

REACT Docs - 中文



Table of contents

• Accessibility	5
• Add react /to /a /website	16
• Addons animation	21
• Addons create /fragment	29
• Addons perf	31
• Addons pure /render /mixin	35
• Addons shallow /compare	36
• Addons shallow /renderer	37
• Addons test /utils	39
• Addons two /way /binding /helpers	45
• Addons update	48
• Addons	51
• Cdn links	53
• Code splitting	54
• Codebase overview	59
• Components and /props	63
• Composition vs /inheritance	68
• Concurrent mode /adoption	71
• Concurrent mode /intro	74
• Concurrent mode /patterns	77
• Concurrent mode /reference	96
• Concurrent mode /suspense	100
• Conditional rendering	115
• Context	120
• Create a /new /react /app	126
• Cross origin /errors	128
• Design principles	130
• Error boundaries	135
• Error decoder	139
• Faq ajax	140
• Faq build	143
• Faq functions	144
• Faq internals	151
• Faq state	152

• Faq structure	155
• Faq styling	157
• Faq versioning	158
• Forms	160
• Forwarding refs	166
• Fragments	168
• Getting started	171
• Handling events	174
• Hello world	177
• Higher order /components	179
• Hooks custom	187
• Hooks effect	192
• Hooks faq	202
• Hooks intro	221
• Hooks overview	225
• Hooks reference	231
• Hooks rules	246
• Hooks state	249
• How to /contribute	255
• Implementation notes	259
• Integrating with /other /libraries	275
• Introducing jsx	283
• Jsx in /depth	287
• Legacy context	295
• Legacy event /pooling	300
• Lifting state /up	301
• Lists and /keys	308
• Optimizing performance	313
• Portals	321
• React without /es6	324
• React without /jsx	328
• Reconciliation	329
• Reference dom /elements	333
• Reference events	337
• Reference glossary	345
• Reference javascript /environment /requirements	349

• Reference profiler	350
• Reference pure /render /mixin	353
• Reference react /component	355
• Reference react /dom /client	369
• Reference react /dom /server	371
• Reference react /dom	376
• Reference react	380
• Reference test /renderer	388
• Refs and /the /dom	394
• Release channels	400
• Render props	403
• Rendering elements	409
• State and /lifecycle	411
• Static type /checking	419
• Strict mode	426
• Testing environments	431
• Testing recipes	433
• Testing	444
• Thinking in /react	445
• Typechecking with /proptypes	450
• Uncontrolled components	455
• Web components	457

无障碍辅助功能

为什么我们需要无障碍辅助功能？

网络无障碍辅助功能（Accessibility，也被称为 **a11y**，因为以 A 开头，以 Y 结尾，中间一共 11 个字母）是一种可以帮助所有人获得服务的设计和创造。无障碍辅助功能是使得辅助技术正确解读网页的必要条件。

React 对于创建可访问网站有着全面的支持，而这通常是通过标准 HTML 技术实现的。

标准和指南

WCAG

[网络内容无障碍指南（Web Content Accessibility Guidelines, WCAG）](#) 为开发无障碍网站提供了指南。

下面的 WCAG 检查表提供了一些概览：

- [Wuhcag 提供的 WCAG 检查表（WCAG checklist from Wuhcag）](#)
- [WebAIM 提供的 WCAG 检查表（WCAG checklist from WebAIM）](#)
- [A11Y Project 提供的检查表（Checklist from The A11Y Project）](#)

WAI-ARIA

[网络无障碍倡议 - 无障碍互联网应用（Web Accessibility Initiative - Accessible Rich Internet Applications）](#) 文件包含了创建完全无障碍 JavaScript 部件所需要的技术。

注意：JSX 支持所有 `aria-*` HTML 属性。虽然大多数 React 的 DOM 变量和属性命名都使用驼峰命名（camelCased），但 `aria-*` 应该像其在 HTML 中一样使用带连字符的命名法（也叫诸如 hyphen-cased, kebab-case, lisp-case）。

```
<input
  type="text"
  aria-label={labelText}
  aria-required="true"
  onChange={onchangeHandler}
  value={inputValue}
  name="name"
/>
```

语义化的 HTML

语义化的 HTML 是无障碍辅助功能网络应用的基础。利用多种 HTML 元素来强化您网站中的信息通常可以使您直接获得无障碍辅助功能。

- [MDN 的 HTML 元素参照（MDN HTML elements reference）](#)

有时，语义化的 HTML 会被破坏。比如当在 JSX 中使用 `<div>` 元素来实现 React 代码功能的时候，又或是在使用列表（``, `` 和 `<dl>`）和 HTML `<table>` 时。在这种情况下，我们应该使用 [React Fragments](#) 来组合各个组件。

举个例子,

```
import React, { Fragment } from 'react';

function ListItem({ item }) {
  return (
    <Fragment>
      <dt>{item.term}</dt>
      <dd>{item.description}</dd>
    </Fragment>
  );
}

function Glossary(props) {
  return (
    <dl>
      {props.items.map(item => (
        <ListItem item={item} key={item.id} />
      ))}
    </dl>
  );
}
```

和其他的元素一样，你可以把一系列的对象映射到一个 fragment 的数组中。

```
function Glossary(props) {
  return (
    <dl>
      {props.items.map(item => (
        // Fragments should also have a `key` prop when mapping collections
        <Fragment key={item.id}>
          <dt>{item.term}</dt>
          <dd>{item.description}</dd>
        </Fragment>
      ))}
    </dl>
  );
}
```

当你不需要在 fragment 标签中添加任何 prop 且你的工具支持的时候，你可以使用 短语法：

```
function ListItem({ item }) {
  return (
    <>
      <dt>{item.term}</dt>
      <dd>{item.description}</dd>
    </>
  );
}
```

更多信息请参见 [Fragments 文档](#)。

无障碍表单

标记

所有的 HTML 表单控制，例如 `<input>` 和 `<textarea>`，都需要被标注来实现无障碍辅助功能。我们需要提供屏幕朗读器以解释性标注。

以下资源向我们展示了如何写标注：

- [W3C 向我们展示如何标注元素](#)
- [WebAIM 向我们展示如何标注元素](#)
- [Paciello Group 解释什么是无障碍名称](#)

尽管这些标准 HTML 实践可以被直接用在 React 中，请注意 `for` 在 JSX 中应该被写作 `htmlFor`：

```
<label htmlFor="namedInput">Name:</label>
<input id="namedInput" type="text" name="name"/>
```

在出错时提醒用户

当出现错误时，所有用户都应该知情。下面的链接告诉我们如何给屏幕朗读器设置错误信息：

- [W3C 展示用户推送](#)
- [WebAIM 关于表单校验的文章](#)

控制焦点

确保你的网络应用在即使只拥有键盘的环境下正常运作。

- [WebAIM 讨论使用键盘进行无障碍访问](#)

键盘焦点及焦点轮廓

键盘焦点的定义是：在 DOM 中，当前被选中来接受键盘信息的元素。我们可以在各处看到键盘焦点，它会被焦点轮廓包围，像下面的这个图像一样。



请不要使用 CSS 移除这个轮廓，比如设置 `outline: 0`，除非你将使用其他的方法实现焦点轮廓。

跳过内容机制

为了帮助和提速键盘导航，我们提供了一种机制，可以帮助用户跳过一些导航段落。

跳转链接（Skiplinks），或者说跳转导航链接（Skip Navigation Links）是一种隐藏的导航链接，它只会在使用键盘导航时可见。使用网页内部锚点和一些式样可以很容易地实现它：

- [WebAIM - 跳转导航链接 \(Skip Navigation Links\)](#)

另外，使用地标元素和角色，比如 `<main>` 和 `<aside>`，作为辅助来划分网页的区域可以让用户快速导航至这些部分。

你可以通过下面的链接了解更多如何使用这些元素来增强无障碍辅助功能：

- [无障碍地标](#)

使用程序管理焦点

我们的 React 应用在运行时会持续更改 HTML DOM，有时这将会导致键盘焦点的丢失或者是被设置到了意料之外的元素上。为了修复这类问题，我们需要以编程的方式让键盘聚焦到正确的方向上。比方说，在一个弹窗被关闭的时候，重新设置键盘焦点到弹窗的打开按钮上。

MDN Web 文档关注了这个问题并向我们解释了可以如何搭建可用键盘导航的 JavaScript 部件。

我们可以用 [DOM 元素的 Refs](#) 在 React 中设置焦点。

用以上技术，我们先在一个 class 组件的 JSX 中创建一个元素的 ref：

```
class CustomTextInput extends React.Component {
  constructor(props) {
    super(props);
    // 创建一个 textInput DOM 元素的 ref
    this.textInput = React.createRef();
  }
  render() {
    // 使用 `ref` 回调函数以在实例的一个变量中存储文本输入 DOM 元素
    // (比如, this.textInput)。
    return (
      <input
        type="text"
        ref={this.textInput}
      />
    );
  }
}
```

然后我们就可以在需要时于其他地方把焦点设置在这个组件上：

```
focus() {
  // 使用原始的 DOM API 显式地聚焦在 text input 上
  // 注意：我们通过访问 “current” 来获得 DOM 节点
  this.textInput.current.focus();
}
```

有时，父组件需要把焦点设置在其子组件的一个元素上。我们可以通过在子组件上设置一个特殊的 prop 来对父组件暴露 [DOM refs](#) 从而把父组件的 ref 传向子节点的 DOM 节点。

```
function CustomTextInput(props) {
  return (
    <div>
      <input ref={props.inputRef} />
    </div>
  );
}
```

```

class Parent extends React.Component {
  constructor(props) {
    super(props);
    this.inputElement = React.createRef();
  }
  render() {
    return (
      <CustomTextInput inputRef={this.inputElement} />
    );
  }
}

// 现在你就可以在需要时设置焦点了
this.inputElement.current.focus();

```

当使用 [HOC](#) 来扩展组件时，我们建议使用 React 的 `forwardRef` 函数来向被包裹的组件转发 `ref`。如果第三方的 HOC 不支持转发 `ref`，上面的方法仍可以作为一种备选方案。

[react-aria-modal](#) 提供了一个很好的焦点管理的例子。这是一个少有的完全无障碍的模态窗口的例子。它不仅把初始焦点设置在了取消按钮上（防止键盘用户意外激活成功操作）和把键盘焦点固定在了窗口之内，关闭窗口时它也会把键盘焦点重置到打开窗口的那个元素上。

注意：

虽然这是一个非常重要的无障碍辅助功能，但它也是一种应该谨慎使用的技术。我们应该在受到干扰时使用它来修复键盘焦点，而不是试图预测用户想要如何使用应用程序。

鼠标和指针事件

确保任何可以使用鼠标和指针完成的功能也可以只通过键盘完成。只依靠指针会产生很多使键盘用户无法使用你的应用的情况。

为了说明这一点，让我们看一下由点击事件引起的破坏无障碍访问的典型示例：外部点击模式，用户可以通过点击元素以外的地方来关闭已打开的弹出框。

Select an option

Load the option

Remove the option

通常实现这个功能的方法是在 `window` 对象中附上一个 `click` 事件以关闭弹窗：

```

class OuterClickExample extends React.Component {
  constructor(props) {
    super(props);

    this.state = { isOpen: false };
    this.toggleContainer = React.createRef();

    this.onClickHandler = this.onClickHandler.bind(this);
    this.onClickOutsideHandler = this.onClickOutsideHandler.bind(this);
  }

  componentDidMount() {
    window.addEventListener('click', this.onClickOutsideHandler);
  }

  componentWillUnmount() {
    window.removeEventListener('click', this.onClickOutsideHandler);
  }

  onClickHandler() {
    this.setState(currentState => ({
      isOpen: !currentState.isOpen
    }));
  }

  onClickOutsideHandler(event) {
    if (this.state.isOpen && !this.toggleContainer.current.contains(event.target))
    {
      this.setState({ isOpen: false });
    }
  }

  render() {
    return (
      <div ref={this.toggleContainer}>
        <button onClick={this.onClickHandler}>Select an option</button>
        {this.state.isOpen && (
          <ul>
            <li>Option 1</li>
            <li>Option 2</li>
            <li>Option 3</li>
          </ul>
        )}
      </div>
    );
  }
}

```

当用户使用指针设备，比如鼠标时，这样做没有问题。但是当只使用键盘时，因为 `window` 对象不会接受到 `click` 事件，用户将无法使用 `tab` 切换到下一个元素。这样会导致用户无法使用你应用中的一些内容，导致不完整的用户体验。

