React

1. 什么是React

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”。

* 声明式

React 使创建交互式 UI 变得轻而易举。为你应用的每一个状态设计简洁的视图，当数据变动时 React 能高效更新并渲染合适的组件。

* 组件化

构建管理自身状态的封装组件，然后对其组合以构成复杂的 UI。

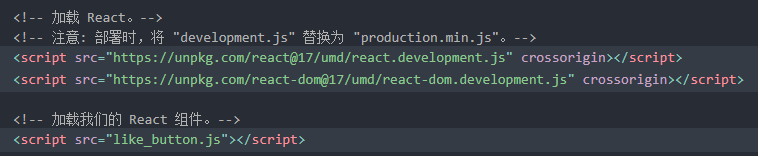
* 跨平台

无论你现在使用什么技术栈，在无需重写现有代码的前提下，通过引入 React 来开发新功能。

1. 安装

* 标签式

React 在设计之初就可以被渐进式适配，并且**你可以根据需要选择性地使用 React**。可能你只想在现有页面中“局部地添加交互性”。使用 React 组件是一种不错的方式。

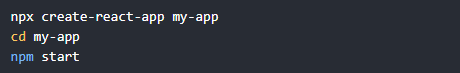


C:\Users\ADMINI~1\AppData\Local\Temp\1646186140(1).png

在项目中尝试 JSX 最快的方法是在页面中添加这个 <script> 标签：

C:\Users\ADMINI~1\AppData\Local\Temp\1646186222(1).png

* 脚手架



1. JSX

C:\Users\ADMINI~1\AppData\Local\Temp\1646186982(1).png

JSX，是一个 JavaScript 的语法扩展。JSX 可以很好地描述 UI 应该呈现出它应有交互的本质形式。它具有 JavaScript 的全部功能。

* JSX中嵌入表达式

C:\Users\ADMINI~1\AppData\Local\Temp\1646187138(1).png

* JSX表示对象

Babel 会把 JSX 转译成一个名为 React.createElement() 函数调用。



1. 元素渲染

元素是构成 React 应用的最小砖块。与浏览器的 DOM 元素不同，React 元素是创建开销极小的普通对象。React DOM 会负责更新 DOM 来与 React 元素保持一致。

* 想要将一个 React 元素渲染到根 DOM 节点中，只需把它们一起传入 [ReactDOM.render()](https://zh-hans.reactjs.org/docs/react-dom.html#render)：

C:\Users\ADMINI~1\AppData\Local\Temp\1646187993(1).png

* 更新已渲染的元素

React 元素是[不可变对象](https://en.wikipedia.org/wiki/Immutable_object)。一旦被创建，你就无法更改它的子元素或者属性。一个元素就像电影的单帧：它代表了某个特定时刻的 UI。

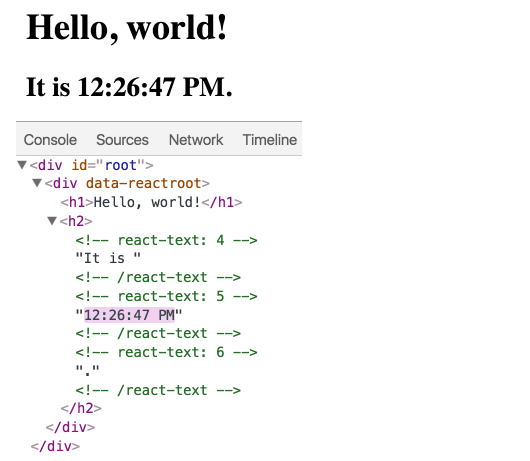
更新 UI 唯一的方式是创建一个全新的元素，并将其传入 [ReactDOM.render()](https://zh-hans.reactjs.org/docs/react-dom.html#render)。

**注意：**

在实践中，大多数 React 应用只会调用一次 [ReactDOM.render()](https://zh-hans.reactjs.org/docs/react-dom.html#render)。

* 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。



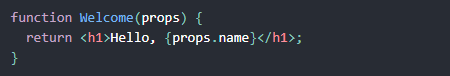
尽管每一秒我们都会新建一个描述整个 UI 树的元素，React DOM 只会更新实际改变了的内容，也就是例子中的文本节点。

