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。

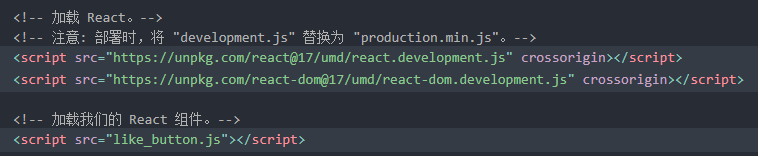
* 跨平台

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。

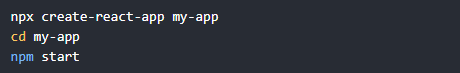


C:\Users\ADMINI~1\AppData\Local\Temp\1646186140(1).png

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。

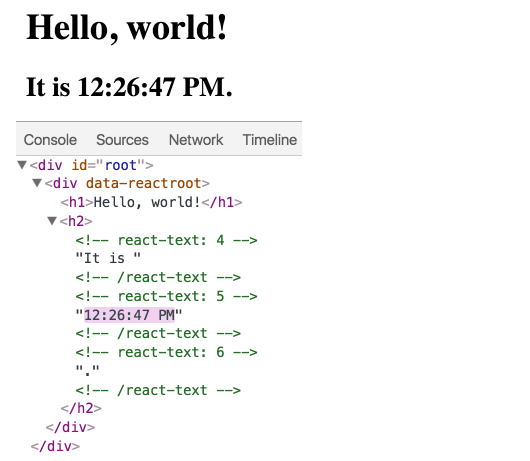
更新 UI 唯一的方式是创建一个全新的元素，并将其传入 [ReactDOM.render()](https://zh-hans.reactjs.org/docs/react-dom.html#render)。

**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。



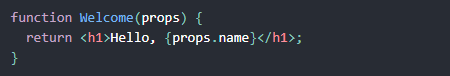
尽管每一秒我们都会新建一个描述整个 UI 树的元素，React DOM 只会更新实际改变了的内容，也就是例子中的文本节点。

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。

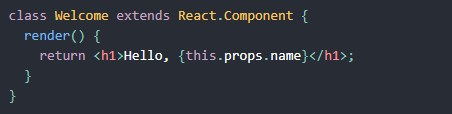
组件，从概念上类似于 JavaScript 函数。它接受任意的入参（即 “props”），并返回用于描述页面展示内容的 React 元素。

* 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。

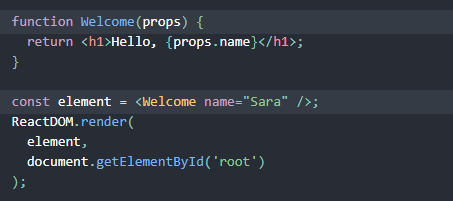
* 类组件

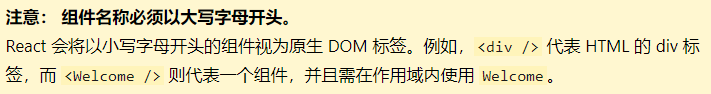


* 渲染组件

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

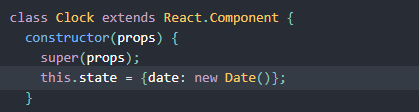
组件无论是使用[函数声明还是通过 class 声明](https://zh-hans.reactjs.org/docs/components-and-props.html#function-and-class-components)，都决不能修改自身的 props。

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。**

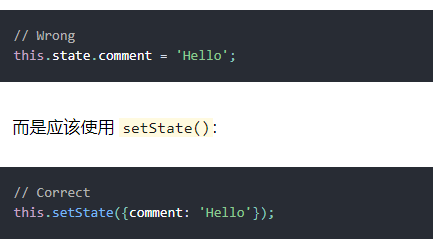
1. State

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

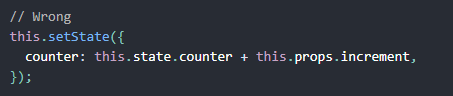


构造函数是唯一可以给 this.state 赋值的地方。

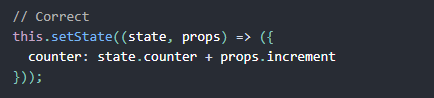
* State更新可能是异步

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

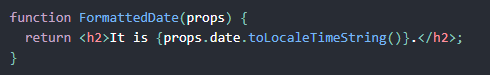
不管是父组件或是子组件都无法知道某个组件是有状态的还是无状态的，并且它们也并不关心它是函数组件还是 class 组件。

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：

**

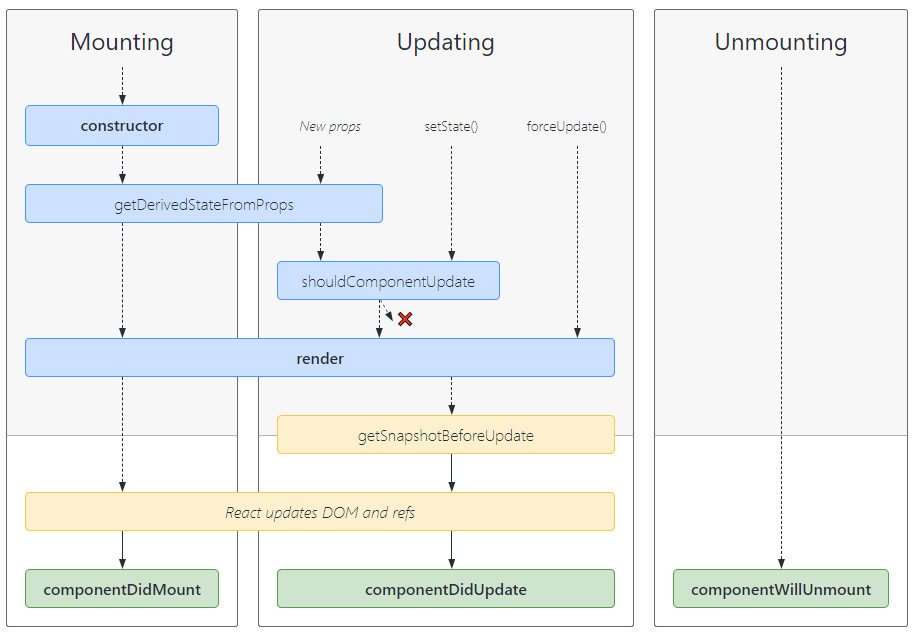
这通常会被叫做“自上而下”或是“单向”的数据流。任何的 state 总是所属于特定的组件，而且从该 state 派生的任何数据或 UI 只能影响树中“低于”它们的组件。