Select an option

Load the option

Remove the option

使用正确的事件触发器，比如 `onBlur` 和 `onFocus`，同样可以达成这项功能：

```
class BlurExample extends React.Component {
  constructor(props) {
    super(props);

    this.state = { isOpen: false };
    this.timeOutId = null;

    this.onClickHandler = this.onClickHandler.bind(this);
    this.onBlurHandler = this.onBlurHandler.bind(this);
    this.onFocusHandler = this.onFocusHandler.bind(this);
  }

  onClickHandler() {
    this.setState(currentState => ({
      isOpen: !currentState.isOpen
    }));
  }

  // 我们在下一个时间点使用 setTimeout 关闭弹窗。
  // 这是必要的，因为失去焦点事件会在新的焦点事件前被触发，
  // 我们需要通过这个步骤确认这个元素的一个子节点
  // 是否得到了焦点。
  onBlurHandler() {
    this.timeOutId = setTimeout(() => {
      this.setState({
        isOpen: false
      });
    }, );
  }

  // 如果一个子节点获得了焦点，不要关闭弹窗。
  onFocusHandler() {
    clearTimeout(this.timeOutId);
  }

  render() {
    // React 通过把失去焦点和获得焦点事件传输给父节点
    // 来帮助我们。
    return (
      <div onBlur={this.onBlurHandler}
            onFocus={this.onFocusHandler}>
        <button onClick={this.onClickHandler}
                  aria-haspopup="true"
                  aria-expanded={this.state.isOpen}>
          Select an option
        </button>
      </div>
    );
  }
}
```

```

        </button>
      {this.state.isOpen && (
        <ul>
          <li>Option 1</li>
          <li>Option 2</li>
          <li>Option 3</li>
        </ul>
      )}
    );
}
}

```

以上代码使得键盘和鼠标用户都可以使用我们的功能。请注意我们添加了 `aria-*` props 以服务屏幕朗读器用户。作为一个简单的例子，我们没有实现使用 `方向键` 来与弹窗互动。

Select an option

Load the option

Remove the option

这只是众多只依赖于鼠标和指针的程序破坏键盘用户的例子之一。始终使用键盘测试会让你迅速发现这些问题，你可以使用适用于键盘的事件处理器来修复这些问题。

更复杂的部件

一个更加复杂的用户体验并不意味着更加难以访问。通过尽可能接近 HTML 编程，无障碍访问会变得更加容易，即使最复杂的部件也可以实现无障碍访问。

这里我们需要了解 [ARIA Roles](#) 和 [ARIA States and Properties](#) 的知识。其中有包含了多种 HTML 属性的工具箱，这些 HTML 属性被 JSX 完全支持并且可以帮助我们搭建完全无障碍，功能强大的 React 组件。

每一种部件都有一种特定的设计模式，并且用户和用户代理都会期待使用相似的方法使用它：

- [WAI-ARIA 创作实践 —— 设计模式和部件](#)
- [Heydon Pickering - ARIA Examples](#)
- [包容性组件 \(Inclusive Components\)](#)

其他考虑因素

设置语言

为了使屏幕朗读器可以使用正确的语音设置，请在网页上设置正确的人类语言：

- [WebAIM —— 文档语言](#)

设置文档标题

为了确保用户可以了解当前网页的内容，我们需要把文档的 `<title>` 设置为可以正确描述当前页面的文字。

- [WCAG —— 理解文档标题的要求](#)

在 React 中，我们可以使用 [React 文档标题组件（React Document Title Component）](#) 来设置标题。

色彩对比度

为了尽可能让视力障碍用户可以阅读你网站上的所有可读文字，请确保你的文字都有足够的色彩对比度。

- [WCAG —— 理解色彩对比度要求](#)
- [有关色彩对比度的一切以及为何你应该重新考虑它](#)
- [A11yProject —— 什么是色彩对比度](#)

手工计算你网站上所有恰当的色彩组合会是乏味的。所以，作为代替，你可以[使用 Colorable 来计算出一个完全无障碍的调色板](#)。

下面介绍的 aXe 和 WAVE 都支持色彩对比度测试并会报告对比度错误。

如果你想扩展对比度测试能力，可以使用以下工具：

- [WebAIM —— 色彩对比度检验工具](#)
- [The Paciello Group —— 色彩对比度分析工具](#)

开发及测试

我们可以利用很多工具来帮助我们创建无障碍的网络应用。

键盘

最最简单也是最最重要的检测是确保你的整个网站都可以被只使用键盘的用户使用和访问。你可以通过如下步骤进行检测：

1. 断开鼠标
2. 使用 `Tab` 和 `Shift+Tab` 来浏览。
3. 使用 `Enter` 来激活元素。
4. 当需要时，使用键盘上的方向键来和某些元素互动，比如菜单和下拉选项。

开发辅助

我们可以直接在 JSX 代码中检测一些无障碍复制功能。通常支持 JSX 的 IDE 会针对 ARIA roles, states 和 properties 提供智能检测。我们也可以使用以下工具：

eslint-plugin-jsx-a11y

ESLint 中的 [eslint-plugin-jsx-a11y](#) 插件为你的 JSX 中的无障碍问题提供了 AST 的语法检测反馈。许多 IDE 都允许你把这些发现直接集成到代码分析和源文件窗口中。

Create React App 中使用了这个插件中的一部分规则。如果你想启用更多的无障碍规则，你可以在项目的根目录中创建一个有如下内容的 `.eslintrc` 文件：

```
{
  "extends": ["react-app", "plugin:jsx-a11y/recommended"],
  "plugins": ["jsx-a11y"]
}
```

在浏览器中测试无障碍辅助功能

已有很多工具可以在您的浏览器内进行网页的无障碍性验证。因为它们只能检测你 HTML 的技术无障碍性，所以请将它们与这里提到的无障碍检测工具一起使用。

aXe,aXe-core 以及 react-axe

Deque 系统提供了 `aXe-core` 以对你的应用进行自动及端至端无障碍性测试。这个组件包含了对 Selenium 的集成。

[无障碍访问引擎 \(The Accessibility Engine\)](#)，简称 `aXe`，是一个基于 `aXe-core` 的无障碍访问性检测器。

在开发和 debug 时，你也可以使用 `@axe-core/react` 组件直接把无障碍访问的发现显示在控制台中。

WebAIM WAVE

[网络无障碍性评估工具 \(Web Accessibility Evaluation Tool\)](#) 也是一个无障碍辅助的浏览器插件。

无障碍辅助功能检测器和无障碍辅助功能树

[无障碍辅助功能树](#)是 DOM 树的一个子集，其中包含了所有 DOM 元素中应该被暴露给无障碍辅助技术（比如屏幕朗读器）的无障碍辅助对象。

在一些浏览器中，我们可以在无障碍辅助功能树中轻松的看到每个元素的无障碍辅助功能信息：

- 在 Firefox 中使用无障碍辅助功能检测器
- 在 Chrome 中激活无障碍辅助功能检测器
- 在 OS X Safari 中使用无障碍辅助功能检测器

屏幕朗读器

使用屏幕朗读器测试应该是你无障碍辅助功能测试的一部分。

请注意，浏览器与屏幕朗读器的组合很重要。我们建议在最适用于你的屏幕朗读器的浏览器中测试你的应用。

常用屏幕朗读器

火狐中的 NVDA

[NonVisual Desktop Access](#)，简称 NVDA，是一个被广泛使用的 Windows 开源屏幕朗读器。

想要了解怎么样最好的使用 NVDA，请参考下面的指南：

- [WebAIM —— 使用 NVDA 来评估网络的可无障碍访问性](#)
- [Deque —— NVDA 键盘快捷键](#)

Safari 中的 VoiceOver

VoiceOver 是苹果设备的自带屏幕朗读器。

想要了解如何激活以及使用 VoiceOver, 请参考下面的指南 :

- [WebAIM —— 使用 VoiceOver 来评估网络的可无障碍访问性](#)
- [Deque —— OS X 中的 VoiceOver 键盘快捷键](#)
- [Deque —— iOS 中的 VoiceOver 快捷键](#)

Internet Explorer 中的 JAWS

[Job Access With Speech](#)又称 JAWS, 是一个常用的 Windows 屏幕朗读器。

想要了解如何最好的使用 VoiceOver, 请参考下面的指南 :

- [WebAIM —— 使用 JAWS 来评估网络的可无障碍访问性](#)
- [Deque —— JAWS 键盘快捷键](#)

其他屏幕朗读器

Google Chrome 中的 ChromeVox

[ChromeVox](#)是 Chromebook 的内置屏幕朗读器, 同时也是 Google Chrome 中的一个插件。

想要了解如何最好的使用 ChromeVox, 请参考下面的指南 :

- [Google Chromebook 帮助 —— 使用内置屏幕朗读器](#)
- [ChromeVox 经典键盘快捷键参考](#)

在网站中添加 React

根据需要选择性地使用 React。

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。

大多数网站不是、也不需要是单页应用程序。通过仅仅几行代码并且无需使用构建工具，试试在你的网站的一小部分中使用 React。然后，你可以逐步扩展它的存在，或只将其涵盖在少数动态部件中。

- [一分钟用上 React](#)
- 可选：使用 React 和 JSX (不需要打包工具！)

一分钟用上 React

在本小节中，我们会展示如何将 React 组件添加到现有的 HTML 页面中。你可以基于自己现有的网站，或创建一个空的 HTML 文件来练习。

不会涉及复杂的工具或安装需求 —— 完成这一节的内容，你只需要连接到网络，以及一分钟的时间。

可选：[下载完整示例 \(2KB 压缩包\)](#)

步骤 1：添加一个 DOM 容器到 HTML

首先，打开你想要编辑的 HTML 页面。添加一个空的 `<div>` 标签作为标记你想要用 React 显示内容的位置。例如：

```
<!-- ... 其它 HTML ... -->
<div id="like_button_container"></div>
<!-- ... 其它 HTML ... -->
```

我们给这个 `<div>` 加上唯一的 `id` HTML 属性。这将允许我们稍后用 JavaScript 代码找到它，并在其中显示一个 React 组件。

提示

你可以像这样在 `<body>` 标签内的任意位置放置一个“容器”`<div>`。根据需要，你可以在一个页面上放置多个独立的 DOM 容器。它们通常是空标签 —— React 会替换 DOM 容器内的任何已有内容。

步骤 2：添加 Script 标签

接下来，在 `</body>` 结束标签之前，向 HTML 页面中添加三个 `<script>` 标签：

```
<!-- ... 其它 HTML ... -->
<!-- 加载 React。-->
```

```

<!-- 注意：部署时，将 "development.js" 替换为 "production.min.js"。-->
<script src="https://unpkg.com/react@18/umd/react.development.js" crossorigin>
</script>
<script src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"
crossorigin></script>

<!-- 加载我们的 React 组件。-->
<script src="like_button.js"></script>

</body>

```

前两个标签加载 React。第三个将加载你的组件代码。

步骤 3: 创建一个 React 组件

在 HTML 页面文件的同级目录下创建一个名为 `like_button.js` 的文件。

查看[这段模板代码](#)并把它粘贴到你创建的文件中。

提示

这段代码定义了一个名为 `LikeButton` 的 React 组件。如果你还不明白这段代码的意思，不用担心 —— 我们将在后续的[入门教程](#)和[核心概念](#)中介绍 React 的构建块。目前，我们先只做到显示！

在 `like_button.js` 的底部，在[模板代码](#)之后，加入以下三行代码。

```

// ... 此前粘贴的代码 ...

const domContainer = document.querySelector('#like_button_container');
const root = ReactDOM.createRoot(domContainer);
root.render();

```

这三行代码会找到我们在步骤 1 中添加在 HTML 里的 `<div>`，用它创建一个 React 应用，然后在其内部显示我们“Like”按钮的 React 组件。

就是这么简单！

没有第四步了。你刚刚已经将第一个 React 组件添加到你的网站中。

查看后续小节，以便查看关于集成 React 的更多提示。

[查看完整的示例源码](#)

[下载完整示例（2KB 压缩包）](#)

提示：重用一个组件

通常，你可能希望在 HTML 页面的多个位置展示 React 组件。下面是一个示例，它显示了三次“Like”按钮，并向各自传入了一些数据：

[查看完整的示例源码](#)

[下载完整示例 \(2KB 压缩包\)](#)

注意

当页面中以 React 驱动的不同部分是相互独立的时候，这种策略通常非常有用。在 React 代码中，使用[组件组合 \(component composition\)](#) 来实现会更容易。

提示：为生产环境压缩 JavaScript 代码

在将你的网站部署到生产环境之前，要注意未经压缩的 JavaScript 可能会显著降低用户的访问速度。

如果你已经压缩了应用代码，如果你确保已部署的 HTML 加载了以 `production.min.js` 结尾的 React 版本，那么你的网站是生产就绪 (**production-ready**) 的：

```
<script src="https://unpkg.com/react@18/umd/react.production.min.js" crossorigin>
</script>
<script src="https://unpkg.com/react-dom@18/umd/react-dom.production.min.js"
crossorigin></script>
```