1. 组件

组件允许你将 UI 拆分为独立可复用的代码片段，并对每个片段进行独立构思。

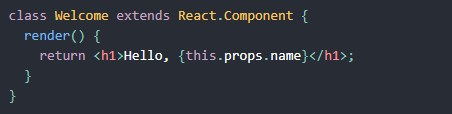
组件，从概念上类似于 JavaScript 函数。它接受任意的入参（即 “props”），并返回用于描述页面展示内容的 React 元素。

* 函数组件



该函数是一个有效的 React 组件，因为它接收唯一带有数据的 “props”（代表属性）对象与并返回一个 React 元素。这类组件被称为“函数组件”，因为它本质上就是 JavaScript 函数。

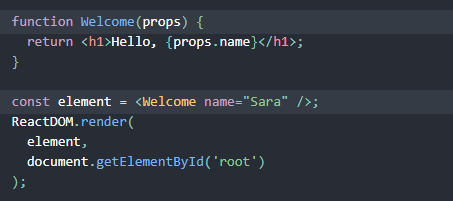
* 类组件

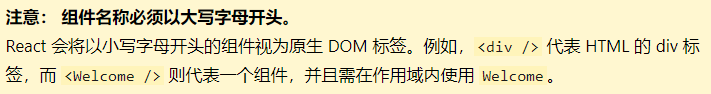


* 渲染组件

C:\Users\ADMINI~1\AppData\Local\Temp\1646189466(1).png

当 React 元素为用户自定义组件时，它会将 JSX 所接收的属性（attributes）以及子组件（children）转换为单个对象传递给组件，这个对象被称之为 “props”。





* Props的只读性

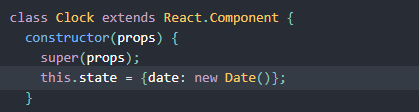
组件无论是使用[函数声明还是通过 class 声明](https://zh-hans.reactjs.org/docs/components-and-props.html#function-and-class-components)，都决不能修改自身的 props。

React 非常灵活，但它也有一个严格的规则：

**所有 React 组件都必须像纯函数一样保护它们的 props 不被更改。**

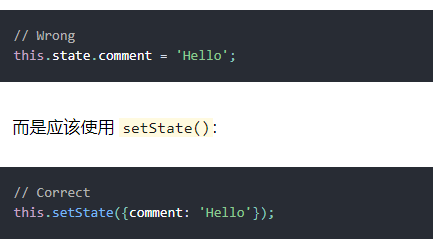
1. State

State 与 props 类似，但是 state 是**私有的**，并且完全受控于当前组件。



通过以上方式将 props 传递到父类的构造函数中

* 正确使用State
* 不要直接修改State

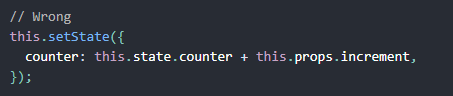


构造函数是唯一可以给 this.state 赋值的地方。

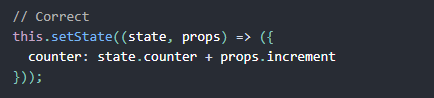
* State更新可能是异步

出于性能考虑，React 可能会把多个 setState() 调用合并成一个调用。

因为 this.props 和 this.state 可能会异步更新，所以你不要依赖他们的值来更新下一个状态。



要解决以这个问题，可以让 setState() 接收一个函数而不是一个对象。这个函数用上一个 state 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：



* State更新可能会被合并

当你调用 setState() 的时候，React 会把你提供的对象合并到当前的 state。

* 数据是向下流动的

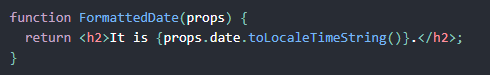
不管是父组件或是子组件都无法知道某个组件是有状态的还是无状态的，并且它们也并不关心它是函数组件还是 class 组件。

这就是为什么称 state 为局部的或是封装的的原因。除了拥有并设置了它的组件，其他组件都无法访问。

组件可以选择把它的 state 作为 props 向下传递到它的子组件中：

*C:\Users\ADMINI~1\AppData\Local\Temp\1646191385(1).png*

FormattedDate 组件会在其 props 中接收参数 date，但是组件本身无法知道它是来自于 Clock 的 state，或是 Clock 的 props，还是手动输入的：