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。

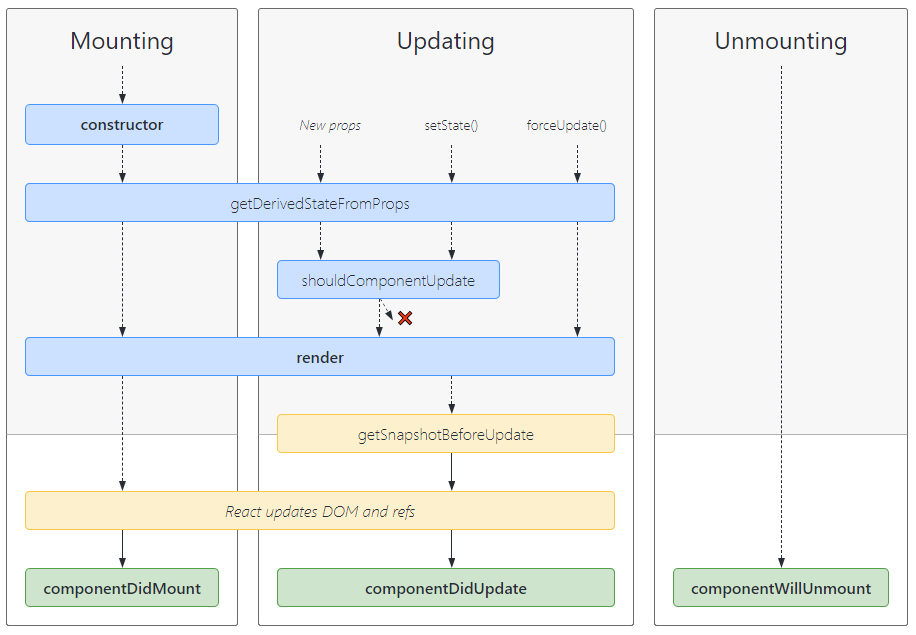
1. 组件生命周期

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。

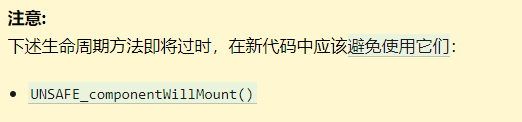
生命周期图谱16.3版本：



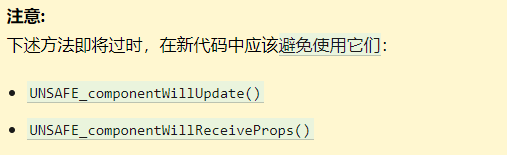
生命周期图谱16.4或以上版本：



* 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)

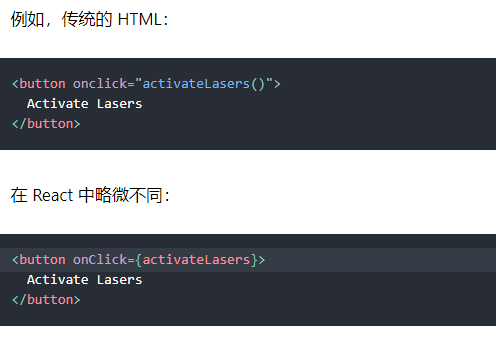


* 卸载时 (当组件从 DOM 中移除时会调用如下方法)：
* [**componentWillUnmount()**](https://zh-hans.reactjs.org/docs/react-component.html#componentwillunmount)
* 错误处理时（当渲染过程，生命周期，或子组件的构造函数中抛出错误）：
* [**static getDerivedStateFromError()**](https://zh-hans.reactjs.org/docs/react-component.html#static-getderivedstatefromerror)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



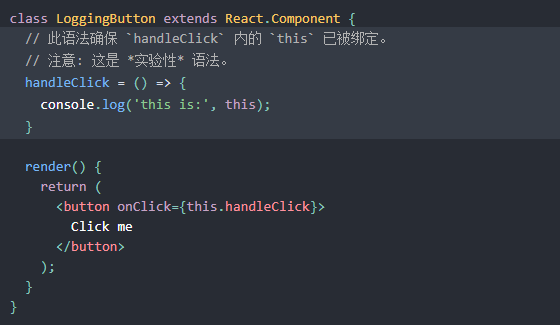
* This

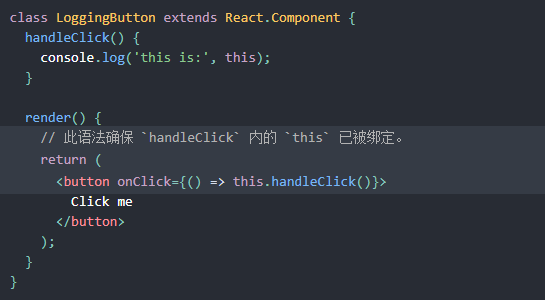
你必须谨慎对待 JSX 回调函数中的 this，在 JavaScript 中，class 的方法默认不会[绑定](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_objects/Function/bind) this。如果你忘记绑定 this.handleClick 并把它传入了 onClick，当你调用这个函数的时候 this 的值为 undefined。



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

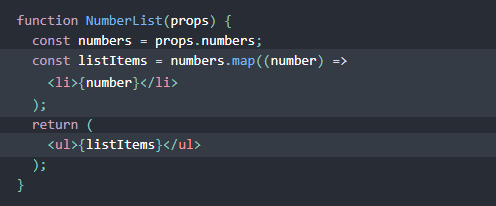
上述两种方式是等价的，分别通过[箭头函数](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions)和 [Function.prototype.bind](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_objects/Function/bind) 来实现

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。

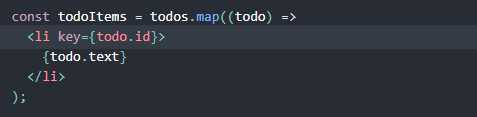
1. 渲染列表



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。

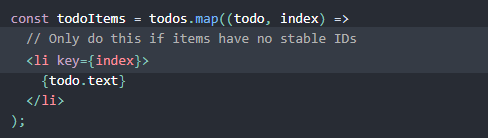
* Key

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



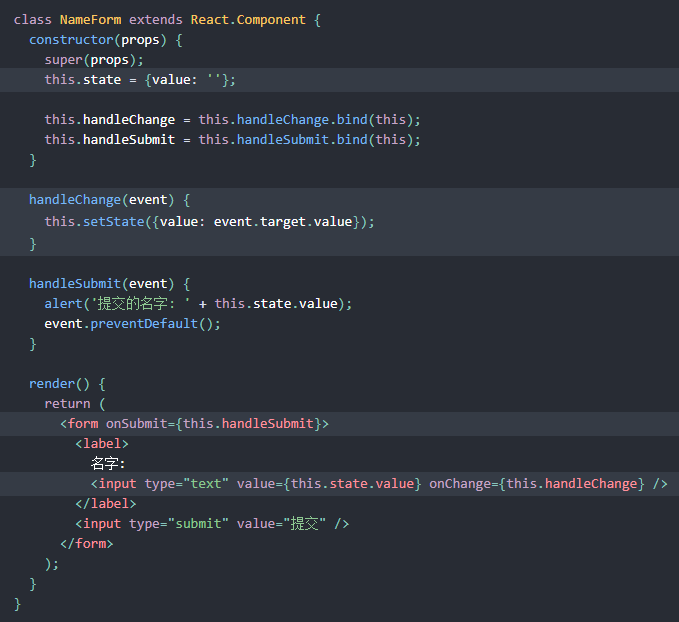
如果列表项目的顺序可能会变化，我们不建议使用索引来用作 key 值，因为这样做会导致性能变差，还可能引起组件状态的问题。

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**”。



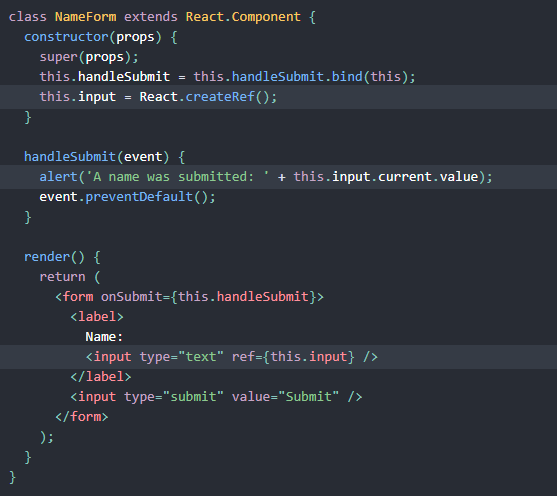
由于在表单元素上设置了 value 属性，因此显示的值将始终为 this.state.value，这使得 React 的 state 成为唯一数据源。