如果你没有一个代码压缩的步骤，[这有一个配置它的方式](#)。

可选：使用 React 和 JSX

在上面的示例中，我们只依赖了浏览器原生支持的特性。这就是为什么我们使用了 JavaScript 函数调用来告诉 React 要显示什么：

```
const e = React.createElement;

// 显示一个 "Like" <button>
return e(
  'button',
  { onClick: () => this.setState({ liked: true }) },
  'Like'
);
```

然而，React 还提供了一种使用 **JSX** 编写界面的方式：

```
// 显示一个 "Like" <button>
return (
  <button onClick={() => this.setState({ liked: true })}>
    Like
  </button>
);
```

这两段代码是等价的。虽然 **JSX 完全是可选的**，但是多数人觉得这样编写 UI 代码更方便 —— 无论是使用 React 还是其它库。

你可以通过[在线编译器](#)试用 JSX。

快速尝试 JSX

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

```
<script src="https://unpkg.com/babel-standalone@6/babel.min.js"></script>
```

现在，你可以在任何 `<script>` 标签内使用 JSX，方法是在为其添加 `type="text/babel"` 属性。这是一个使用了 [JSX 的 HTML 文件的例子](#)，你可以下载并尝试使用。

这种方式适合于学习和创建简单的示例。然而，它会使你的网站变慢，并且不适用于生产环境。当你准备好更进一步时，删除你添加的这个新的 `<script>` 标签以及 `type="text/babel"` 属性。取而代之的，在下一小节中，你将设置一个 JSX 预处理器来自动转换所有 `<script>` 标签的内容。

将 JSX 添加到项目

将 JSX 添加到项目中并不需要诸如打包工具或开发服务器那样复杂的工具。本质上，添加 JSX 就像添加 CSS 预处理器一样。唯一的要求是你在计算机上安装了 [Node.js](#)。

在终端上跳转到你的项目文件夹，然后粘贴这两个命令：

1. 步骤 1：执行 `npm init -y` (如果失败，[这是修复办法](#))
2. 步骤 2：执行 `npm install babel-cli@6 babel-preset-react-app@3`

提示

我们在这里使用 `npm` 只是用来安装 JSX 预处理器，之后你不再需要它。React 和应用程序代码都可以继续使用 `<script>` 标签而不做任何更改。

恭喜！你刚刚为你的项目加入了一个**生产就绪 (production-ready)** 的 JSX 配置环境。

运行 JSX 预处理器

创建一个名为 `src` 的文件夹并执行这个终端命令：

```
npx babel --watch src --out-dir . --presets react-app/prod
```

注意：

`npx` 不是拼写错误 —— 它是 `npm 5.2+` 附带的 package 运行工具。

如果你看到一个错误消息显示为：“You have mistakenly installed the `babel` package”，你可能错过了[上一步](#)。在同一个文件夹中执行它，然后重试。

不要等待它运行结束 —— 这个命令启动了一个对 JSX 的自动监听器。

如果此时你用这段 [JSX 入门代码](#) 创建一个 `src/like_button.js` 文件，监听器会创建一个预处理过的 `like_button.js` 文件，它包含了适用于浏览器的普通 JavaScript 代码。当你编辑带有 JSX 的源文件时，转换过程将自动重新执行。

这样，在旧浏览器上也能够使用现代 JavaScript 的语法特性，比如 `class`。我们刚才使用的工具叫 Babel，你可以从[它的文档](#)中了解更多。

如果你认为你已经习惯了构建工具，并希望它们能为你做更多事，[下一章节](#)描述了一些最流行和易上手的工具链。如果不使用构建工具——直接以 `scripts` 标签的方式也可以很好地完成工作！

Animation Add-Ons

Note:

`ReactTransitionGroup` and `ReactCSSTransitionGroup` have been moved to the `react-transition-group` package that is maintained by the community. Its 1.x branch is completely API-compatible with the existing addons. Please file bugs and feature requests in the [new repository](#).

The `ReactTransitionGroup` add-on component is a low-level API for animation, and `ReactCSSTransitionGroup` is an add-on component for easily implementing basic CSS animations and transitions.

High-level API: ReactCSSTransitionGroup

`ReactCSSTransitionGroup` is a high-level API based on `ReactTransitionGroup` and is an easy way to perform CSS transitions and animations when a React component enters or leaves the DOM. It's inspired by the excellent [ng-animate](#) library.

Importing

```
import ReactCSSTransitionGroup from 'react-transition-group'; // ES6
var ReactCSSTransitionGroup = require('react-transition-group'); // ES5 with npm
```

```
class TodoList extends React.Component {
  constructor(props) {
    super(props);
    this.state = {items: ['hello', 'world', 'click', 'me']};
    this.handleAdd = this.handleAdd.bind(this);
  }

  handleAdd() {
    const newItems = this.state.items.concat([
      prompt('Enter some text')
    ]);
    this.setState({items: newItems});
  }

  handleRemove(i) {
    let newItems = this.state.items.slice();
    newItems.splice(i, 1);
    this.setState({items: newItems});
  }

  render() {
    const items = this.state.items.map((item, i) => (
      <div key={i} onClick={() => this.handleRemove(i)}>
        {item}
      </div>
    )));
  }
}
```

```

    return (
      <div>
        <button onClick={this.handleAdd}>Add Item</button>
        <ReactCSSTransitionGroup
          transitionName="example"
          transitionEnterTimeout={500}
          transitionLeaveTimeout={300}>
          {items}
        </ReactCSSTransitionGroup>
      </div>
    );
  }
}

```

Note:

You must provide `the key attribute` for all children of `ReactCSSTransitionGroup`, even when only rendering a single item. This is how React will determine which children have entered, left, or stayed.

In this component, when a new item is added to `ReactCSSTransitionGroup` it will get the `example-enter` CSS class and the `example-enter-active` CSS class added in the next tick. This is a convention based on the `transitionName` prop.

You can use these classes to trigger a CSS animation or transition. For example, try adding this CSS and adding a new list item:

```

.example-enter {
  opacity: 0.01;
}

.example-enter.example-enter-active {
  opacity: 1;
  transition: opacity 500ms ease-in;
}

.example-leave {
  opacity: 1;
}

.example-leave.example-leave-active {
  opacity: 0.01;
  transition: opacity 300ms ease-in;
}

```

You'll notice that animation durations need to be specified in both the CSS and the render method; this tells React when to remove the animation classes from the element and -- if it's leaving -- when to remove the element from the DOM.

Animate Initial Mounting

`ReactCSSTransitionGroup` provides the optional prop `transitionAppear`, to add an extra transition phase at the initial mount of the component. There is generally no transition phase at the initial mount as the default value of `transitionAppear` is `false`. The following is an example which passes the prop `transitionAppear` with the value `true`.

```
render() {
  return (
    <ReactCSSTransitionGroup
      transitionName="example"
      transitionAppear={true}
      transitionAppearTimeout={500}
      transitionEnter={false}
      transitionLeave={false}>
      <h1>Fading at Initial Mount</h1>
    </ReactCSSTransitionGroup>
  );
}
```

During the initial mount `ReactCSSTransitionGroup` will get the `example-appear` CSS class and the `example-appear-active` CSS class added in the next tick.

```
.example-appear {
  opacity: 0.01;
}

.example-appear.example-appear-active {
  opacity: 1;
  transition: opacity .5s ease-in;
}
```

At the initial mount, all children of the `ReactCSSTransitionGroup` will `appear` but not `enter`. However, all children later added to an existing `ReactCSSTransitionGroup` will `enter` but not `appear`.

Note:

The prop `transitionAppear` was added to `ReactCSSTransitionGroup` in version 0.13. To maintain backwards compatibility, the default value is set to `false`.

However, the default values of `transitionEnter` and `transitionLeave` are `true` so you must specify `transitionEnterTimeout` and `transitionLeaveTimeout` by default. If you don't need either enter or leave animations, pass `transitionEnter={false}` or `transitionLeave={false}`.

Custom Classes

It is also possible to use custom class names for each of the steps in your transitions. Instead of passing a string into `transitionName` you can pass an object containing either the `enter` and `leave` class names, or an object containing the `enter`, `enter-active`, `leave-active`, and `leave` class names. If only the `enter` and `leave` classes are provided, the `enter-active` and `leave-active` classes will be determined by appending '`-active`' to the end of the class name. Here are two examples using custom classes:

```
// ...
<ReactCSSTransitionGroup
  transitionName={ {
    enter: 'enter',
    enterActive: 'enterActive',
    leave: 'leave',
    leaveActive: 'leaveActive',
    appear: 'appear',
    appearActive: 'appearActive'
  } }>
  {item}
</ReactCSSTransitionGroup>

<ReactCSSTransitionGroup
  transitionName={ {
    enter: 'enter',
    leave: 'leave',
    appear: 'appear'
  } }>
  {item2}
</ReactCSSTransitionGroup>
// ...
```

Animation Group Must Be Mounted To Work

In order for it to apply transitions to its children, the `ReactCSSTransitionGroup` must already be mounted in the DOM or the prop `transitionAppear` must be set to `true`.

The example below would **not** work, because the `ReactCSSTransitionGroup` is being mounted along with the new item, instead of the new item being mounted within it. Compare this to the [Getting Started](#) section above to see the difference.

```
render() {
  const items = this.state.items.map((item, i) => (
    <div key={item} onClick={() => this.handleRemove(i)}>
      <ReactCSSTransitionGroup transitionName="example">
        {item}
      </ReactCSSTransitionGroup>
    </div>
  )));
  return (
    <div>
      <button onClick={this.handleAdd}>Add Item</button>
      {items}
    </div>
  );
}
```

Animating One or Zero Items

In the example above, we rendered a list of items into `ReactCSSTransitionGroup`. However, the children of `ReactCSSTransitionGroup` can also be one or zero items. This makes it possible to animate a single element entering or leaving. Similarly, you can animate a new element replacing the current element. For example, we can implement a simple image carousel like this:

```
import ReactCSSTransitionGroup from 'react-transition-group';

function ImageCarousel(props) {
  return (
    <div>
      <ReactCSSTransitionGroup
        transitionName="carousel"
        transitionEnterTimeout={300}
        transitionLeaveTimeout={300}>
        <img src={props.imageSrc} key={props.imageSrc} />
      </ReactCSSTransitionGroup>
    </div>
  );
}
```

Disabling Animations

You can disable animating `enter` or `leave` animations if you want. For example, sometimes you may want an `enter` animation and no `leave` animation, but `ReactCSSTransitionGroup` waits for an animation to complete before removing your DOM node. You can add `transitionEnter={false}` or `transitionLeave={false}` props to `ReactCSSTransitionGroup` to disable these animations.

Note:

When using `ReactCSSTransitionGroup`, there's no way for your components to be notified when a transition has ended or to perform any more complex logic around animation. If you want more fine-grained control, you can use the lower-level `ReactTransitionGroup` API which provides the hooks you need to do custom transitions.

Low-level API: `ReactTransitionGroup`

Importing

```
import ReactTransitionGroup from 'react-addons-transition-group' // ES6
var ReactTransitionGroup = require('react-addons-transition-group') // ES5 with
npm
```

`ReactTransitionGroup` is the basis for animations. When children are declaratively added or removed from it (as in the [example above](#)), special lifecycle methods are called on them.

- `componentWillAppear()`

- `componentDidAppear()`
- `componentWillEnter()`
- `componentDidEnter()`
- `componentWillLeave()`
- `componentDidLeave()`

Rendering a Different Component

`ReactTransitionGroup` renders as a `span` by default. You can change this behavior by providing a `component` prop. For example, here's how you would render a ``:

```
<ReactTransitionGroup component="ul">
  {/* ... */}
</ReactTransitionGroup>
```

Any additional, user-defined, properties will become properties of the rendered component. For example, here's how you would render a `` with CSS class:

```
<ReactTransitionGroup component="ul" className="animated-list">
  {/* ... */}
</ReactTransitionGroup>
```

Every DOM component that React can render is available for use. However, `component` does not need to be a DOM component. It can be any React component you want; even ones you've written yourself! Just write `component={List}` and your component will receive `this.props.children`.

Rendering a Single Child

People often use `ReactTransitionGroup` to animate mounting and unmounting of a single child such as a collapsible panel. Normally `ReactTransitionGroup` wraps all its children in a `span` (or a custom `component` as described above). This is because any React component has to return a single root element, and `ReactTransitionGroup` is no exception to this rule.