**

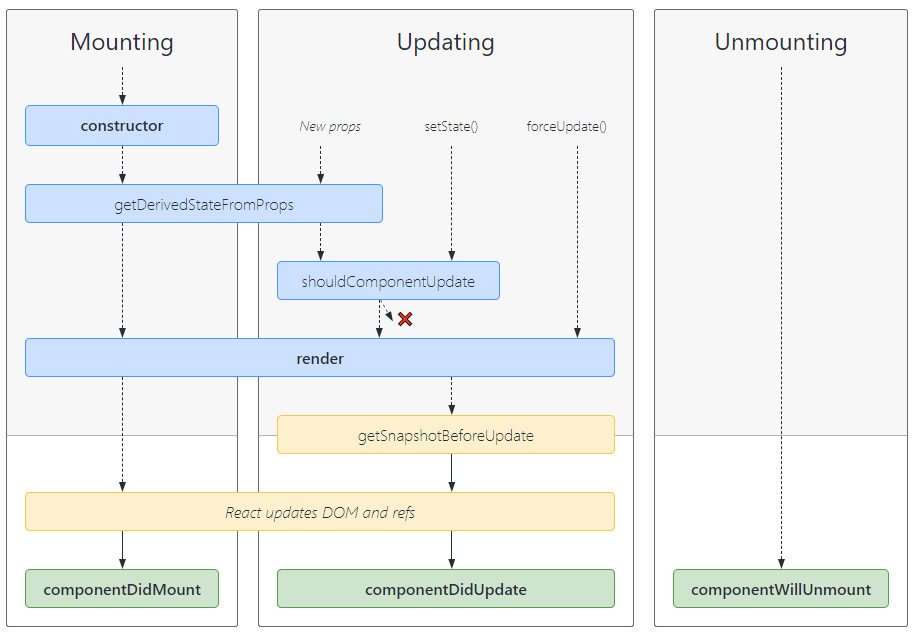
这通常会被叫做“自上而下”或是“单向”的数据流。任何的 state 总是所属于特定的组件，而且从该 state 派生的任何数据或 UI 只能影响树中“低于”它们的组件。