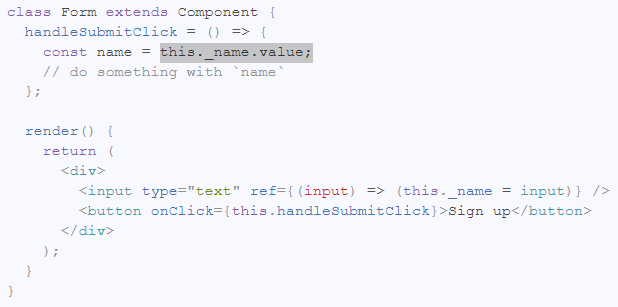
* 非受控组件

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。

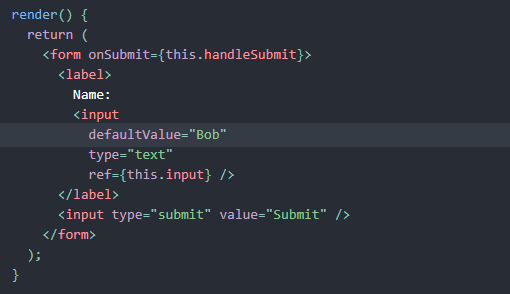
要编写一个非受控组件，而不是为每个状态更新都编写数据处理函数，你可以 [使用 ref](https://zh-hans.reactjs.org/docs/refs-and-the-dom.html) 来从 DOM 节点中获取表单数据。





* 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。



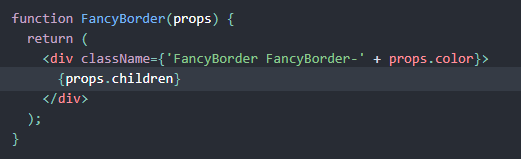
1. 状态提升

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。

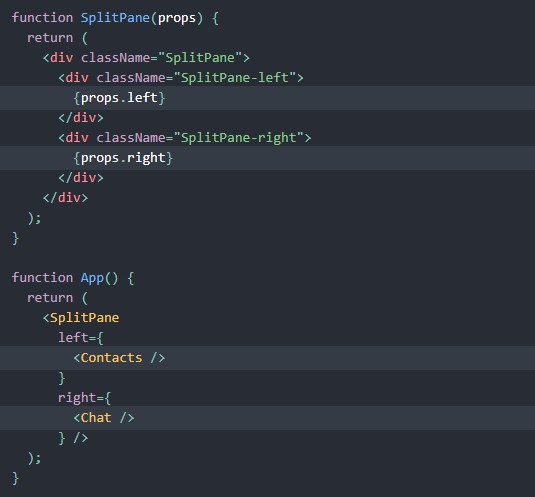
1. 组合（Composition）

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



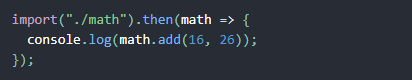
<Contacts /> 和 <Chat /> 之类的 React 元素本质就是对象（object），所以你可以把它们当作 props，像其他数据一样传递。这种方法可能使你想起别的库中“槽”（slot）的概念，但在 React 中没有“槽”这一概念的限制，你可以将任何东西作为 props 进行传递。

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。

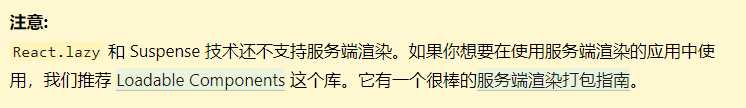
对你的应用进行代码分割能够帮助你“**懒加载**”当前用户所需要的内容，能够显著地提高你的应用性能。尽管并没有减少应用整体的代码体积，但你可以避免加载用户永远不需要的代码，并在初始加载的时候减少所需加载的代码量。

* 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()：

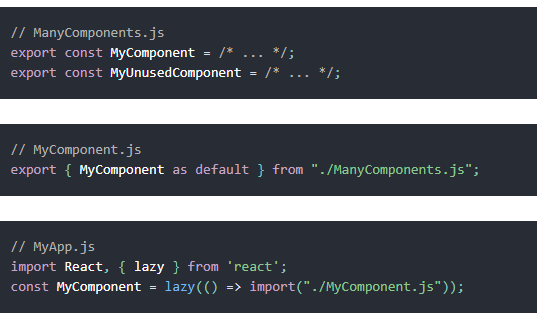


React.lazy 函数能让你像渲染常规组件一样处理动态引入（的组件）。



* 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。

* 使用





* 使用前考虑

Context 主要应用场景在于很多不同层级的组件需要访问同样一些的数据。请谨慎使用，因为这会使得组件的复用性变差。

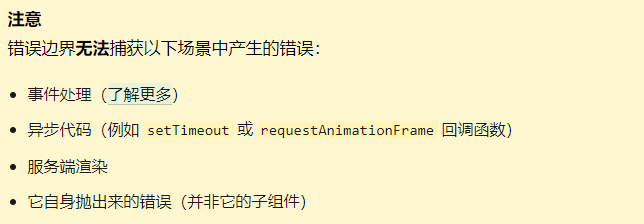
**如果你只是想避免层层传递一些属性，**[**组件组合（component composition）**](https://zh-hans.reactjs.org/docs/composition-vs-inheritance.html)**有时候是一个比 context 更好的解决方案。**

1. 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](https://github.com/facebook/react/issues/4026) [可能无法追踪的](https://github.com/facebook/react/issues/6895) [错误](https://github.com/facebook/react/issues/8579)。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。

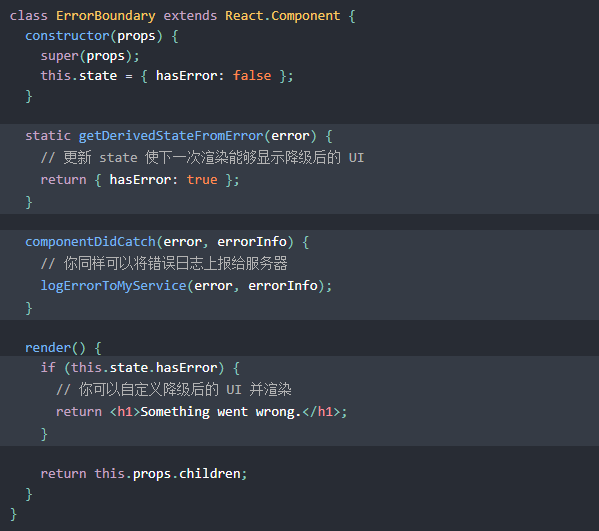
为了解决这个问题，React 16 引入了一个新的概念 —— 错误边界。

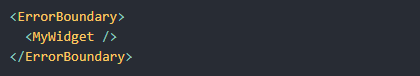
错误边界是一种 React 组件，这种组件**可以捕获发生在其子组件树任何位置的 JavaScript 错误，并打印这些错误，同时展示降级 UI**，而并不会渲染那些发生崩溃的子组件树。错误边界可以捕获发生在整个子组件树的渲染期间、生命周期方法以及构造函数中的错误。



* 使用

如果一个 class 组件中定义了 [**static getDerivedStateFromError()**](https://zh-hans.reactjs.org/docs/react-component.html#static-getderivedstatefromerror) 或[**componentDidCatch()**](https://zh-hans.reactjs.org/docs/react-component.html#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。





错误边界的工作方式类似于 JavaScript 的 **catch** {}，不同的地方在于错误边界只针对 React 组件。只有 class 组件才可以成为错误边界组件。大多数情况下, 你只需要声明一次错误边界组件, 并在整个应用中使用它。

注意**错误边界仅可以捕获其子组件的错误**，它无法捕获其自身的错误。如果一个错误边界无法渲染错误信息，则错误会冒泡至最近的上层错误边界，这也类似于 JavaScript 中 catch {} 的工作机制。

1. **Refs转发**

Ref 转发是一项将 [ref](https://zh-hans.reactjs.org/docs/refs-and-the-dom.html) 自动地通过组件传递到其一子组件的技巧。对于大多数应用中的组件来说，这通常不是必需的。但其对某些组件，尤其是可重用的组件库是很有用的。