However if you only need to render a single child inside `ReactTransitionGroup`, you can completely avoid wrapping it in a `` or any other DOM component. To do this, create a custom component that renders the first child passed to it directly:

```
function FirstChild(props) {
  const childrenArray = React.Children.toArray(props.children);
  return childrenArray[0] || null;
}
```

Now you can specify `FirstChild` as the `component` prop in `<ReactTransitionGroup>` props and avoid any wrappers in the result DOM:

```
<ReactTransitionGroup component={FirstChild}>
  {someCondition ? <MyComponent /> : null}
</ReactTransitionGroup>
```

This only works when you are animating a single child in and out, such as a collapsible panel. This approach wouldn't work when animating multiple children or replacing the single child with another child, such as an image carousel. For an image carousel, while the current image is animating out, another image will animate in, so `<ReactTransitionGroup>` needs to give them a common DOM parent. You can't avoid the wrapper for multiple children, but you can customize the wrapper with the `component` prop as described above.

Reference

`componentWillAppear()` **{#componentwillappear}**

`componentWillAppear(callback)`

This is called at the same time as `componentDidMount()` for components that are initially mounted in a `TransitionGroup`. It will block other animations from occurring until `callback` is called. It is only called on the initial render of a `TransitionGroup`.

`componentDidAppear()` **{#componentdidappear}**

`componentDidAppear()`

This is called after the `callback` function that was passed to `componentWillAppear` is called.

`componentWillEnter()` **{#componentwillenter}**

`componentWillEnter(callback)`

This is called at the same time as `componentDidMount()` for components added to an existing `TransitionGroup`. It will block other animations from occurring until `callback` is called. It will not be called on the initial render of a `TransitionGroup`.

`componentDidEnter()` **{#componentdidenter}**

`componentDidEnter()`

This is called after the `callback` function that was passed to `componentWillEnter()` is called.

`componentWillLeave()` **{#componentwillleave}**

`componentWillLeave(callback)`

This is called when the child has been removed from the `ReactTransitionGroup`. Though the child has been removed, `ReactTransitionGroup` will keep it in the DOM until `callback` is called.

componentDidLeave() {#componentdidleave}

`componentDidLeave()`

This is called when the `willLeave` `callback` is called (at the same time as `componentWillUnmount()`).

[Go to TOC](#)

Keyed Fragments

Note:

`React.addons` entry point is deprecated as of React v15.5. We now have first class support for fragments which you can read about [here](#).

Importing

```
import createFragment from 'react-addons-create-fragment'; // ES6
var createFragment = require('react-addons-create-fragment'); // ES5 with npm
```

Overview

In most cases, you can use the `key` prop to specify keys on the elements you're returning from `render`. However, this breaks down in one situation: if you have two sets of children that you need to reorder, there's no way to put a key on each set without adding a wrapper element.

That is, if you have a component such as:

```
function Swapper(props) {
  let children;
  if (props.swapped) {
    children = [props.rightChildren, props.leftChildren];
  } else {
    children = [props.leftChildren, props.rightChildren];
  }
  return <div>{children}</div>;
}
```

The children will unmount and remount as you change the `swapped` prop because there aren't any keys marked on the two sets of children.

To solve this problem, you can use the `createFragment` add-on to give keys to the sets of children.

```
Array<ReactNode> createFragment(object children) {#arrayreactnode-createfragmentobject-children}
```

Instead of creating arrays, we write:

```
import createFragment from 'react-addons-create-fragment';

function Swapper(props) {
  let children;
  if (props.swapped) {
    children = createFragment({
      right: props.rightChildren,
      left: props.leftChildren
    });
  }
  return <div>{children}</div>;
}
```

```
    } else {
      children = createFragment({
        left: props.leftChildren,
        right: props.rightChildren
      });
    }
    return <div>{children}</div>;
}
```

The keys of the passed object (that is, `left` and `right`) are used as keys for the entire set of children, and the order of the object's keys is used to determine the order of the rendered children. With this change, the two sets of children will be properly reordered in the DOM without unmounting.

The return value of `createFragment` should be treated as an opaque object; you can use the `React.Children` helpers to loop through a fragment but should not access it directly. Note also that we're relying on the JavaScript engine preserving object enumeration order here, which is not guaranteed by the spec but is implemented by all major browsers and VMs for objects with non-numeric keys.

Performance Tools

Note:

As of React 16, `react-addons-perf` is not supported. Please use [your browser's profiling tools](#) to get insight into which components re-render.

Importing

```
import Perf from 'react-addons-perf'; // ES6
var Perf = require('react-addons-perf'); // ES5 with npm
```

Overview

React is usually quite fast out of the box. However, in situations where you need to squeeze every ounce of performance out of your app, it provides a `shouldComponentUpdate()` method where you can add optimization hints to React's diff algorithm.

In addition to giving you an overview of your app's overall performance, `Perf` is a profiling tool that tells you exactly where you need to put these methods.

See these articles for an introduction to React performance tooling:

- ["How to Benchmark React Components"](#)
- ["Performance Engineering with React"](#)
- ["A Deep Dive into React Perf Debugging"](#)

Development vs. Production Builds

If you're benchmarking or seeing performance problems in your React apps, make sure you're testing with the [minified production build](#). The development build includes extra warnings that are helpful when building your apps, but it is slower due to the extra bookkeeping it does.

However, the perf tools described on this page only work when using the development build of React. Therefore, the profiler only serves to indicate the *relatively* expensive parts of your app.

Using Perf

The `Perf` object can be used with React in development mode only. You should not include this bundle when building your app for production.

Getting Measurements

- `start()`
- `stop()`

- `getLastMeasurements()`

Printing Results

The following methods use the measurements returned by `Perf.getLastMeasurements()` to pretty-print the result.

- `printInclusive()`
 - `printExclusive()`
 - `printWasted()`
 - `printOperations()`
 - `printDOM()`
-

Reference

`start() {#start}`

`stop() {#stop}`

```
Perf.start()  
// ...  
Perf.stop()
```

Start/stop the measurement. The React operations in-between are recorded for analyses below. Operations that took an insignificant amount of time are ignored.

After stopping, you will need `Perf.getLastMeasurements()` to get the measurements.

`getLastMeasurements() {#getlastmeasurements}`

```
Perf.getLastMeasurements()
```

Get the opaque data structure describing measurements from the last start-stop session. You can save it and pass it to the other print methods in `Perf` to analyze past measurements.

Note

Don't rely on the exact format of the return value because it may change in minor releases. We will update the documentation if the return value format becomes a supported part of the public API.

`printInclusive() {#printinclusive}`

```
Perf.printInclusive(measurements)
```

Prints the overall time taken. When no arguments are passed, `printInclusive` defaults to all the measurements from the last recording. This prints a nicely formatted table in the console, like so:

(index)	Owner > component	Inclusive time (ms)	Instances
0	"<root> > App"	15.31	37
1	"App > Box"	6.47	37
2	"Box > Box2"	1.96	37

printExclusive() {#printexclusive}

`Perf.printExclusive(measurements)`

"Exclusive" times don't include the times taken to mount the components: processing props, calling `componentWillMount` and `componentDidMount`, etc.

(index)	Component c...	Total inclus...	Exclusive m...	Exclusive r...	Mount time ...	Render time...	Instances
0	"App"	15.31	1.76	1.23	0.04	0.03	37

printWasted() {#printwasted}

`Perf.printWasted(measurements)`

The most useful part of the profiler.

"Wasted" time is spent on components that didn't actually render anything, e.g. the render stayed the same, so the DOM wasn't touched.

(index)	Owner > component	Wasted time (ms)	Instances
0	"<root> > App"	15.317999947001226	37
1	"App > Box"	6.479999952716753	37
2	"Box > Box2"	1.9630000460892916	37

printOperations() {#printoperations}

`Perf.printOperations(measurements)`

Prints the underlying DOM manipulations, e.g. "set innerHTML" and "remove".

(index)	data-reactid	type	args
0	"0"	"set innerHTML"	"<div data-reactid=".0\..."

printDOM() {#printdom}

`Perf.printDOM(measurements)`

This method has been renamed to `printOperations()`. Currently `printDOM()` still exists as an alias but it prints a deprecation warning and will eventually be removed.

[Go to TOC](#)

PureRenderMixin

Note:

`PureRenderMixin` is a legacy add-on. Use `React.PureComponent` instead.

Importing

```
import PureRenderMixin from 'react-addons-pure-render-mixin'; // ES6
var PureRenderMixin = require('react-addons-pure-render-mixin'); // ES5 with npm
```

Overview

If your React component's render function renders the same result given the same props and state, you can use this mixin for a performance boost in some cases.

Example:

```
const createReactClass = require('create-react-class');

createReactClass({
  mixins: [PureRenderMixin],

  render: function() {
    return <div className={this.props.className}>foo</div>;
  }
});
```

Under the hood, the mixin implements `shouldComponentUpdate`, in which it compares the current props and state with the next ones and returns `false` if the equalities pass.

Note:

This only shallowly compares the objects. If these contain complex data structures, it may produce false-negatives for deeper differences. Only mix into components which have simple props and state, or use `forceUpdate()` when you know deep data structures have changed. Or, consider using [immutable objects](#) to facilitate fast comparisons of nested data.

Furthermore, `shouldComponentUpdate` skips updates for the whole component subtree. Make sure all the children components are also "pure".

[Go to TOC](#)

Shallow Compare

Note:

`shallowCompare` is a legacy add-on. Use `React.memo` or `React.PureComponent` instead.

Importing

```
import shallowCompare from 'react-addons-shallow-compare'; // ES6
var shallowCompare = require('react-addons-shallow-compare'); // ES5 with npm
```

Overview

Before `React.PureComponent` was introduced, `shallowCompare` was commonly used to achieve the same functionality as `PureRenderMixin` while using ES6 classes with React.

If your React component's render function is "pure" (in other words, it renders the same result given the same props and state), you can use this helper function for a performance boost in some cases.

Example:

```
export class SampleComponent extends React.Component {
  shouldComponentUpdate(nextProps, nextState) {
    return shallowCompare(this, nextProps, nextState);
  }

  render() {
    return <div className={this.props.className}>foo</div>;
  }
}
```

`shallowCompare` performs a shallow equality check on the current `props` and `nextProps` objects as well as the current `state` and `nextState` objects.

It does this by iterating on the keys of the objects being compared and returning true when the values of a key in each object are not strictly equal.

`shallowCompare` returns `true` if the shallow comparison for props or state fails and therefore the component should update.

`shallowCompare` returns `false` if the shallow comparison for props and state both pass and therefore the component does not need to update.

[Go to TOC](#)

浅层渲染

如何引入

```
import ShallowRenderer from 'react-test-renderer/shallow'; // ES6
var ShallowRenderer = require('react-test-renderer/shallow'); // ES5 with npm
```

概览

当为 React 编写单元测试时，浅层渲染十分有用。浅层渲染可以只渲染组件的“第一层”，并且对组件的 render 方法的返回值进行断言，不必担心子组件的行为，子组件并没有实例化或被渲染。并且浅层渲染不依赖 DOM。

例如，如果你有如下的组件：

```
function MyComponent() {
  return (
    <div>
      <span className="heading">Title</span>
      <Subcomponent foo="bar" />
    </div>
  );
}
```

你可以使用断言：

```
import ShallowRenderer from 'react-test-renderer/shallow';

// 测试代码:
const renderer = new ShallowRenderer();
renderer.render(<MyComponent />);
const result = renderer.getRenderOutput();

expect(result.type).toBe('div');
expect(result.props.children).toEqual([
  <span className="heading">Title</span>,
  <Subcomponent foo="bar" />
]);
```

浅层测试（Shallow testing）目前有些局限性，即不支持 refs。

注意：

建议你查阅 Enzyme 的[浅层渲染的 API](#)。它在相同的功能基础上提供了更棒更高级的 API。

参考

`shallowRenderer.render()` [\[#shallowrendererrender\]](#)

你可以把 shallowRenderer 看作用来渲染测试中组件的“容器”，且可以从容器中取到该组件的输出内容。

`shallowRenderer.render()` 和 `ReactDOM.render()` 很像，但是它不依赖 DOM 且只渲染一层。这意味着你可以对组件和其子组件的实现进行隔离测试。

`shallowRenderer.getRenderOutput()` {#shallowrenderergetrenderoutput}

在 `shallowRenderer.render()` 被调用后，你可以使用 `shallowRenderer.getRenderOutput()` 来获取浅层渲染的输出内容。

然后，你就可以开始对输出内容进行断言操作。

Test Utilities

如何引入

```
import ReactTestUtils from 'react-dom/test-utils'; // ES6
var ReactTestUtils = require('react-dom/test-utils'); // ES5 使用 npm 的方式
```