如果你把一个以组件构成的树想象成一个 props 的数据瀑布的话，那么每一个组件的 state 就像是在任意一点上给瀑布增加额外的水源，但是它只能向下流动。

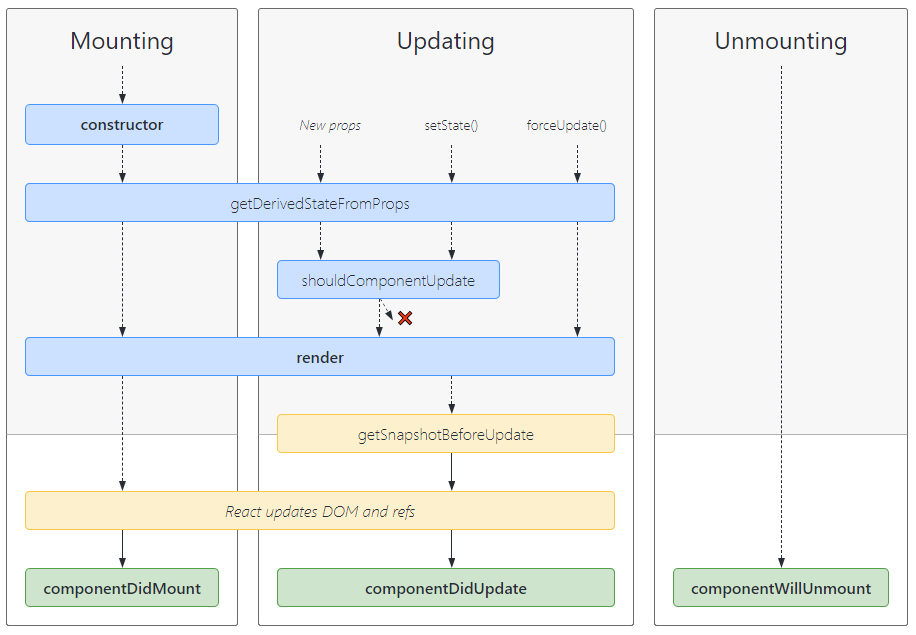
1. 组件生命周期

每个组件都包含 “生命周期方法”，你可以重写这些方法，以便于在运行过程中特定的阶段执行这些方法。

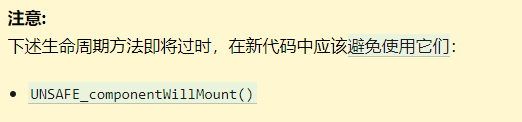
生命周期图谱16.3版本：



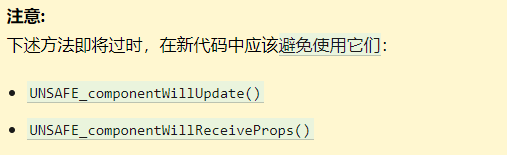
生命周期图谱16.4或以上版本：



* 挂载时（当组件实例被创建并插入 DOM 中时，其生命周期调用顺序如下）：
* [**constructor()**](https://zh-hans.reactjs.org/docs/react-component.html#constructor)
* [static getDerivedStateFromProps()](https://zh-hans.reactjs.org/docs/react-component.html#static-getderivedstatefromprops)
* [**render()**](https://zh-hans.reactjs.org/docs/react-component.html#render)
* [**componentDidMount()**](https://zh-hans.reactjs.org/docs/react-component.html#componentdidmount)



* 更新时（当组件的 props 或 state 发生变化时会触发更新）：
* [**static getDerivedStateFromProps()**](https://zh-hans.reactjs.org/docs/react-component.html#static-getderivedstatefromprops)
* [**shouldComponentUpdate()**](https://zh-hans.reactjs.org/docs/react-component.html#shouldcomponentupdate)
* [**render()**](https://zh-hans.reactjs.org/docs/react-component.html#render)
* [getSnapshotBeforeUpdate()](https://zh-hans.reactjs.org/docs/react-component.html#getsnapshotbeforeupdate)
* [**componentDidUpdate()**](https://zh-hans.reactjs.org/docs/react-component.html#componentdidupdate)

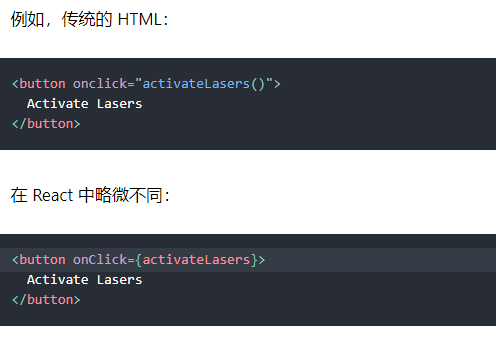


* 卸载时 (当组件从 DOM 中移除时会调用如下方法)：
* [**componentWillUnmount()**](https://zh-hans.reactjs.org/docs/react-component.html#componentwillunmount)
* 错误处理时（当渲染过程，生命周期，或子组件的构造函数中抛出错误）：
* [**static getDerivedStateFromError()**](https://zh-hans.reactjs.org/docs/react-component.html#static-getderivedstatefromerror)
* [**componentDidCatch()**](https://zh-hans.reactjs.org/docs/react-component.html#componentdidcatch)

1. 事件处理

React 事件的命名采用小驼峰式（camelCase），而不是纯小写。

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。



在 React 中另一个不同点是你不能通过返回 false 的方式阻止默认行为。你必须显式的使用 preventDefault。例如，传统的 HTML 中阻止表单的默认提交行为，你可以这样写：



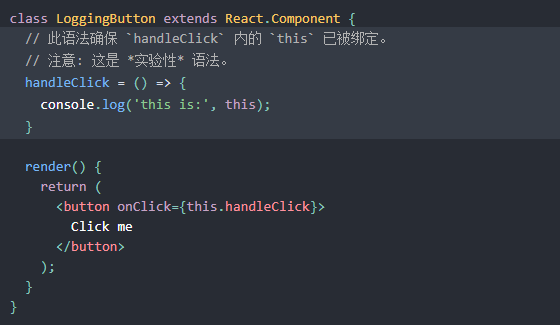
* This

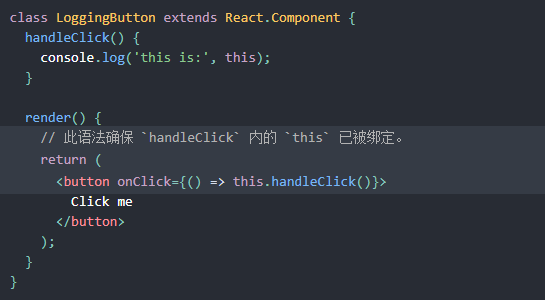
你必须谨慎对待 JSX 回调函数中的 this，在 JavaScript 中，class 的方法默认不会[绑定](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_objects/Function/bind) this。如果你忘记绑定 this.handleClick 并把它传入了 onClick，当你调用这个函数的时候 this 的值为 undefined。