* **何时使用Ref**
* 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。

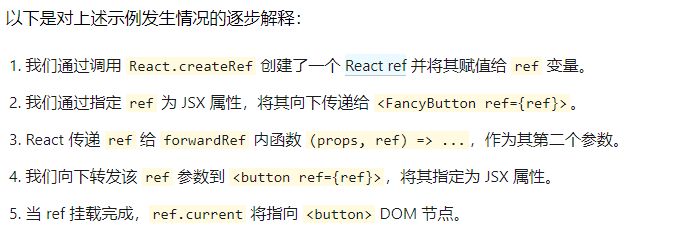
举个例子，避免在 Dialog 组件里暴露 open() 和 close() 方法，最好传递 isOpen 属性。

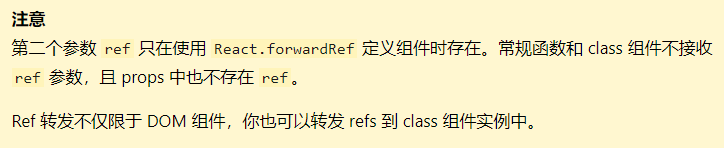
* **转发Ref到DOM组件**

在下面的示例中，FancyButton 使用 **React.forwardRef** 来获取传递给它的 ref，然后转发到它渲染的 DOM button：



这样，使用 FancyButton 的组件可以获取底层 DOM 节点 button 的 ref ，并在必要时访问，就像其直接使用 DOM button 一样。





* **访问Ref**

当 ref 被传递给 render 中的元素时，对该节点的引用可以在 ref 的 current 属性中被访问。

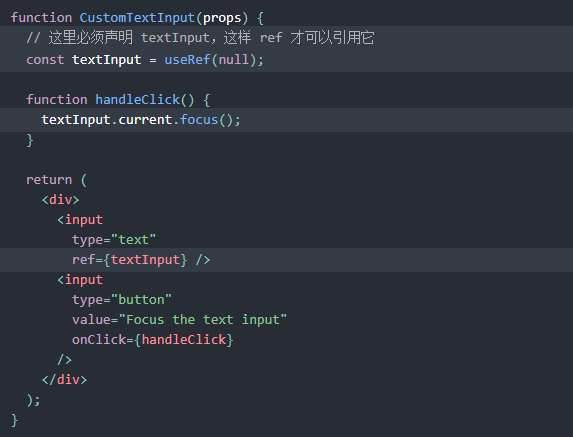
**C:\Users\ADMINI~1\AppData\Local\Temp\1646278616(1).png**

React 会在组件挂载时给 current 属性传入 DOM 元素，并在组件卸载时传入 null 值。ref 会在 componentDidMount 或 componentDidUpdate 生命周期钩子触发前更新。

**ref 的值根据节点的类型而有所不同：**

* 当 ref 属性用于 HTML 元素时，构造函数中使用 React.createRef() 创建的 ref 接收底层 DOM 元素作为其 current 属性
* 当 ref 属性用于自定义 class 组件时，ref 对象接收组件的挂载实例作为其 current 属性
* **你不能在函数组件上使用 ref 属性**，因为他们没有实例
* **函数组件使用Ref**

如果要在函数组件中使用 **ref**，你可以使用 [**forwardRef**](https://zh-hans.reactjs.org/docs/forwarding-refs.html)（可与 [**useImperativeHandle**](https://zh-hans.reactjs.org/docs/hooks-reference.html#useimperativehandle) 结合使用），或者可以将该组件转化为 class 组件。

****

## 组件库维护者的注意事项

## **当你开始在组件库中使用**forwardRef**时，你应当将其视为一个破坏性更改，并发布库的一个新的主版本。** 这是因为你的库可能会有明显不同的行为（例如 refs 被分配给了谁，以及导出了什么类型），并且这样可能会导致依赖旧行为的应用和其他库崩溃。

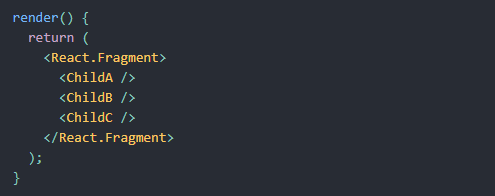
## 出于同样的原因，当 React.forwardRef 存在时有条件地使用它也是不推荐的：它改变了你的库的行为，并在升级 React 自身时破坏用户的应用。

* **在高阶组件中使用Refs转发**

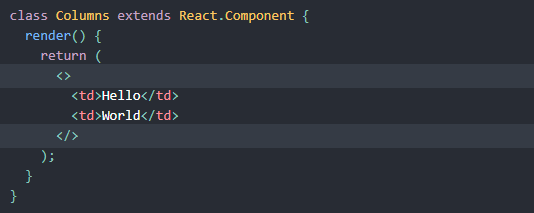


1. Frangments

React 中的一个常见模式是一个组件返回多个元素。Fragments 允许你将子列表分组，而无需向 DOM 添加额外节点。



你可以使用一种新的，且更简短的语法来声明 Fragments。它看起来像空标签：

以上这种模式不支持 key 或属性。

1. 高阶组件（HOC）

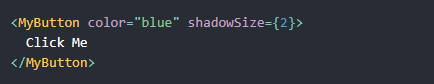
高阶组件（**HOC**）是 React 中用于复用组件逻辑的一种高级技巧。**HOC** 自身不是 React API 的一部分，它是一种基于 React 的组合特性而形成的设计模式。

具体而言，**高阶组件是参数为组件，返回值为新组件的函数。**

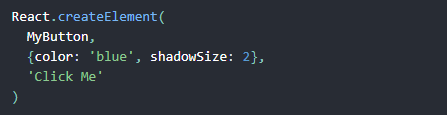
组件是将 props 转换为 UI，而高阶组件是将组件转换为另一个组件。

1. 深入JSX

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



会编译为：



### React 必须在作用域内

由于 JSX 会编译为 React.createElement 调用形式，所以 React 库也必须包含在 JSX 代码作用域内。

如果你不使用 JavaScript 打包工具而是直接通过 <script> 标签加载 React，则必须将 React 挂载到全局变量中。

### 在 JSX 类型中使用点语法

在 JSX 中，你也可以使用点语法来引用一个 React 组件。当你在一个模块中导出许多 React 组件时，这会非常方便。

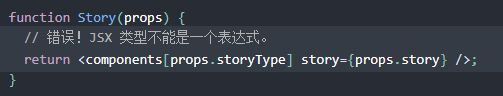


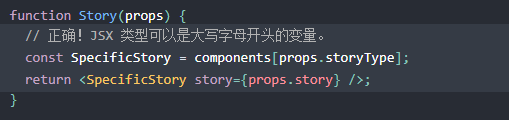
### 组件必须以大写字母开头

以小写字母开头的元素代表一个 HTML **内置组件**，比如 <div> 或者 <span> 会生成相应的字符串 'div' 或者 'span' 传递给 React.createElement（作为参数）。大写字母开头的元素则对应着在 JavaScript 引入或自定义的组件，如 <Foo /> 会编译为 React.createElement(Foo)。

### 在运行时选择类型

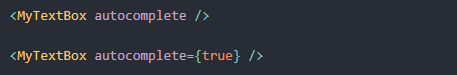
你不能将通用表达式作为 React 元素类型。如果你想通过通用表达式来（动态）决定元素类型，你需要首先将它赋值给大写字母开头的变量。这通常用于根据 prop 来渲染不同组件的情况下:





### Props 默认值为 “True”

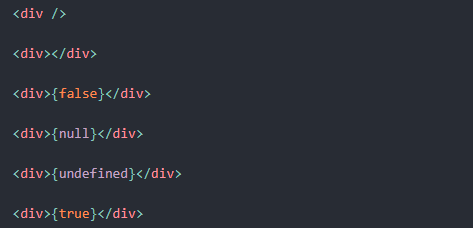
如果你没给 prop 赋值，它的默认值是 true。以下两个 JSX 表达式是等价的：