概览

`ReactTestUtils` 可搭配你所选的测试框架，轻松实现 React 组件测试。在 Facebook 内部，我们使用 [Jest](#) 来轻松实现 JavaScript 测试。你可以从 [Jest 官网的 React 教程](#)中了解如何开始使用它。

注意：

我们推荐使用 [React Testing Library](#)，它使得针对组件编写测试用例就像终端用户在使用它一样方便。

当使用的 React 版本 ≤ 16 时，可以使用 [Enzyme](#) 的测试工具，通过它能够轻松对 React 组件的输出进行断言、操控和遍历。

- `act()`
- `mockComponent()`
- `isElement()`
- `isElementOfType()`
- `isDOMComponent()`
- `isCompositeComponent()`
- `isCompositeComponentWithType()`
- `findAllInRenderedTree()`
- `scryRenderedDOMComponentsWithClass()`
- `findRenderedDOMComponentWithClass()`
- `scryRenderedDOMComponentsWithTag()`
- `findRenderedDOMComponentWithTag()`
- `scryRenderedComponentsWithType()`
- `findRenderedComponentWithType()`
- `renderIntoDocument()`
- `Simulate`

参考

`act()` {#act}

为断言准备一个组件，包裹要渲染的代码并在调用 `act()` 时执行更新。这会使得测试更接近 React 在浏览器中的工作方式。

注意

如果你使用了 `react-test-renderer`，它也提供了与 `act` 行为相同的函数。

例如，假设我们有个 `Counter` 组件：

```
class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {count: 0};
    this.handleClick = this.handleClick.bind(this);
  }
  componentDidMount() {
    document.title = `You clicked ${this.state.count} times`;
  }
  componentDidUpdate() {
    document.title = `You clicked ${this.state.count} times`;
  }
  handleClick() {
    this.setState(state => ({
      count: state.count + 1,
    }));
  }
  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button onClick={this.handleClick}>
          Click me
        </button>
      </div>
    );
  }
}
```

以下是其测试代码：

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import { act } from 'react-dom/test-utils';
import Counter from './Counter';

let container;

beforeEach(() => {
  container = document.createElement('div');
  document.body.appendChild(container);
});

afterEach(() => {
  document.body.removeChild(container);
  container = null;
});

it('can render and update a counter', () => {
  // 首先测试 render 和 componentDidMount
  act(() => {
    ReactDOM.createRoot(container).render(<Counter />);
  });
});
```

```

});  

const button = container.querySelector('button');  

const label = container.querySelector('p');  

expect(label.textContent).toBe('You clicked 0 times');  

expect(document.title).toBe('You clicked 0 times');  
  

// 再测试 render 和 componentDidUpdate  

act(() => {  

  button.dispatchEvent(new MouseEvent('click', {bubbles: true}));  

});  

expect(label.textContent).toBe('You clicked 1 times');  

expect(document.title).toBe('You clicked 1 times');  

});

```

千万不要忘记，只有将 DOM 容器添加到 `document` 时，触发 DOM 事件才生效。你可以使用类似于 [React Testing Library](#) 等库来减少样板代码 (boilerplate code)。

- `recipes` 文档包含了关于 `act()` 函数的示例、用法及更多详细信息。

`mockComponent()` **{#mockcomponent}**

```

mockComponent(  

  componentClass,  

  [mockTagName]  

)

```

将模拟组件模块传入这个方法后，React 内部会使用有效的方法填充该模块，使其成为虚拟的 React 组件。与通常的渲染不同，组件将变成一个简单的 `<div>` (如果提供了 `mockTagName` 则是其他标签)，包含任何提供的子级。

注意：

`mockComponent()` 是一个过时的 API，我们推荐使用 `jest.mock()` 来代替。

`isElement()` **{#iselement}**

```
isElement(element)
```

当 `element` 是任何一种 React 元素时，返回 `true`。

`isElementOfType()` **{#iselementoftype}**

```

isElementOfType(  

  element,  

  componentClass  

)

```

当 `element` 是一种 React 元素，并且它的类型是参数 `componentClass` 的类型时，返回 `true`。

isDOMComponent() {#isdomcomponent}

```
isDOMComponent(instance)
```

当 `instance` 是一个 DOM 组件（比如 `<div>` 或 ``）时，返回 `true`。

isCompositeComponent() {#iscompositecomponent}

```
isCompositeComponent(instance)
```

当 `instance` 是一个用户自定义的组件，比如一个类或者一个函数时，返回 `true`。

isCompositeComponentWithType() {#iscompositecomponentwithtype}

```
isCompositeComponentWithType(
  instance,
  componentClass
)
```

当 `instance` 是一个组件，并且它的类型是参数 `componentClass` 的类型时，返回 `true`。

findAllInRenderedTree() {#findallinrenderedtree}

```
findAllInRenderedTree(
  tree,
  test
)
```

遍历所有在参数 `tree` 中的组件，记录所有 `test(component)` 为 `true` 的组件。单独调用此方法不是很有用，但是它常常被作为底层 API 被其他测试方法使用。

**scryRenderedDOMComponentsWithClass()
{#scryrendereddomcomponentswithclass}**

```
scryRenderedDOMComponentsWithClass(
  tree,
  className
)
```

查找渲染树中组件的所有 DOM 元素，这些组件是 css 类名与参数 `className` 匹配的 DOM 组件。

**findRenderedDOMComponentWithClass()
{#findrendereddomcomponentwithclass}**

```
findRenderedDOMComponentWithClass(
  tree,
  className
)
```

用法与 `scryRenderedDOMComponentsWithClass()` 保持一致，但期望仅返回一个结果。不符合预期的情况下会抛出异常。

`scryRenderedDOMComponentsWithTag()` {#scryrendereddomcomponentswithtag}

```
scryRenderedDOMComponentsWithTag(
  tree,
  tagName
)
```

查找渲染树中组件的所有的 DOM 元素，这些组件是标记名与参数 `tagName` 匹配的 DOM 组件。

`findRenderedDOMComponentWithTag()` {#findrendereddomcomponentwithtag}

```
findRenderedDOMComponentWithTag(
  tree,
  tagName
)
```

用法与 `scryRenderedDOMComponentsWithTag()` 保持一致，但期望仅返回一个结果。不符合预期的情况下会抛出异常。

`scryRenderedComponentsWithType()` {#scryrenderedcomponentswithtype}

```
scryRenderedComponentsWithType(
  tree,
  componentClass
)
```

查找组件类型等于 `componentClass` 组件的所有实例。

`findRenderedComponentWithType()` {#findrenderedcomponentwithtype}

```
findRenderedComponentWithType(
  tree,
  componentClass
)
```

用法与 `scryRenderedComponentsWithType()` 保持一致，但期望仅返回一个结果。不符合预期的情况下会抛出异常。

`renderIntoDocument()` {#renderintodocument}

```
renderIntoDocument(element)
```

渲染 React 元素到 document 中的某个单独的 DOM 节点上。这个函数需要一个 DOM 对象。它实际相当于：

```
const domContainer = document.createElement('div');
ReactDOM.createRoot(domContainer).render(element);
```

注意：

你需要在引入 React 之前确保 window 存在，window.document 和 window.document.createElement 能在全局环境中获取到。不然 React 会认为它没有权限去操作 DOM，以及像 setState 这样的方法将不可用。

其他工具方法

Simulate {#simulate}

```
Simulate.{eventName}(
  element,
  [eventData]
)
```

使用可选的 eventData 事件数据来模拟在 DOM 节点上触发事件。

React 所支持的所有事件 在 Simulate 中都有对应的方法。

点击元素

```
// <button ref={(node) => this.button = node}>...</button>
const node = this.button;
ReactTestUtils.Simulate.click(node);
```

修改一个 input 输入框的值，然后按回车键。

```
// <input ref={(node) => this.textInput = node} />
const node = this.textInput;
node.value = 'giraffe';
ReactTestUtils.Simulate.change(node);
ReactTestUtils.Simulate.keyDown(node, {key: "Enter", keyCode: 13, which: 13});
```

注意：

你必须提供一切需要在组件中用到的事件属性（比如：keyCode、which 等等），因为 React 没有为你创建这些属性。

Two-way Binding Helpers

Note:

`LinkedStateMixin` is deprecated as of React v15. The recommendation is to explicitly set the value and change handler, instead of using `LinkedStateMixin`.

Importing

```
import LinkedStateMixin from 'react-addons-linked-state-mixin'; // ES6
var LinkedStateMixin = require('react-addons-linked-state-mixin'); // ES5 with npm
```

Overview

`LinkedStateMixin` is an easy way to express two-way binding with React.

In React, data flows one way: from owner to child. We think that this makes your app's code easier to understand. You can think of it as "one-way data binding."

However, there are lots of applications that require you to read some data and flow it back into your program. For example, when developing forms, you'll often want to update some React `state` when you receive user input. Or perhaps you want to perform layout in JavaScript and react to changes in some DOM element size.

In React, you would implement this by listening to a "change" event, read from your data source (usually the DOM) and call `setState()` on one of your components. "Closing the data flow loop" explicitly leads to more understandable and easier-to-maintain programs. See [our forms documentation](#) for more information.

Two-way binding -- implicitly enforcing that some value in the DOM is always consistent with some React `state` -- is concise and supports a wide variety of applications. We've provided `LinkedStateMixin`: syntactic sugar for setting up the common data flow loop pattern described above, or "linking" some data source to React `state`.

Note:

`LinkedStateMixin` is just a thin wrapper and convention around the `onChange / setState()` pattern. It doesn't fundamentally change how data flows in your React application.

LinkedStateMixin: Before and After

Here's a simple form example without using `LinkedStateMixin`:

```
var createReactClass = require('create-react-class');

var NoLink = createReactClass({
  getInitialState: function() {
    return {message: 'Hello!'};
  },
  handleChange: function(event) {
    this.setState({message: event.target.value});
  },
  render: function() {
    var message = this.state.message;
    return <input type="text" value={message} onChange={this.handleChange}>;
  }
});
```

This works really well and it's very clear how data is flowing, however, with a lot of form fields it could get a bit verbose. Let's use `LinkedStateMixin` to save us some typing:

```
var createReactClass = require('create-react-class');

var WithLink = createReactClass({
  mixins: [LinkedStateMixin],
  getInitialState: function() {
    return {message: 'Hello!'};
  },
  render: function() {
    return <input type="text" valueLink={this.linkState('message')} />;
  }
});
```

`LinkedStateMixin` adds a method to your React component called `linkState()`. `linkState()` returns a `valueLink` object which contains the current value of the React state and a callback to change it.

`valueLink` objects can be passed up and down the tree as props, so it's easy (and explicit) to set up two-way binding between a component deep in the hierarchy and state that lives higher in the hierarchy.

Note that checkboxes have a special behavior regarding their `value` attribute, which is the value that will be sent on form submit if the checkbox is checked (defaults to `on`). The `value` attribute is not updated when the checkbox is checked or unchecked. For checkboxes, you should use `checkedLink` instead of `valueLink`:

```
<input type="checkbox" checkedLink={this.linkState('booleanValue')} />
```

Under the Hood

There are two sides to `LinkedStateMixin`: the place where you create the `valueLink` instance and the place where you use it. To prove how simple `LinkedStateMixin` is, let's rewrite each side separately to be more explicit.

valueLink Without LinkedStateMixin

```
var createReactClass = require('create-react-class');

var WithoutMixin = createReactClass({
```

```

getInitialState: function() {
  return {message: 'Hello!'};
},
handleChange: function(newValue) {
  this.setState({message: newValue});
},
render: function() {
  var valueLink = {
    value: this.state.message,
    requestChange: this.handleChange
  };
  return <input type="text" valueLink={valueLink} />;
}
);

```

As you can see, `valueLink` objects are very simple objects that just have a `value` and `requestChange` prop. And `LinkedStateMixin` is similarly simple: it just populates those fields with a `value` from `this.state` and a callback that calls `this.setState()`.

LinkedStateMixin Without valueLink

```

var LinkedStateMixin = require('react-addons-linked-state-mixin');
var createReactClass = require('create-react-class');

var WithoutLink = createReactClass({
  mixins: [LinkedStateMixin],
  getInitialState: function() {
    return {message: 'Hello!'};
  },
  render: function() {
    var valueLink = this.linkState('message');
    var handleChange = function(e) {
      valueLink.requestChange(e.target.value);
    };
    return <input type="text" value={valueLink.value} onChange={handleChange} />;
  }
});

```

The `valueLink` prop is also quite simple. It simply handles the `onChange` event and calls `this.props.valueLink.requestChange()` and also uses `this.props.valueLink.value` instead of `this.props.value`. That's it!

Immutability Helpers

Note:

`update` is a legacy add-on. Use `immutability-helper` instead.

Importing