这并不是 React 特有的行为；这其实与 [JavaScript 函数工作原理](https://www.smashingmagazine.com/2014/01/understanding-javascript-function-prototype-bind/)有关。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。

如果觉得使用 bind 很麻烦，这里有两种方式可以解决。





* 向事件处理程序传递参数

C:\Users\ADMINI~1\AppData\Local\Temp\1646198674(1).png

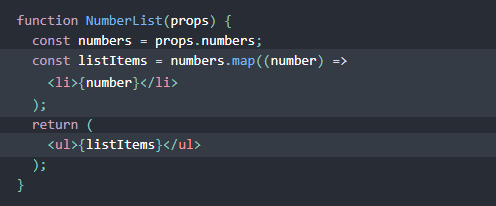
上述两种方式是等价的，分别通过[箭头函数](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions)和 [Function.prototype.bind](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_objects/Function/bind) 来实现

1. 条件渲染

在 React 中，你可以创建不同的组件来封装各种你需要的行为。然后，依据应用的不同状态，你可以只渲染对应状态下的部分内容。

React 中的条件渲染和 JavaScript 中的一样，使用 JavaScript 运算符 [if](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/if...else) 或者[条件运算符](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Conditional_Operator)去创建元素来表现当前的状态，然后让 React 根据它们来更新 UI。

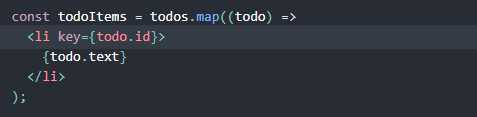
1. 渲染列表



当我们运行这段代码，将会看到一个警告 a key should be provided for list items，意思是当你创建一个元素时，必须包括一个特殊的 key 属性。

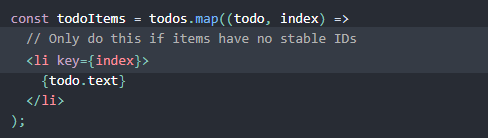
* Key

key 帮助 React 识别哪些元素改变了，比如被添加或删除。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，我们使用数据中的 id 来作为元素的 key：

当元素没有确定 id 的时候，万不得已你可以使用元素索引 index 作为 key



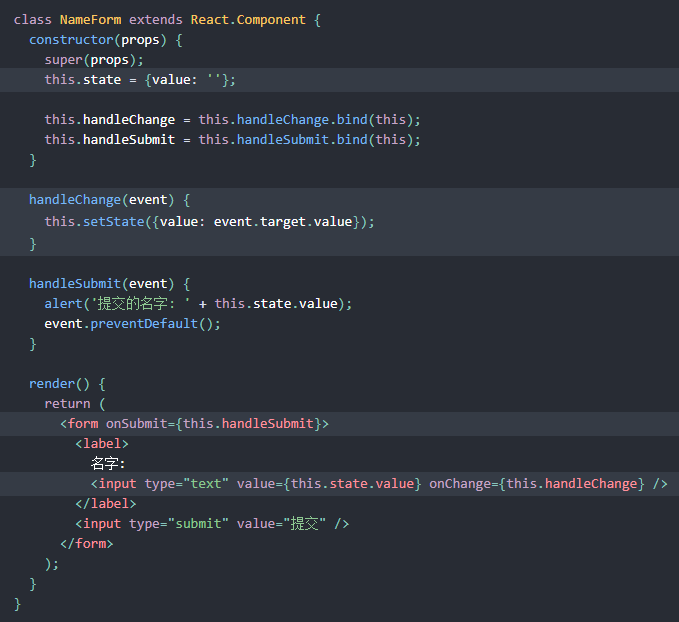
如果列表项目的顺序可能会变化，我们不建议使用索引来用作 key 值，因为这样做会导致性能变差，还可能引起组件状态的问题。

1. 表单

* 受控组件

在 HTML 中，表单元素（如<input>、 <textarea> 和 <select>）通常自己维护 state，并根据用户输入进行更新。而在 React 中，可变状态（mutable state）通常保存在组件的 state 属性中，并且只能通过使用 [setState()](https://zh-hans.reactjs.org/docs/react-component.html#setstate)来更新。