通常不建议不传递 value 给 prop，因为这可能与 [ES6 对象简写](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Object_initializer#New_notations_in_ECMAScript_2015)混淆，{foo} 是 {foo: foo} 的简写，而不是 {foo: true}。这样实现只是为了保持和 HTML 中标签属性的行为一致。

### 布尔类型、Null 以及 Undefined 将会忽略

false, null, undefined, and true 是合法的子元素。但它们并不会被渲染。以下的 JSX 表达式渲染结果相同：

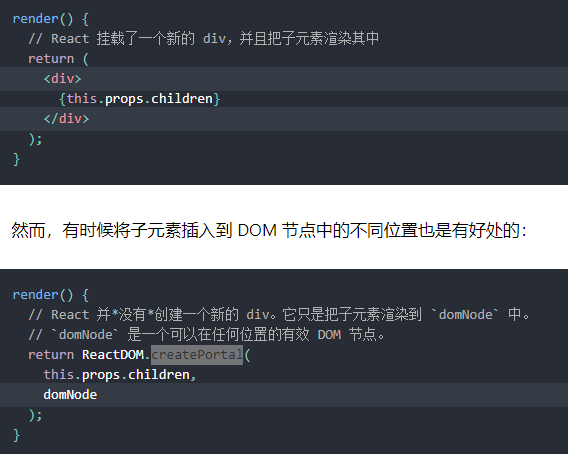


1. Portals

**Portal** 提供了一种将子节点渲染到存在于父组件以外的 **DOM** 节点的优秀的方案。

C:\Users\ADMINI~1\AppData\Local\Temp\1646273357(1).png

第一个参数（child）是任何[可渲染的 React 子元素](https://zh-hans.reactjs.org/docs/react-component.html#render)，例如一个元素，字符串或 fragment。第二个参数（container）是一个 DOM 元素。

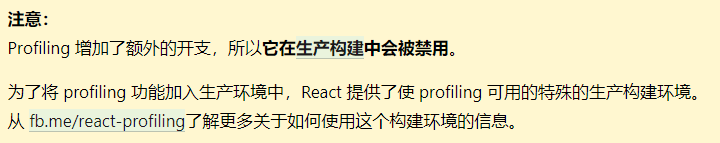


一个 portal 的典型用例是当父组件有 overflow: hidden 或 z-index 样式时，但你需要子组件能够在视觉上“跳出”其容器。例如，对话框、悬浮卡以及提示框：

## 通过 Portal 进行事件冒泡

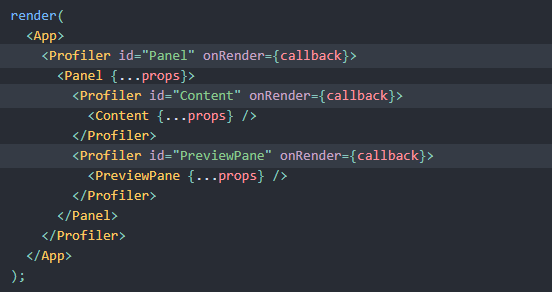
1. **Profiler**

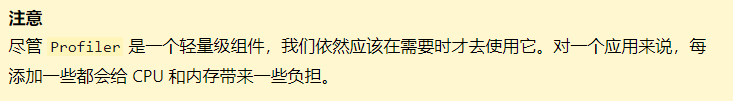
Profiler 测量一个 React 应用多久渲染一次以及渲染一次的“代价”。 它的目的是识别出应用中渲染较慢的部分，或是可以使用[类似 memoization 优化](https://zh-hans.reactjs.org/docs/hooks-faq.html#how-to-memoize-calculations)的部分，并从相关优化中获益。

****

* **用法**

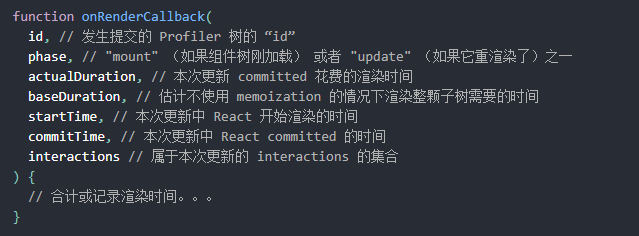
Profiler 能添加在 React 树中的任何地方来测量树中这部分渲染所带来的开销。 它需要两个 prop ：一个是 id(string)，一个是当组件树中的组件“提交”更新的时候被React调用的回调函数 onRender(function)。

****

****

* **onRender回调**

Profiler 需要一个 onRender 函数作为参数。 React 会在 profile 包含的组件树中任何组件 “提交” 一个更新的时候调用这个函数。 它的参数描述了渲染了什么和花费了多久。

****

* **id: string** - 发生提交的 Profiler 树的 id。 如果有多个 profiler，它能用来分辨树的哪一部分发生了“提交”。
* **phase: "mount" | "update"** - 判断是组件树的第一次装载引起的重渲染，还是由 props、state 或是 hooks 改变引起的重渲染。
* **actualDuration: number** - 本次更新在渲染 Profiler 和它的子代上花费的时间。 这个数值表明使用 memoization 之后能表现得多好。（例如 [React.memo](https://zh-hans.reactjs.org/docs/react-api.html#reactmemo)，[useMemo](https://zh-hans.reactjs.org/docs/hooks-reference.html#usememo)，[shouldComponentUpdate](https://zh-hans.reactjs.org/docs/hooks-faq.html#how-do-i-implement-shouldcomponentupdate)）。 理想情况下，由于子代只会因特定的 prop 改变而重渲染，因此这个值应该在第一次装载之后显著下降。
* **baseDuration: number** - 在 Profiler 树中最近一次每一个组件 render 的持续时间。 这个值估计了最差的渲染时间。（例如当它是第一次加载或者组件树没有使用 memoization）。
* **startTime: number** - 本次更新中 React 开始渲染的时间戳。
* **commitTime: number** - 本次更新中 React commit 阶段结束的时间戳。 在一次 commit 中这个值在所有的 profiler 之间是共享的，可以将它们按需分组。
* **interactions: Set** - 当更新被制定时，[“interactions”](https://fb.me/react-interaction-tracing) 的集合会被追踪。（例如当 render 或者 setState 被调用时）。

## Diffing 算法

## 设计动机

## 在某一时间节点调用 React 的 render() 方法，会创建一棵由 React 元素组成的树。在下一次 state 或 props 更新时，相同的 render() 方法会返回一棵不同的树。React 需要基于这两棵树之间的差别来判断如何高效的更新 UI，以保证当前 UI 与最新的树保持同步。

此算法有一些通用的解决方案，即生成将一棵树转换成另一棵树的最小操作次数。然而，即使使用[最优的算法](http://grfia.dlsi.ua.es/ml/algorithms/references/editsurvey_bille.pdf)，该算法的复杂程度仍为 O(n3)，其中 n 是树中元素的数量。

如果在 React 中使用该算法，那么展示 1000 个元素则需要 10 亿次的比较。这个开销实在是太过高昂。于是 React 在以下两个假设的基础之上提出了一套 O(n) 的启发式算法：

1. 两个不同类型的元素会产生出不同的树；
2. 开发者可以通过设置 key 属性，来告知渲染哪些子元素在不同的渲染下可以保存不变；

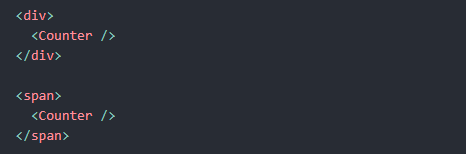
### 对比不同类型的元素

当对比两棵树时，React 首先比较两棵树的根节点。不同类型的根节点元素会有不同的形态。

当根节点为不同类型的元素时，React 会拆卸原有的树并且建立起新的树。举个例子，当一个元素从 <a> 变成 <img>，从 <Article> 变成 <Comment>，或从 <Button> 变成 <div> 都会触发一个完整的重建流程。