```
import update from 'react-addons-update'; // ES6
var update = require('react-addons-update'); // ES5 with npm
```

Overview

React lets you use whatever style of data management you want, including mutation. However, if you can use immutable data in performance-critical parts of your application it's easy to implement a fast `shouldComponentUpdate()` method to significantly speed up your app.

Dealing with immutable data in JavaScript is more difficult than in languages designed for it, like [Clojure](#). However, we've provided a simple immutability helper, `update()`, that makes dealing with this type of data much easier, *without* fundamentally changing how your data is represented. You can also take a look at Facebook's [Immutable-js](#) and the [Advanced Performance](#) section for more detail on Immutable-js.

The Main Idea

If you mutate data like this:

```
myData.x.y.z = 7;
// or...
myData.a.b.push(9);
```

You have no way of determining which data has changed since the previous copy has been overwritten. Instead, you need to create a new copy of `myData` and change only the parts of it that need to be changed. Then you can compare the old copy of `myData` with the new one in `shouldComponentUpdate()` using triple-equals:

```
const newData = deepCopy(myData);
newData.x.y.z = 7;
newData.a.b.push(9);
```

Unfortunately, deep copies are expensive, and sometimes impossible. You can alleviate this by only copying objects that need to be changed and by reusing the objects that haven't changed. Unfortunately, in today's JavaScript this can be cumbersome:

```
const newData = extend(myData, {
  x: extend(myData.x, {
    y: extend(myData.x.y, {z: 7}),
```

```
  },
  a: extend(myData.a, {b: myData.a.b.concat(9)})
});
```

While this is fairly performant (since it only makes a shallow copy of `log n` objects and reuses the rest), it's a big pain to write. Look at all the repetition! This is not only annoying, but also provides a large surface area for bugs.

update() {#update}

`update()` provides simple syntactic sugar around this pattern to make writing this code easier. This code becomes:

```
import update from 'react-addons-update';

const newData = update(myData, {
  x: {y: {z: {$set: 7}}},
  a: {b: {$push: [9]}}
});
```

While the syntax takes a little getting used to (though it's inspired by MongoDB's query language) there's no redundancy, it's statically analyzable and it's not much more typing than the mutative version.

The `$`-prefixed keys are called *commands*. The data structure they are "mutating" is called the *target*.

Available Commands

- `{$push: array} push()` all the items in `array` on the target.
- `{$unshift: array} unshift()` all the items in `array` on the target.
- `{$splice: array of arrays}` for each item in `arrays` call `splice()` on the target with the parameters provided by the item.
- `{$set: any}` replace the target entirely.
- `{$merge: object}` merge the keys of `object` with the target.
- `{$apply: function}` passes in the current value to the function and updates it with the new returned value.

Examples

Simple push

```
const initialArray = [1, 2, 3];
const newArray = update(initialArray, {$push: [4]}); // => [1, 2, 3, 4]
```

`initialArray` is still `[1, 2, 3]`.

Nested collections

```
const collection = [1, 2, {a: [12, 17, 15]}];
const newCollection = update(collection, {2: {a: {$splice: [[1, 1, 13, 14]]}}});
// => [1, 2, {a: [12, 13, 14, 15]}]
```

This accesses `collection`'s index `2`, key `a`, and does a splice of one item starting from index `1` (to remove `17`) while inserting `13` and `14`.

Updating a value based on its current one

```
const obj = {a: 5, b: 3};
const newObj = update(obj, {b: {$apply: function(x) {return x * 2;}}});
// => {a: 5, b: 6}
// This is equivalent, but gets verbose for deeply nested collections:
const newObj2 = update(obj, {b: {$set: obj.b * 2}});
```

(Shallow) Merge

```
const obj = {a: 5, b: 3};
const newObj = update(obj, {$merge: {b: 6, c: 7}}); // => {a: 5, b: 6, c: 7}
```

Add-Ons

Note:

`React.addons` entry point is deprecated as of React v15.5. The add-ons have moved to separate modules, and some of them have been deprecated.

The React add-ons are a collection of useful utility modules for building React apps. **These should be considered experimental** and tend to change more often than the core.

- `createFragment`, to create a set of externally-keyed children.

The add-ons below are in the development (unminified) version of React only:

- `Perf`, a performance profiling tool for finding optimization opportunities.
- `ReactTestUtils`, simple helpers for writing test cases.

Legacy Add-ons

The add-ons below are considered legacy and their use is discouraged. They will keep working in observable future, but there is no further development.

- `PureRenderMixin`. Use `React.PureComponent` instead.
- `shallowCompare`, a helper function that performs a shallow comparison for props and state in a component to decide if a component should update. We recommend using `React.PureComponent` instead.
- `update`. Use `kolodny/immutability-helper` instead.
- `ReactDOMFactories`, pre-configured DOM factories to make React easier to use without JSX.

Deprecated Add-ons

- `LinkedStateMixin` has been deprecated.
- `TransitionGroup` and `CSSTransitionGroup` have been deprecated in favor of [their drop-in replacements](#).

Using React with Add-ons

You can install the add-ons individually from npm (e.g. `npm install react-addons-create-fragment`) and import them:

```
import createFragment from 'react-addons-create-fragment'; // ES6
var createFragment = require('react-addons-create-fragment'); // ES5 with npm
```

When using React 15 or earlier from a CDN, you can use `react-with-addons.js` instead of `react.js`:

```
<script src="https://unpkg.com/react@15/dist/react-with-addons.js"></script>
```

The add-ons will be available via the `React.addons` global (e.g. `React.addons.TestUtils`).

[Go to TOC](#)

CDN 链接

可以通过 CDN 获得 React 和 ReactDOM 的 UMD 版本。

```
<script crossorigin src="https://unpkg.com/react@18/umd/react.development.js">
</script>
<script crossorigin src="https://unpkg.com/react-dom@18/umd/react-
dom.development.js"></script>
```

上述版本仅用于开发环境，不适合用于生产环境。压缩优化后可用于生产的 React 版本可通过如下方式引用：

```
<script crossorigin src="https://unpkg.com/react@18/umd/react.production.min.js">
</script>
<script crossorigin src="https://unpkg.com/react-dom@18/umd/react-
dom.production.min.js"></script>
```

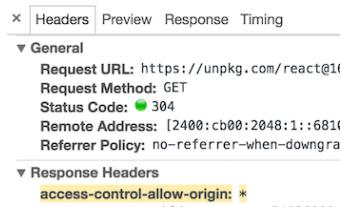
如果需要加载指定版本的 `react` 和 `react-dom`，可以把 `18` 替换成所需加载的版本号。

为什么要使用 `crossorigin` 属性？

如果你通过 CDN 的方式引入 React，我们建议你设置 `crossorigin` 属性：

```
<script crossorigin src="... "></script>
```

我们同时建议你验证使用的 CDN 是否设置了 `Access-Control-Allow-Origin: *` HTTP 请求头：



`Access-Control-Allow-Origin: *`

这样能在 React 16 及以上的版本中有更好的[错误处理体验](#)。

代码分割

打包

大多数 React 应用都会使用 [Webpack](#), [Rollup](#) 或 [Browserify](#) 这类的构建工具来打包文件。打包是一个将文件引入并合并到一个单独文件的过程，最终形成一个“bundle”。接着在页面上引入该 bundle，整个应用即可一次性加载。

示例

App文件：

```
// app.js
import { add } from './math.js';

console.log(add(16, 26)); // 42
```

```
// math.js
export function add(a, b) {
  return a + b;
}
```

打包后文件：

```
function add(a, b) {
  return a + b;
}

console.log(add(16, 26)); // 42
```

注意：

最终你打包后的文件可能看起来会和上面的例子区别很大。

如果你正在使用 [Create React App](#), [Next.js](#), [Gatsby](#), 或者类似的工具，你可以直接使用内置的 Webpack 配置来构建你的应用。

如果你没有使用这类工具，你就需要自己来进行配置。例如，查看 Webpack 文档上的[安装](#)和[入门教程](#)。

代码分割

打包是个非常棒的技术，但随着你的应用增长，你的代码包也将随之增长。尤其是在整合了体积巨大的第三方库的情况下。你需要关注你代码包中所包含的代码，以避免因体积过大而导致加载时间过长。

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](#), [Rollup](#) 和 [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() {#import}

在你的应用中引入代码分割的最佳方式是通过动态 `import()` 语法。

使用之前：

```
import { add } from './math';
console.log(add(16, 26));
```

使用之后：

```
import("./math").then(math => {
  console.log(math.add(16, 26));
});
```

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](#)该特性。[Next.js](#) 也已支持该特性而无需进行配置。

如果你自己配置 Webpack，你可能要阅读下 Webpack 关于[代码分割](#)的指南。你的 Webpack 配置应该[类似于此](#)。

当使用 `Babel` 时，你要确保 `Babel` 能够解析动态 `import` 语法而不是将其进行转换。对于这一要求你需要 [@babel/plugin-syntax-dynamic-import](#) 插件。

React.lazy {#reactlazy}

`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）。

使用之前：

```
import OtherComponent from './OtherComponent';
```

使用之后：

```
const OtherComponent = React.lazy(() => import('./OtherComponent'));
```

此代码将会在组件首次渲染时，自动导入包含 `OtherComponent` 组件的包。

`React.lazy` 接受一个函数，这个函数需要动态调用 `import()`。它必须返回一个 `Promise`，该 `Promise` 需要 `resolve` 一个 `default` `export` 的 React 组件。

然后应在 `Suspense` 组件中渲染 `lazy` 组件，如此使得我们可以使用在等待加载 `lazy` 组件时做优雅降级（如 `loading` 指示器等）。

```
import React, { Suspense } from 'react';
const OtherComponent = React.lazy(() => import('./OtherComponent'));

function MyComponent() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
```

```

        <OtherComponent />
      </Suspense>
    </div>
  );
}

```

`fallback` 属性接受任何在组件加载过程中你想展示的 React 元素。你可以将 `Suspense` 组件置于懒加载组件之上的任何位置。你甚至可以用一个 `Suspense` 组件包裹多个懒加载组件。

```

import React, { Suspense } from 'react';

const OtherComponent = React.lazy(() => import('./OtherComponent'));
const AnotherComponent = React.lazy(() => import('./AnotherComponent'));

function MyComponent() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <section>
          <OtherComponent />
          <AnotherComponent />
        </section>
      </Suspense>
    </div>
  );
}

```

避免兜底

任何组件都可能因渲染而暂停，甚至是已经展示给用户的组件。为了使屏幕内容始终一致，如果一个已经显示的组件暂停，React 必须隐藏它的树，直到最近的 `<Suspense>` 边界。然而，从用户的角度来看，这可能会使人很困惑。

参考这个标签切换的示例：

```

import React, { Suspense } from 'react';
import Tabs from './Tabs';
import Glimmer from './Glimmer';

const Comments = React.lazy(() => import('./Comments'));
const Photos = React.lazy(() => import('./Photos'));

function MyComponent() {
  const [tab, setTab] = React.useState('photos');

  function handleTabSelect(tab) {
    setTab(tab);
  };

  return (
    <div>
      <Tabs onTabSelect={handleTabSelect} />
      <Suspense fallback={<Glimmer />}>
        {tab === 'photos' ? <Photos /> : <Comments />}
      </Suspense>
    </div>
  );
}

```

在这个示例中，如果标签从 `'photos'` 切换为 `'comments'`，但 `Comments` 会暂停，用户会看到屏幕闪烁。这符合常理，因为用户不想看到 `'photos'`，而 `Comments` 组件还没有准备好渲染其内容，而 React 为了保证用户体验的一致性，只能显示上面的 `Glimmer`，别无选择。

然而，有时这种用户体验并不可取。特别是在准备新 UI 时，展示“旧”的 UI 会体验更好。你可以尝试使用新的 `startTransition` API 来让 React 实现这一点：

```
function handleTabSelect(tab) {
  startTransition(() => {
    setTab(tab);
  });
}
```

此处代码会告知 React，将标签切换为 `'comments'` 不会标记为紧急更新，而是标记为需要一些准备时间的 `transition`。然后 React 会保留旧的 UI 并进行交互，当它准备好时，会切换为 `<Comments />`，具体请参阅 [Transitions](#) 以了解更多信息。

异常捕获边界 (Error boundaries)

如果模块加载失败（如网络问题），它会触发一个错误。你可以通过[异常捕获边界 \(Error boundaries\)](#) 技术来处理这些情况，以显示良好的用户体验并管理恢复事宜。

