状态。

在 HTML 当中，像 <input>, <textarea>, 和 <select> 这类表单元素会维持自身状态，并根据用户输入进行更新。但在React中，可变的状态通常保存在组件的状态属性中，并且只能用 setState() 方法进行更新。

**Select 下拉菜单**

在 React 中，不使用 selected 属性，而在根 select 标签上用 value 属性来表示选中项。

**React Refs**

React 支持一种非常特殊的属性 Ref ，你可以用来绑定到 render() 输出的任何组件上。

这个特殊的属性允许你引用 render() 返回的相应的支撑实例（ backing instance ）。这样就可以确保在任何时间总是拿到正确的实例。

Props 和组合为你提供了清晰而安全地定制组件外观和行为的灵活方式。注意：组件可以接受任意 props，包括基本数据类型，React 元素以及函数。

**错误边界**

过去，组件内的 JavaScript 错误会导致 React 的内部状态被破坏，并且在下一次渲染时 产生 可能无法追踪的 错误。这些错误基本上是由较早的其他代码（非 React 组件代码）错误引起的，但 React 并没有提供一种在组件中优雅处理这些错误的方式，也无法从错误中恢复。

**错误边界（Error Boundaries）**

部分 UI 的 JavaScript 错误不应该导致整个应用崩溃，为了解决这个问题，React 16 引入了一个新的概念 —— 错误边界。

错误边界是一种 React 组件，这种组件可以捕获并打印发生在其子组件树任何位置的 JavaScript 错误，并且，它会渲染出备用 UI，而不是渲染那些崩溃了的子组件树。错误边界在渲染期间、生命周期方法和整个组件树的构造函数中捕获错误。

如果一个 class 组件中定义了 static getDerivedStateFromError() 或 componentDidCatch() 这两个生命周期方法中的任意一个（或两个）时，那么它就变成一个错误边界。当抛出错误后，请使用 static getDerivedStateFromError() 渲染备用 UI ，使用 componentDidCatch() 打印错误信息。

key 是唯一可以传递给 Fragment 的属性。

**高阶组件（HOC）**是 React 中用于复用组件逻辑的一种高级技巧。HOC 自身不是 React API 的一部分，它是一种基于 React 的组合特性而形成的设计模式。

具体而言，高阶组件是参数为组件，返回值为新组件的函数。

const EnhancedComponent = higherOrderComponent(WrappedComponent);

组件是将 props 转换为 UI，而高阶组件是将组件转换为另一个组件。

请注意，HOC 不会修改传入的组件，也不会使用继承来复制其行为。相反，HOC 通过将组件包装在容器组件中来组成新组件。HOC 是纯函数，没有副作用。

不要改变原始组件。使用组合。

不要试图在 HOC 中修改组件原型（或以其他方式改变它）。

**不要在 render 方法中使用 HOC**

React 的 diff 算法（称为协调）使用组件标识来确定它是应该更新现有子树还是将其丢弃并挂载新子树。 如果从 render 返回的组件与前一个渲染中的组件相同（===），则 React 通过将子树与新子树进行区分来递归更新子树。 如果它们不相等，则完全卸载前一个子树。

实际上，JSX 仅仅只是 React.createElement(component, props, ...children) 函数的语法糖。

**setState 只在合成事件和钩子函数中是“异步”的，在原生事件和 setTimeout 中都是同步的。**

**React合成事件和原生事件区别**

React合成事件一套机制：React并不是将click事件直接绑定在dom上面，而是采用事件冒泡的形式冒泡到document上面，然后React将事件封装给正式的函数处理运行和处理。

**React合成事件理解**

如果DOM上绑定了过多的事件处理函数，整个页面响应以及内存占用可能都会受到影响。React为了避免这类DOM事件滥用，同时屏蔽底层不同浏览器之间的事件系统差异，实现了一个中间层——SyntheticEvent。

1.当用户在为onClick添加函数时，React并没有将Click时间绑定在DOM上面。

2.而是在document处监听所有支持的事件，当事件发生并冒泡至document处时，React将事件内容封装交给中间层SyntheticEvent（负责所有事件合成）

3.所以当事件触发的时候，对使用统一的分发函数dispatchEvent将指定函数执行。

**以下用代码来展示两者的区别：**

class Test extends Component {

constructor() {

super(arguments);

this.onReactClick.bind(this);

}

componentDidMount() {

const parentDom = ReactDOM.findDOMNode(this);

const childrenDom = parentDom.queneSelector(".button");

childrenDom .addEventListener('click', this.onDomClick, false);

}

onDomClick() { // 事件委托

console.log('Javascript Dom click');

}

onReactClick() { // react合成事件

console.log('React click');

}

render() {

<div>

<button className="button" onClick={this.onReactClick()}>点击</button>

</div>

}

}

**结果：**

Dom click

React click

**可以看待原生绑定快于合成事件绑定。**

**注意点：**

不要将原生事件（addEventListener）和React合成事件一起混合使用，这两个机制是不一样的。

**redux和vuex的区别**

1)redux 是 flux 的一种实现，redux 不单单可以用在 react 上面。

2)vuex 是 redux 的基础上进行改变，对仓库的管理更加明确。

3)数据流向不一样，vuex 的同异步有不同的流向，而 redux 的同异步是一样的。

4)redux 使用的是不可变的数据，而 vuex 的数据是可变的，redux 每次修改更新数据，其实就是用新的数据替换旧的数据，而 vuex 是直接修改原数据。

**react 和 vue 的 diff 过程有什么区别**

React 是这么干的：你给我一个数据，我根据这个数据生成一个全新的 Virtual DOM，然后跟我上一次生成的 Virtual DOM 去 diff，得到一个 Patch，然后把这个 Patch 打到浏览器的 DOM 上去。完事。并且这里的 Patch 显然不是完整的 Virtual DOM，而是新的 Virtual DOM 和上一次的 Virtual DOM 经过 diff 后的差异化的部分。

Vue 在渲染过程中，会跟踪每一个组件的依赖关系，不需要重新渲染整个组件树。

React 每当应用的状态被改变时，全部子组件都会重新渲染。这可以通过 shouldComponentUpdate 这个生命周期方法来进行控制。

React diff的是 Dom，而 Vue diff 的是数据。

**下面是几个适合使用 refs 的情况**

管理焦点，文本选择或媒体播放。

触发强制动画。

集成第三方 DOM 库。

避免使用 refs 来做任何可以通过声明式实现来完成的事情。

**你不能在函数组件上使用 ref 属性，因为它们没有实例**

**可以在函数组件内部使用 ref 属性，只要它指向一个 DOM 元素或 class 组件**

**Render Props**

术语 “render prop” 是指一种在 React 组件之间使用一个值为函数的 prop 共享代码的简单技术

具有 render prop 的组件接受一个函数，该函数返回一个 React 元素并调用它而不是实现自己的渲染逻辑。

<DataProvider render={data => (

<h1>Hello {data.target}</h1>

)}/>