当卸载一棵树时，对应的 DOM 节点也会被销毁。组件实例将执行 **componentWillUnmount()**方法。当建立一棵新的树时，对应的 DOM 节点会被创建以及插入到 DOM 中。组件实例将执行 **UNSAFE\_componentWillMount()** 方法，紧接着 **componentDidMount()**方法。所有与之前的树相关联的 state 也会被销毁。

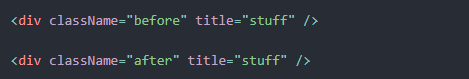
在根节点以下的组件也会被卸载，它们的状态会被销毁。比如，当比对以下更变时：



React 会销毁 Counter 组件并且重新装载一个新的组件。

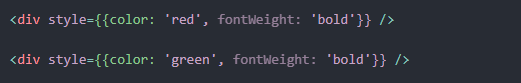
* **对比同一类型的元素**

当对比两个相同类型的 React 元素时，React 会保留 DOM 节点，仅比对及更新有改变的属性。比如：

****

通过对比这两个元素，React 知道只需要修改 DOM 元素上的 className 属性。

当更新 style 属性时，React 仅更新有所更变的属性。比如：

****

通过对比这两个元素，React 知道只需要修改 DOM 元素上的 color 样式，无需修改 fontWeight。

在处理完当前节点之后，React 继续对子节点进行递归。

### 对比同类型的组件元素

当一个组件更新时，组件实例会保持不变，因此可以在不同的渲染时保持 state 一致。React 将更新该组件实例的 props 以保证与最新的元素保持一致，并且调用该实例的 **UNSAFE\_componentWillReceiveProps()**、**UNSAFE\_componentWillUpdate()** 以及 **componentDidUpdate()** 方法。

下一步，调用 render() 方法，diff 算法将在之前的结果以及新的结果中进行递归。

## 对子节点进行递归

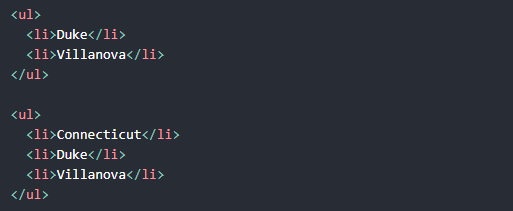
默认情况下，当递归 DOM 节点的子元素时，React 会同时遍历两个子元素的列表；当产生差异时，生成一个 mutation。

**在子元素列表末尾新增元素时，更新开销比较小。比如：**

****

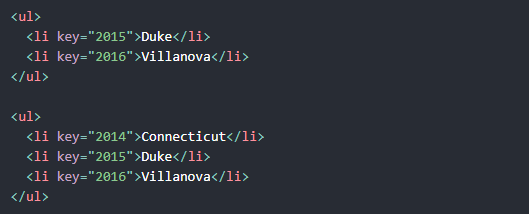
React 会先匹配两个 **<li>first</li>** 对应的树，然后匹配第二个元素 **<li>second</li>**对应的树，最后插入第三个元素的 **<li>third</li>** 树。

**如果只是简单的将新增元素插入到表头，那么更新开销会比较大。比如：**

**** React 并不会意识到应该保留**<li>Duke</li>**和**<li>Villanova</li>**，而是会重建每一个子元素。这种情况会带来性能问题

* **Keys**

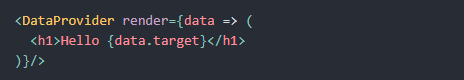
为了解决上述问题，React 引入了 key 属性。当子元素拥有 key 时，React 使用 key 来匹配原有树上的子元素以及最新树上的子元素。以下示例在新增 key 之后，使得树的转换效率得以提高：

****

现在 React 知道只有带着 '2014' key 的元素是新元素，带着 '2015' 以及 '2016' key 的元素仅仅移动了。

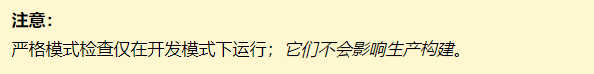
1. **Render Pro**

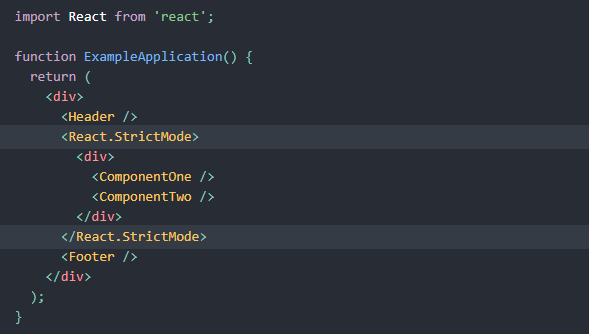
[“render prop”](https://cdb.reacttraining.com/use-a-render-prop-50de598f11ce) 是指一种在 React 组件之间使用一个值为函数的 prop 共享代码的简单技术。

****

1. **严格模式**

StrictMode 是一个用来突出显示应用程序中潜在问题的工具。与 Fragment 一样，StrictMode 不会渲染任何可见的 UI。它为其后代元素触发额外的检查和警告。

****

****

**StrictMode 目前有助于：**

* [识别不安全的生命周期](https://zh-hans.reactjs.org/docs/strict-mode.html#identifying-unsafe-lifecycles)
* [关于使用过时字符串 ref API 的警告](https://zh-hans.reactjs.org/docs/strict-mode.html#warning-about-legacy-string-ref-api-usage)
* [关于使用废弃的 findDOMNode 方法的警告](https://zh-hans.reactjs.org/docs/strict-mode.html#warning-about-deprecated-finddomnode-usage)
* [检测意外的副作用](https://zh-hans.reactjs.org/docs/strict-mode.html#detecting-unexpected-side-effects)
* [检测过时的 context API](https://zh-hans.reactjs.org/docs/strict-mode.html#detecting-legacy-context-api)

1. **HOOK**

Hook 是 React 16.8 的新增特性。它可以让你在不编写 class 的情况下使用 state 以及其他的 React 特性。

Hook 是一些可以让你在函数组件里“钩入” React state 及生命周期等特性的函数。Hook 不能在 class 组件中使用 —— 这使得你不使用 class 也能使用 React。

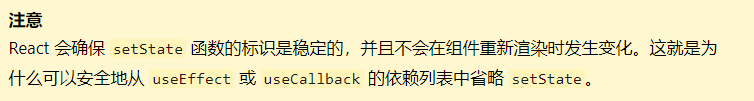
* **使用规则**
* 只能在**函数最外层**调用 Hook。不要在循环、条件判断或者子函数中调用
* 只能在 **React 的函数组件**中调用 Hook。不要在其他 JavaScript 函数中调用。（还有一个地方可以调用 Hook —— 就是自定义的 Hook 中
* **React内置HOOK**
* **useState：**

**C:\Users\ADMINI~1\AppData\Local\Temp\1646291907(1).png**

setState 函数用于更新 state。它接收一个新的 state 值并将组件的一次重新渲染加入队列。

**C:\Users\ADMINI~1\AppData\Local\Temp\1646291974(1).png**

在后续的重新渲染中，useState 返回的第一个值将始终是更新后最新的 state。

****

****

* **useEffect：**

如果你熟悉 React class 的生命周期函数，你可以把 useEffect Hook 看做 componentDidMount，componentDidUpdate 和