```
import React, { Suspense } from 'react';
import MyErrorBoundary from './MyErrorBoundary';

const OtherComponent = React.lazy(() => import('./OtherComponent'));
const AnotherComponent = React.lazy(() => import('./AnotherComponent'));

const MyComponent = () => (
  <div>
    <MyErrorBoundary>
      <Suspense fallback={<div>Loading...</div>}>
        <section>
          <OtherComponent />
          <AnotherComponent />
        </section>
      </Suspense>
    </MyErrorBoundary>
  </div>
);
```

基于路由的代码分割

决定在哪引入代码分割需要一些技巧。你需要确保选择的位置能够均匀地分割代码包而不会影响用户体验。

一个不错的选择是从路由开始。大多数网络用户习惯于页面之间能有个加载切换过程。你也可以选择重新渲染整个页面，这样您的用户就不必在渲染的同时再和页面上的其他元素进行交互。

这里是一个例子，展示如何在你的应用中使用 `React.lazy` 和 `React Router` 这类的第三方库，来配置基于路由的代码分割。

```
import React, { Suspense, lazy } from 'react';
import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';
```

```

const Home = lazy(() => import('./routes/Home'));
const About = lazy(() => import('./routes/About'));

const App = () => (
  <Router>
    <Suspense fallback={<div>Loading...</div>}>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
      </Routes>
    </Suspense>
  </Router>
);

```

命名导出(Named Exports)

`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。

```

// ManyComponents.js
export const MyComponent = /* ... */;
export const MyUnusedComponent = /* ... */;

```

```

// MyComponent.js
export { MyComponent as default } from "./ManyComponents.js";

```

```

// MyApp.js
import React, { lazy } from 'react';
const MyComponent = lazy(() => import("./MyComponent.js"));

```

源码概览

本节将对 React 的源码架构、约定及其实现进行概述。

如果您想[参与 React](#)的开发，我们希望这份指南可以帮助你更加轻松地进行修改。

我们并不推荐在 React 应用中遵循这些约定。有许多约定是历史原因，并且之后也许会有所修改。

项目根目录

当克隆 [React 仓库](#) 之后，你们将看到项目根目录的信息：

- `packages` 包含元数据（比如 `package.json`）和 React 仓库中所有 package 的源码（子目录 `src`）。**如果你需要修改源代码，那么每个包的 `src` 子目录是你最需要花费精力的地方。**
- `fixtures` 包含一些给贡献者准备的小型 React 测试项目。
- `build` 是 React 的输出目录。源码仓库中并没有这个目录，但是它会在你克隆 React 并且第一次[构建它](#)之后出现。

文档位于 [React 仓库之外的一个独立仓库中](#)。

还有一些其他的顶层目录，但是它们几乎都是工具类的，并且在贡献代码时基本不会涉及。

共置测试

我们没有单元测试的顶层目录。而是将它们放置在所需测试文件的相同目录下的 `__tests__` 的目录之中。

比如，一个用于 `setInnerHTML.js` 的测试文件，会存放在 `__tests__/setInnerHTML-test.js`，就在它同级目录下。

warning 和 invariant

React 代码库直接使用 `console.error` 来展示 warnings：

```
if (__DEV__) {
  console.error('Something is wrong.');
}
```

`warning` 仅在开发环境中启用。在生产环境中，他们会被完全剔除掉。如果你需要在生产环境禁止执行某些代码，请使用 `invariant` 模块代替 `warning`：

```
var invariant = require('invariant');

invariant(
  2 + 2 === 4,
  'You shall not pass!'
);
```

当 `invariant` 判别条件为 `false` 时，会将 `invariant` 的信息作为错误抛出

“Invariant”用于声明“这个条件应总为 true”。你可以把它当成一种断言。

保持开发和生产环境的行为相似是十分重要的，因此 `invariant` 在开发和生产环境下都会抛出错误。不同点在于在生产环境中这些错误信息会被自动替换成错误代码，这样可以让输出库文件变得更小。

开发环境与生产环境

你可以在代码库中使用 `__DEV__` 这个伪全局变量，用于管理开发环境中需运行的代码块

这在编译阶段会被内联，在 CommonJS 构建中，转化成 `process.env.NODE_ENV !== 'production'` 这样的判断。

对于独立构建来说，在没有 `minify` 的构建中，它会变成 `true`，同时在 `minify` 的构建中，检测到的 `if` 代码块会被完全剔除。

```
if (__DEV__) {
  // 仅在开发环境下执行的代码
}
```

Flow

我们最近将 `Flow` 引入源码，用于类型检查。在许可证头部的注释中，标记为 `@flow` 注释的文件是已经经过类型检查的。

我们接受[添加 Flow 注释到现有代码](#)。Flow 注释看上去像这样：

```
ReactRef.detachRefs = function(
  instance: ReactInstance,
  element: ReactElement | string | number | null | false,
): void {
  // ...
}
```

如果可以的话，新代码应尽量使用 Flow 注释。你可以运行 `yarn flow`，用 Flow 本地检查你的代码。

Multiple Packages

React 采用 `monorepo` 的管理方式。仓库中包含多个独立的包，以便于更改可以一起联调，并且问题只会出现在同一地方。

React Core

React “Core” 中包含所有[全局 React API](#)，比如：

- `React.createElement()`
- `React.Component`
- `React.Children`

React 核心只包含定义组件必要的 API。它不包含[协调](#)算法或者其他平台特定的代码。它同时适用于 React DOM 和 React Native 组件。

React 核心代码在源码的 `packages/react` 目录中。在 npm 上发布为 `react` 包。相应的独立浏览器构建版本称为 `react.js`，它会导出一个称为 `React` 的全局对象。

渲染器

React 最初只是服务于 DOM，但是这之后被改编成也能同时支持原生平台的 [React Native](#)。因此，在 React 内部机制中引入了“渲染器”这个概念。

渲染器用于管理一棵 React 树，使其根据底层平台进行不同的调用。

渲染器同样位于 `packages/` 目录下：

- [ReactDOM Renderer](#) 将 React 组件渲染成 DOM。它实现了全局 `ReactDOM API`，这在 npm 上作为 `react-dom` 包。这也同样可以作为单独浏览器版本使用，称为 `react-dom.js`，导出一个 `ReactDOM` 的全局对象。
- [React Native Renderer](#) 将 React 组件渲染为 Native 视图。此渲染器在 React Native 内部使用。
- [React Test Renderer](#) 将 React 组件渲染为 JSON 树。这用于 Jest 的快照测试特性。在 npm 上作为 `react-test-renderer` 包发布。

另外一个官方支持的渲染器的是 `react-art`。它曾经是一个独立的 [GitHub 仓库](#)，但是现在我们将此加入了主源代码树。

注意:

严格说来，`react-native-renderer` 实现了 React 和 React Native 的连接。真正渲染 Native 视图的平台特代码及组件都存储在 [React Native 仓库](#) 中。

reconcilers

即便 React DOM 和 React Native 渲染器的区别很大，但也需要共享一些逻辑。特别是 [协调](#) 算法需要尽可能相似，这样可以让声明式渲染，自定义组件，state，生命周期方法和 refs 等特性，保持跨平台工作一致。

为了解决这个问题，不同的渲染器彼此共享一些代码。我们称 React 的这一部分为“reconciler”。当处理类似于 `setState()` 这样的更新时，reconciler 会调用树中组件上的 `render()`，然后决定是否进行挂载，更新或是卸载操作。

Reconciler 没有单独的包，因为他们暂时没有公共 API。相反，它们被如 React DOM 和 React Native 的渲染器排除在外。

Stack reconciler

"stack" reconciler 是 React 15 及更早的解决方案。虽然我们已经停止了对它的使用，但是这在[下一章节](#)有详细的文档。

Fiber reconciler

"fiber" reconciler 是一个新尝试，致力于解决 stack reconciler 中固有的问题，同时解决一些历史遗留问题。Fiber 从 React 16 开始变成了默认的 reconciler。

它的主要目标是：

- 能够把可中断的任务切片处理。
- 能够调整优先级，重置并复用任务。
- 能够在父元素与子元素之间交错处理，以支持 React 中的布局。
- 能够在 `render()` 中返回多个元素。

- 更好地支持错误边界。

你可以在[这里](#)和[这里](#)，深入了解 React Fiber 架构。虽然这已经在 React 16 中启用了，但是 `async` 特性还没有默认开启。

源代码在 `packages/react-reconciler` 目录下。

事件系统

React 在原生事件基础上进行了封装，以抹平浏览器间差异。其源码在 `packages/react-dom/src/events` 目录下。

下一章节学习什么？

查看下一章节去学习 reconciler 在 pre-React 16 中的实现。我们还没有为新的 reconciler 内部原理编写文档。

[Go to TOC](#)

组件 & Props

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。本指南旨在介绍组件的相关理念。你可以[参考详细组件 API](#)。

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。

函数组件与 class 组件

定义组件最简单的方式就是编写 JavaScript 函数：

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

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。

你同时还可以使用 ES6 的 `class` 来定义组件：

```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

上述两个组件在 React 里是等效的。

我们将在[下一章节](#)中讨论关于函数组件和 class 组件的额外特性。

渲染组件

之前，我们遇到的 React 元素都只是 DOM 标签：

```
const element = <div />;
```

不过，React 元素也可以是用户自定义的组件：

```
const element = <Welcome name="Sara" />;
```

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”。

例如，这段代码会在页面上渲染“Hello, Sara”：

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

```
const root = ReactDOM.createRoot(document.getElementById('root'));
const element = <Welcome name="Sara" />;
root.render(element);
```

在 CodePen 上试试

让我们来回顾一下这个例子中发生了什么：

1. 我们调用 `root.render()` 函数，并传入 `<Welcome name="Sara" />` 作为参数。
2. React 调用 `Welcome` 组件，并将 `{name: 'Sara'}` 作为 props 传入。
3. `Welcome` 组件将 `<h1>Hello, Sara</h1>` 元素作为返回值。
4. React DOM 将 DOM 高效地更新为 `<h1>Hello, Sara</h1>`。

注意：组件名称必须以大写字母开头。

React 会将以小写字母开头的组件视为原生 DOM 标签。例如，`<div />` 代表 HTML 的 `div` 标签，而 `<Welcome />` 则代表一个组件，并且需在作用域内使用 `Welcome`。

你可以在[深入 JSX](#) 中了解更多关于此规范的原因。

组合组件

组件可以在其输出中引用其他组件。这就可以让我们用同一组件来抽象出任意层次的细节。按钮，表单，对话框，甚至整个屏幕的内容：在 React 应用程序中，这些通常都会以组件的形式表示。

例如，我们可以创建一个可以多次渲染 `Welcome` 组件的 `App` 组件：

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

function App() {
  return (
    <div>
      <Welcome name="Sara" />
      <Welcome name="Cahal" />
      <Welcome name="Edite" />
    </div>
  );
}
```

在 CodePen 上试试

通常来说，每个新的 React 应用程序的顶层组件都是 `App` 组件。但是，如果你将 React 集成到现有的应用程序中，你可能需要使用像 `Button` 这样的小组件，并自下而上地将这类组件逐步应用到视图层的每一处。

提取组件

将组件拆分为更小的组件。

例如，参考如下 `Comment` 组件：

```
function Comment(props) {
  return (
    <div className="Comment">
      <div className="UserInfo">
        <img className="Avatar"
          src={props.author.avatarUrl}
          alt={props.author.name}>
      />
      <div className="UserInfo-name">
        {props.author.name}
      </div>
    </div>
    <div className="Comment-text">
      {props.text}
    </div>
    <div className="Comment-date">
      {formatDate(props.date)}
    </div>
  );
}
```

[在 CodePen 上试试](#)

该组件用于描述一个社交媒体网站上的评论功能，它接收 `author`（对象），`text`（字符串）以及 `date`（日期）作为 props。

该组件由于嵌套的关系，变得难以维护，且很难复用它的各个部分。因此，让我们从中提取一些组件出来。

首先，我们将提取 `Avatar` 组件：

```
function Avatar(props) {
  return (
    <img className="Avatar"
      src={props.user.avatarUrl}
      alt={props.user.name}>
  );
}
```

`Avatar` 不需知道它在 `Comment` 组件内部是如何渲染的。因此，我们给它的 props 起了一个更通用的名字：`user`，而不是 `author`。

我们建议从组件自身的角度命名 props，而不是依赖于调用组件的上下文命名。

我们现在针对 `Comment` 做些微小调整：

```
function Comment(props) {
  return (
    <div className="Comment">
      <div className="UserInfo">
        <Avatar user={props.author} />
        <div className="UserInfo-name">
          {props.author.name}
        </div>
      </div>
    </div>
  );
}
```