我们可以把两者结合起来，使 React 的 state 成为“唯一数据源”。渲染表单的 React 组件还控制着用户输入过程中表单发生的操作。被 React 以这种方式控制取值的表单输入元素就叫做“**受控组件**”。



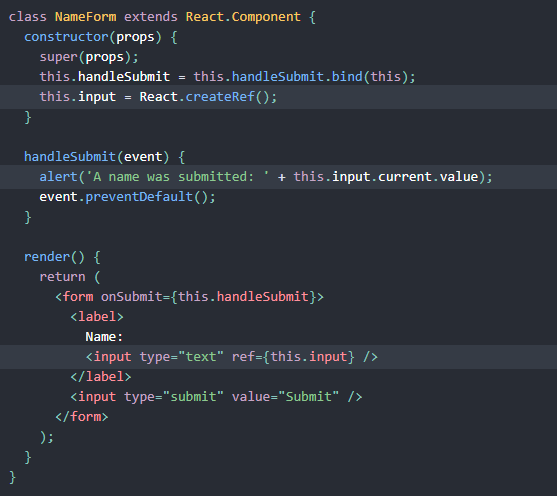
由于在表单元素上设置了 value 属性，因此显示的值将始终为 this.state.value，这使得 React 的 state 成为唯一数据源。

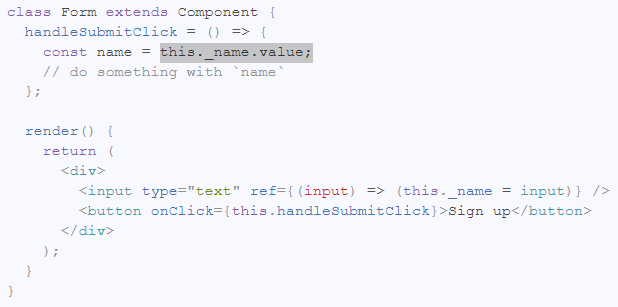
* 非受控组件

有时使用受控组件会很麻烦，因为你需要为数据变化的每种方式都编写事件处理函数，并通过一个 React 组件传递所有的输入 state。当你将之前的代码库转换为 React 或将 React 应用程序与非 React 库集成时，这可能会令人厌烦。在这些情况下，你可能希望使用[**非受控组件**](https://zh-hans.reactjs.org/docs/uncontrolled-components.html), 这是实现输入表单的另一种方式。

在大多数情况下，我们推荐使用 [**受控组件**](https://zh-hans.reactjs.org/docs/forms.html#controlled-components) 来处理表单数据。在一个受控组件中，表单数据是由 React 组件来管理的。另一种替代方案是使用非受控组件，这时表单数据将交由 **DOM** 节点来处理。

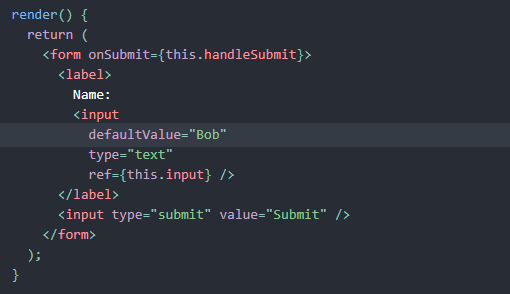
要编写一个非受控组件，而不是为每个状态更新都编写数据处理函数，你可以 [使用 ref](https://zh-hans.reactjs.org/docs/refs-and-the-dom.html) 来从 DOM 节点中获取表单数据。





* 默认值：

在 React 渲染生命周期时，表单元素上的 value 将会覆盖 DOM 节点中的值。在非受控组件中，你经常希望 React 能赋予组件一个初始值，但是不去控制后续的更新。 在这种情况下, 你可以指定一个 defaultValue 属性，而不是 value。在一个组件已经挂载之后去更新 defaultValue 属性的值，不会造成 DOM 上值的任何更新。