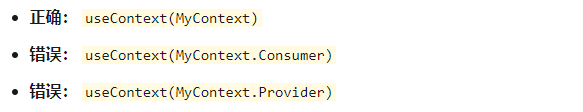
componentWillUnmount 这三个函数的组合。

* **useContext：**

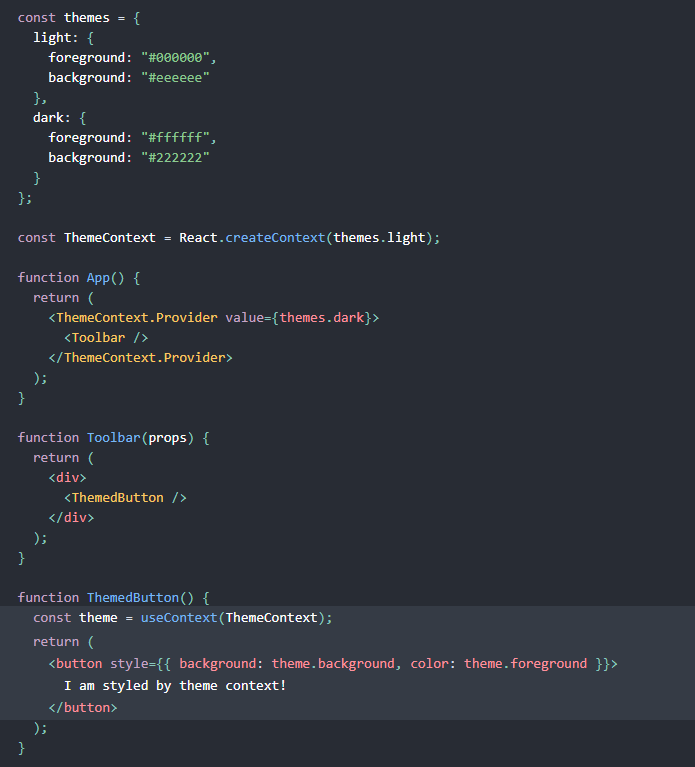
接收一个 context 对象（React.createContext 的返回值）并返回该 context 的当前值。当前的 context 值由上层组件中距离当前组件最近的 <MyContext.Provider> 的 value prop 决定。

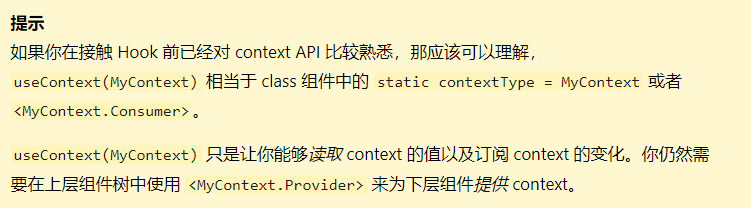
当组件上层最近的 <MyContext.Provider> 更新时，该 Hook 会触发重渲染，并使用最新传递给 MyContext provider 的 context value 值。即使祖先使用 [React.memo](https://zh-hans.reactjs.org/docs/react-api.html#reactmemo) 或 [shouldComponentUpdate](https://zh-hans.reactjs.org/docs/react-component.html#shouldcomponentupdate)，也会在组件本身使用 useContext 时重新渲染。

别忘记 useContext 的参数必须是 context 对象本身：

****

调用了 useContext 的组件总会在 context 值变化时重新渲染。如果重渲染组件的开销较大，你可以 [通过使用 memoization 来优化](https://github.com/facebook/react/issues/15156#issuecomment-474590693)。

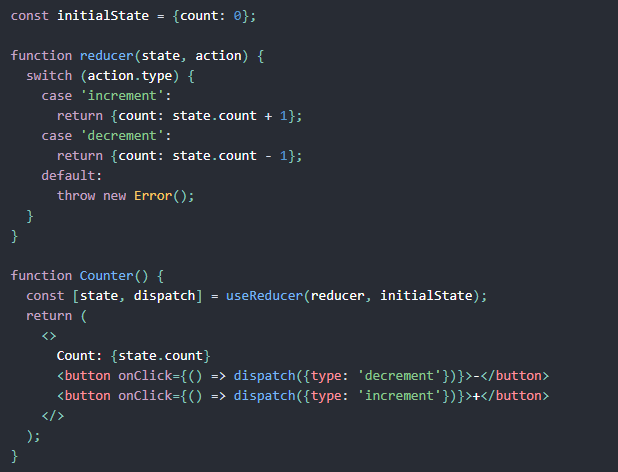


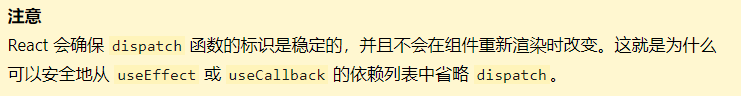
****

* **useReducer：**

[useState](https://zh-hans.reactjs.org/docs/hooks-reference.html#usestate) 的替代方案。它接收一个形如 (state, action) => newState

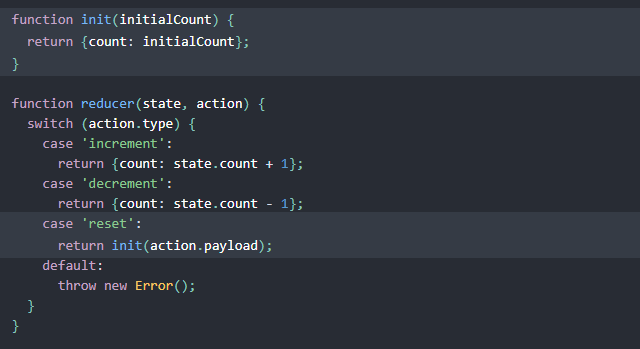
的 reducer，并返回当前的 state 以及与其配套的 dispatch 方法

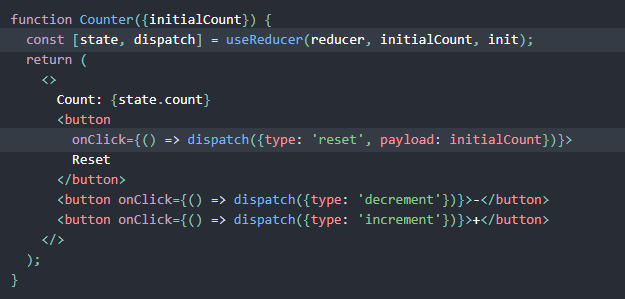
****

****

#### 惰性初始化：

你可以选择惰性地创建初始 state。为此，需要将 init 函数作为 useReducer 的**第三个参数**传入，这样初始 state 将被设置为 **init(initialArg)**。





这么做可以将用于计算 state 的逻辑提取到 reducer 外部，这也为将来对重置 state 的 action 做处理提供了便利：

#### 跳过 dispatch：

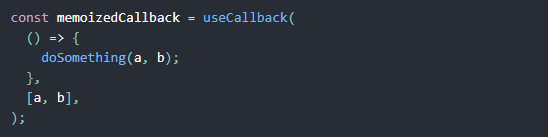
如果 Reducer Hook 的返回值与当前 state 相同，React 将跳过子组件的渲染及副作用的执行。（React 使用 [Object.is 比较算法](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/is#Description) 来比较 state。）

需要注意的是，React 可能仍需要在跳过渲染前再次渲染该组件。不过由于 React 不会对组件树的“深层”节点进行不必要的渲染，所以大可不必担心。如果你在渲染期间执行了高开销的计算，则可以使用 useMemo

来进行优化。

* **useCallback：**

把内联回调函数及依赖项数组作为参数传入 **useCallback**，它将返回该回调函数的 **memoized** 版本，该回调函数仅在某个依赖项**改变时才会更新**。当你把回调函数传递给经过优化的并使用引用相等性去避免非必要渲染（例如 shouldComponentUpdate）的子组件时，它将非常有用。

****

useCallback(fn, deps) 相当于 useMemo(() => fn, deps)。

* **useMemo：**

把“创建”函数和依赖项数组作为参数传入 useMemo，它仅会在某个依赖项改变时才重新计算 memoized 值。这种优化有助于避免在每次渲染时都进行高开销的计算。

**C:\Users\ADMINI~1\AppData\Local\Temp\1646293752(1).png**

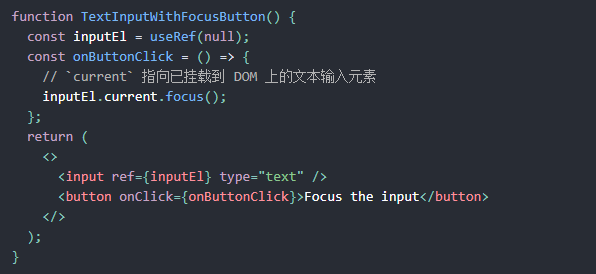
记住，传入 useMemo 的函数会在渲染期间执行。请不要在这个函数内部执行与渲染无关的操作，诸如副作用这类的操作属于 useEffect 的适用范畴，而不是 useMemo。

如果没有提供依赖项数组，useMemo 在每次渲染时都会计算新的值。

**你可以把 useMemo 作为性能优化的手段，但不要把它当成语义上的保证。**将来，React 可能会选择“遗忘”以前的一些 memoized 值，并在下次渲染时重新计算它们，比如为离屏组件释放内存。先编写在没有 useMemo 的情况下也可以执行的代码 —— 之后再在你的代码中添加 useMemo，以达到优化性能的目的。

* **useRef：**

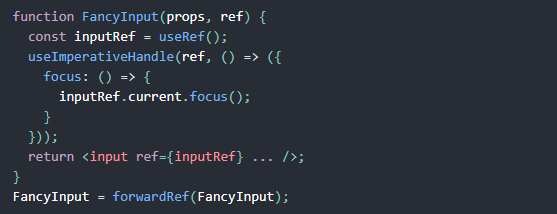
useRef 返回一个可变的 ref 对象，其 .current 属性被初始化为传入的参数（initialValue）。返回的 ref 对象在组件的整个生命周期内持续存在。

****

请记住，当 ref 对象内容发生变化时，useRef 并不会通知你。变更 .current 属性不会引发组件重新渲染。如果想要在 React 绑定或解绑 DOM 节点的 ref 时运行某些代码，则需要使用[回调 ref](https://zh-hans.reactjs.org/docs/hooks-faq.html#how-can-i-measure-a-dom-node) 来实现。

* **useImperativeHandle：**

useImperativeHandle 可以让你在使用 ref 时自定义暴露给父组件的实例值。在大多数情况下，应当避免使用 ref 这样的命令式代码。useImperativeHandle 应当与 [forwardRef](https://zh-hans.reactjs.org/docs/react-api.html#reactforwardref) 一起使用：

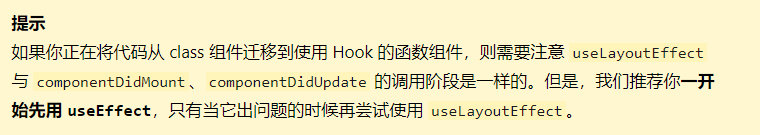
****

在本例中，渲染 <FancyInput ref={inputRef} />的父组件可以调用 inputRef.current.focus()

* **useLayoutEffect：**

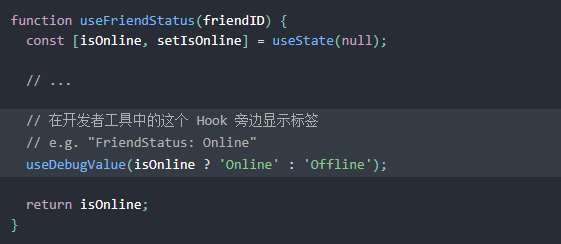
其函数签名与 useEffect 相同，但它会在所有的 DOM 变更之后同步调用 effect。可以使用它来读取 DOM 布局并同步触发重渲染。在浏览器执行绘制之前，useLayoutEffect 内部的更新计划将被同步刷新

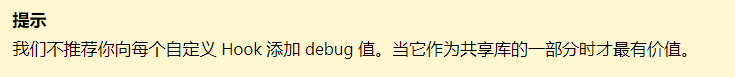
尽可能使用标准的 useEffect 以避免阻塞视觉更新。

****

* **useDebugValue：**

useDebugValue 可用于在 React 开发者工具中显示自定义 hook 的标签。

****

****

#### 延迟格式化 debug 值：

在某些情况下，格式化值的显示可能是一项开销很大的操作。除非需要检查 Hook，否则没有必要这么做。

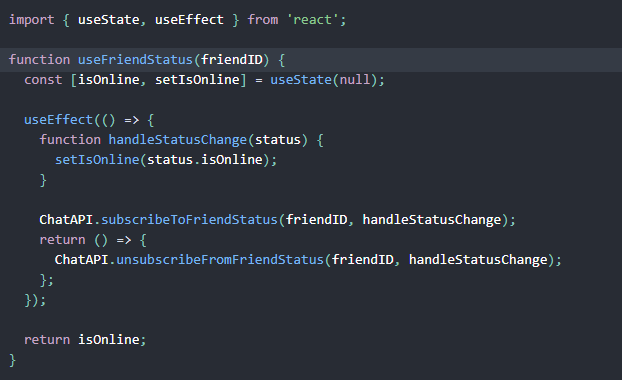
因此，useDebugValue 接受一个格式化函数作为可选的第二个参数。该函数只有在 Hook 被检查时才会被调用。它接受 debug 值作为参数，并且会返回一个格式化的显示值。

C:\Users\ADMINI~1\AppData\Local\Temp\1646294795(1).png

* **自定义HOOK**

通过自定义 Hook，可以将组件逻辑提取到可重用的函数中。

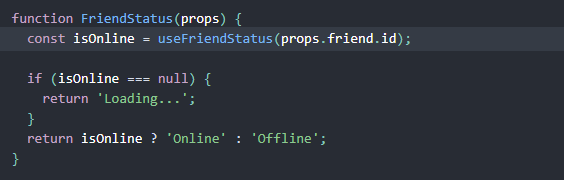
**自定义 Hook 是一个函数，其名称以 “use” 开头，函数内部可以调用其他的 Hook。** 例如，下面的 useFriendStatus 是我们第一个自定义的 Hook:

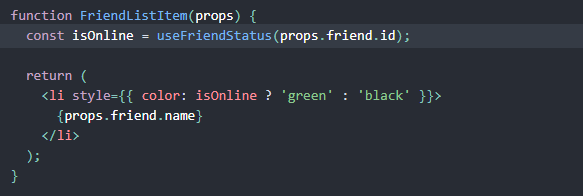
****

* **使用自定义HOOK**

我们一开始的目标是在 FriendStatus 和 FriendListItem 组件中去除重复的逻辑，即：这两个组件都想知道好友是否在线。

现在我们已经把这个逻辑提取到 useFriendStatus 的自定义 Hook 中，然后就可以使用它了：

****

****

**自定义 Hook 必须以 “use” 开头吗？**必须如此。这个约定非常重要。不遵循的话，由于无法判断某个函数是否包含对其内部 Hook 的调用，React 将无法自动检查你的 Hook 是否违反了 [Hook 的规则](https://zh-hans.reactjs.org/docs/hooks-rules.html)。

**在两个组件中使用相同的 Hook 会共享 state 吗？**不会。自定义 Hook 是一种重用状态逻辑的机制(例如设置为订阅并存储当前值)，所以每次使用自定义 Hook 时，其中的所有 state 和副作用都是完全隔离的。

**自定义 Hook 如何获取独立的 state？**每次调用 Hook，它都会获取独立的 state。由于我们直接调用了 useFriendStatus，从 React 的角度来看，我们的组件只是调用了 useState 和 useEffect。 正如我们在[之前章节](https://zh-hans.reactjs.org/docs/hooks-effect.html#tip-use-multiple-effects-to-separate-concerns)中[了解到的](https://zh-hans.reactjs.org/docs/hooks-state.html#tip-using-multiple-state-variables)一样，我们可以在一个组件中多次调用useState 和 useEffect，它们是完全独立的