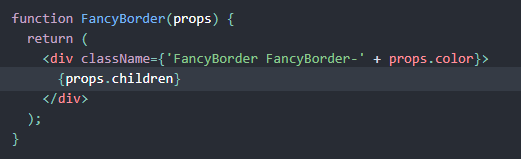
1. 状态提升

通常，多个组件需要反映相同的变化数据，这时我们建议将共享状态提升到最近的共同父组件中去。

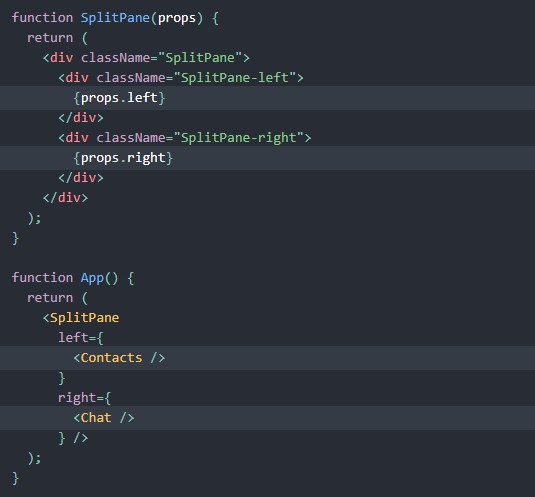
1. 组合（Composition）

有些组件无法提前知晓它们子组件的具体内容。在 Sidebar（侧边栏）和 Dialog（对话框）等展现通用容器（box）的组件中特别容易遇到这种情况。类似于vue中的slot（插槽）

React使用一个特殊的 children prop 来将他们的子组件传递到渲染结果中：



少数情况下，你可能需要在一个组件中预留出几个“洞”。这种情况下，我们可以不使用 children，而是自行约定：将所需内容传入 props，并使用相应的 prop。类似于vue中的具名插槽



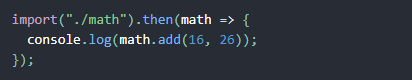
<Contacts /> 和 <Chat /> 之类的 React 元素本质就是对象（object），所以你可以把它们当作 props，像其他数据一样传递。这种方法可能使你想起别的库中“槽”（slot）的概念，但在 React 中没有“槽”这一概念的限制，你可以将任何东西作为 props 进行传递。

1. 无障碍
2. 代码分割

为了避免搞出大体积的代码包，在前期就思考该问题并对代码包进行分割是个不错的选择。 代码分割是由诸如 [Webpack](https://webpack.docschina.org/guides/code-splitting/)，[Rollup](https://rollupjs.org/guide/en/#code-splitting) 和 Browserify（[factor-bundle](https://github.com/browserify/factor-bundle)）这类打包器支持的一项技术，能够创建多个包并在运行时动态加载。

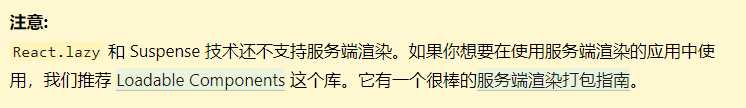
对你的应用进行代码分割能够帮助你“**懒加载**”当前用户所需要的内容，能够显著地提高你的应用性能。尽管并没有减少应用整体的代码体积，但你可以避免加载用户永远不需要的代码，并在初始加载的时候减少所需加载的代码量。

* import()：



当 Webpack 解析到该语法时，会自动进行代码分割。如果你使用 Create React App，该功能已开箱即用，你可以[立刻使用](https://create-react-app.dev/docs/code-splitting/)该特性。[Next.js](https://nextjs.org/docs/advanced-features/dynamic-import) 也已支持该特性而无需进行配置。

* React.lazy()：

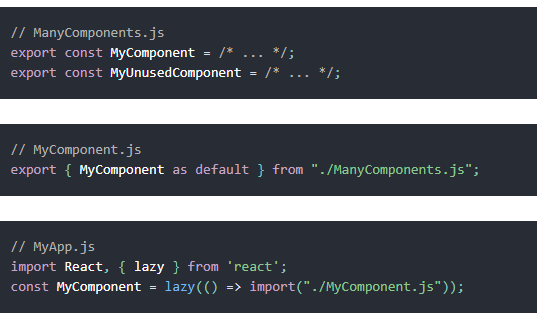


React.lazy 函数能让你像渲染常规组件一样处理动态引入（的组件）。



* 命名导出

React.lazy 目前只支持默认导出（default exports）。如果你想被引入的模块使用命名导出（named exports），你可以创建一个中间模块，来重新导出为默认模块。这能保证 tree shaking 不会出错，并且不必引入不需要的组件。



1. 上下文（Context）：

Context 提供了一个无需为每层组件手动添加 props，就能在组件树间进行数据传递的方法。