```

        </div>
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}

```

接下来，我们将提取 `UserInfo` 组件，该组件在用户名旁渲染 `Avatar` 组件：

```

function UserInfo(props) {
  return (
    <div className="UserInfo">
      <Avatar user={props.user} />
      <div className="UserInfo-name">
        {props.user.name}
      </div>
    </div>
  );
}

```

进一步简化 `Comment` 组件：

```

function Comment(props) {
  return (
    <div className="Comment">
      <UserInfo user={props.author} />
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}

```

[在 CodePen 上试试](#)

最初看上去，提取组件可能是一件繁重的工作，但是，在大型应用中，构建可复用组件库是完全值得的。根据经验来看，如果 UI 中有一部分被多次使用（`Button`，`Panel`，`Avatar`），或者组件本身就足够复杂（`App`，`FeedStory`，`Comment`），那么它就是一个可提取出独立组件的候选项。

Props 的只读性

组件无论是使用[函数声明还是通过 class 声明](#)，都绝不能修改自身的 props。来看下这个 `sum` 函数：

```

function sum(a, b) {
  return a + b;
}

```

这样的函数被称为“[纯函数](#)”，因为该函数不会尝试更改入参，且多次调用下相同的入参始终返回相同的结果。

相反，下面这个函数则不是纯函数，因为它更改了自己的入参：

```
function withdraw(account, amount) {  
  account.total -= amount;  
}
```

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。

当然，应用程序的 UI 是动态的，并会伴随着时间的推移而变化。在[下一章节](#)中，我们将介绍一种新的概念，称之为“state”。在不违反上述规则的情况下，state 允许 React 组件随用户操作、网络响应或者其他变化而动态更改输出内容。

组合 vs 继承

React 有十分强大的组合模式。我们推荐使用组合而非继承来实现组件间的代码重用。

在这篇文档中，我们将考虑初学 React 的开发人员使用继承时经常会遇到的一些问题，并展示如何通过组合思想来解决这些问题。

包含关系

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。

我们建议这些组件使用一个特殊的 `children` prop 来将他们的子组件传递到渲染结果中：

```
function FancyBorder(props) {
  return (
    <div className={'FancyBorder FancyBorder-' + props.color}>
      {props.children}
    </div>
  );
}
```

这使得别的组件可以通过 JSX 嵌套，将任意组件作为子组件传递给它们。

```
function WelcomeDialog() {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">
        Welcome
      </h1>
      <p className="Dialog-message">
        Thank you for visiting our spacecraft!
      </p>
    </FancyBorder>
  );
}
```

[在 CodePen 上尝试](#)

<`FancyBorder`> JSX 标签中的所有内容都会作为一个 `children` prop 传递给 `FancyBorder` 组件。因为 `FancyBorder` 将 `{props.children}` 渲染在一个 `<div>` 中，被传递的这些子组件最终都会出现在输出结果中。

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。

```
function SplitPane(props) {
  return (
    <div className="SplitPane">
      <div className="SplitPane-left">
        {props.left}
      </div>
      <div className="SplitPane-right">
```

```

        {props.right}
      </div>
    </div>
  );
}

function App() {
  return (
    <SplitPane
      left={
        <Contacts />
      }
      right={
        <Chat />
      } />
  );
}

```

[在 CodePen 上尝试](#)

<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。

特例关系

有些时候，我们会把一些组件看作是其他组件的特殊实例，比如 `WelcomeDialog` 可以说是 `Dialog` 的特殊实例。

在 React 中，我们也可以通过组合来实现这一点。“特殊”组件可以通过 props 定制并渲染“一般”组件：

```

function Dialog(props) {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">
        {props.title}
      </h1>
      <p className="Dialog-message">
        {props.message}
      </p>
    </FancyBorder>
  );
}

function WelcomeDialog() {
  return (
    <Dialog
      title="Welcome"
      message="Thank you for visiting our spacecraft!" />
  );
}

```

[在 CodePen 上尝试](#)

组合也同样适用于以 class 形式定义的组件。

```

function Dialog(props) {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">
        {props.title}
      </h1>
      <p className="Dialog-message">
        {props.message}
      </p>
      {props.children}
    </FancyBorder>
  );
}

class SignUpDialog extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.handleSignUp = this.handleSignUp.bind(this);
    this.state = {login: ''};
  }

  render() {
    return (
      <Dialog title="Mars Exploration Program"
        message="How should we refer to you?">
        <input value={this.state.login}
          onChange={this.handleChange} />
        <button onClick={this.handleSignUp}>
          Sign Me Up!
        </button>
      </Dialog>
    );
  }

  handleChange(e) {
    this.setState({login: e.target.value});
  }

  handleSignUp() {
    alert(`Welcome aboard, ${this.state.login}!`);
  }
}

```

[在 CodePen 上尝试](#)

那么继承呢？

在 Facebook，我们在成百上千个组件中使用 React。我们并没有发现需要使用继承来构建组件层次的情况。

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。

如果你想要在组件间复用非 UI 的功能，我们建议将其提取为一个单独的 JavaScript 模块，如函数、对象或者类。组件可以直接引入（import）而无需通过 extend 继承它们。

[Go to TOC](#)

使用 Concurrent 模式(实验性)

注意：

本章节所描述的功能还处于实验阶段，在稳定版本中尚不可用。它面向的人群是早期使用者以及好奇心较强的人。

本页面中许多信息现已过时，仅仅是为了存档而存在。欲了解最新信息，请参阅 [React 18 Alpha 版公告](#)。

在 React 18 发布前，我们将用稳定的文档替代此章节。

- [安装](#)

- [此实验版本适用于谁？](#)

- [开启 concurrent 模式](#)

- [有何期望](#)

- [迁移步骤：blocking 模式](#)

- [为什么有这么多模式？](#)

- [特性对比](#)

安装

concurrent 模式目前仅在 React 的[实验版本](#)可用。安装命令：

```
npm install react@experimental react-dom@experimental
```

实验版本与语义版本不同。 在 `@experimental` 版本，API 会随时增删改。

实验版本经常会包含破坏性更改

你可以在个人项目或新分支中尝试这些实验性功能，但我们并不建议在生产环境中使用它们。在 Facebook，我们确实在生产环境中使用它们，但我们在使用的同时也修复了 bug。**再次强调，请不要在生产环境使用！**

此实验版本适用于谁？

这个版本主要针对功能早期使用者、库作者和对此好奇的人。

我们在生产中使用这段代码，但是仍然有一些 bug、缺少许多特性，且文档存在缺陷。我们希望了解更多关于 concurrent 模式中的出现的问题，以便更好地为未来发布稳定版本做准备。

开启 concurrent 模式

通常，当我们给 React 添加功能的时候，你可以立即使用。比如 Fragments，Context，甚至 Hook。你都可以直接在代码里使用他们，而不用修改之前的代码。

concurrent 模式并不是这样。它给引入了新的语义，改变了 React 的工作方式。否则不能启用[这些新功能](#)。这就是它被分组到了新的模式，而不是相继的发布出来。

你不能为某个子树单独启用 concurrent 模式。你应该在 `ReactDOM.render()` 里启用它。

这会在整个 `<App />` 结构树里启用 concurrent 模式：

```
import ReactDOM from 'react-dom';

// 如果你之前的代码是：
// ReactDOM.render(<App />, document.getElementById('root'));

// 你可以用下面的代码引入 concurrent 模式：

ReactDOM.unstable_createRoot(
  document.getElementById('root')
).render(<App />);
```

注意：

concurrent 模式 API 如 `createRoot` 只存在于 React 实验版本。

在 concurrent 模式下，生命周期[之前被标记过](#)为“不安全”是真的不安全，会比现在的 React 出现更多的 bug。在你的 app 完全兼容[严格模式](#)之前，我们不建议使用 concurrent 模式。

有何期望

如果你有已完成的大型 app，或者你的 app 有大量第三方依赖，请不要立即使用 concurrent 模式。**比如在 Facebook 我们在新网站的开发上使用 concurrent 模式，并没有打算在旧的网站开发上使用。**这是因为我们旧网站的代码里使用了不安全的生命周期方法、不兼容第三方库、与 concurrent 模式也不兼容。

根据我们的经验，使用常见的 React 开发模式，并且不依赖外部状态管理的代码最容易切换到 concurrent 模式。在接下来的几周内，我们会列出常见的问题和解决方案。

迁移步骤：blocking 模式

对于较旧的代码库，concurrent 模式可能步子迈的太大。这就是我们在实验版本中提供“blocking 模式”的原因。你可以通过使用 `createBlockingRoot` 代替 `createRoot` 尝试一下。它仅提供了 concurrent 模式的小部分功能，但它更接近于 React 今天的工作方式，可以作为迁移的一个步骤。

回顾：

- **legacy 模式：** `ReactDOM.render(<App />, rootNode)`。这是当前 React app 使用的方式。当前没有计划删除本模式，但是这个模式可能不支持这些新功能。
- **blocking 模式：** `ReactDOM.createBlockingRoot(rootNode).render(<App />)`。目前正在实验中。作为迁移到 concurrent 模式的第一步。
- **concurrent 模式：** `ReactDOM.createRoot(rootNode).render(<App />)`。目前在实验中，未来稳定之后，打算作为 React 的默认开发模式。这个模式开启了所有的新功能。

为什么有这么多模式？

我们认为提供渐进的迁移策略比进行破坏性的更改或者使 React 停滞不前是更好的选择。

实际上，我们希望今天使用 legacy 模式的大多数 app 至少能迁移到 blocking 模式（如果不能迁移到 concurrent 模式）。对于希望在短期内支持所有模式的库而言，碎片化可能是很讨厌的事情。但是组件将生态系统从 legacy 模式中移除，也会解决一些影响 React 主要库的问题。比如获取布局时令人迷惑的 Suspense 行为和缺乏一致性的批处理。legacy 模式下，如果不修改语义就无法修复的许多错误，在 blocking 模式和 concurrent 模式下就不存。

你可以把 blocking 模式当作 concurrent 模式的“优雅降级”版本。所以长远来看，模式的数量会收敛，不用考虑不同的模式。但就目前而言，模式是一项重要的迁移策略。能让每个人都能决定自己什么时候迁移，并按照自己的速度进行迁移。

特性对比

legacy 模式	blocking 模式	concurrent 模式	---	---	---		String Refs				Legacy Context
							Suspense				Suspense
SSR + Hydration				Progressive Hydration				Selective Hydration			Cooperative Multitasking
											Automatic batching of multiple setStates
									*		Priority-based Rendering
											Interruptible Prerendering
											useTransition
											useDeferredValue
											Suspense Reveal "Train"

* : legacy 模式在合成事件中有自动批处理的功能，但仅限于一个浏览器任务。非 React 事件想使用这个功能必须使用 `unstable_batchedUpdates`。在 blocking 模式和 concurrent 模式下，所有的 `setState` 在默认情况下都是批处理的。

** : 会在开发中发出警告。

Concurrent 模式介绍 (实验性)

注意：

本章节所描述的功能还处于实验阶段，在稳定版本中尚不可用。它面向的人群是早期使用者以及好奇心较强的人。

本页面中许多信息现已过时，仅仅是为了存档而存在。欲了解最新信息，请参阅 [React 18 Alpha 版公告](#)。

在 React 18 发布前，我们将用稳定的文档替代此章节。

本页面提供了 Concurrent 模式的理论概述。有关更多实用性的介绍，你可能需要查看以下部分：

- [Suspense 用于数据获取](#) 描述了一种在 React 组件中获取数据的新机制。
- [Concurrent UI Patterns](#) 展示了一些通过 Concurrent 模式和 Suspense 实现的 UI 模式。
- [采用 Concurrent 模式](#) 解释了如何在你的项目中尝试 Concurrent 模式。
- [Concurrent 模式的 API 索引](#) 记录了在实验性版本中可用的新 API。

什么是 Concurrent 模式？

Concurrent 模式是一组 React 的新功能，可帮助应用保持响应，并根据用户的设备性能和网速进行适当的调整。

这些功能尚处于试验阶段，可能会发生改变。它们还不是稳定的 React 版本中的一部分，但是你可以在实验版本中尝试它们。

阻塞 vs 可中断渲染

为了解释 Concurrent 模式，我们将使用版本控制作为比喻。如果你在团队中工作，你可能使用了像 Git 这样的版本控制系统并在分支上进行工作。当一个分支准备就绪时，你可以将你的工作合并到 main 中，以便他人拉取。

在版本控制存在之前，开发工作流程有很大的不同。不存在分支的概念。如果你想编辑某些文件，你必须告诉所有人在你完成编辑工作之前不要触碰这些文件。你甚至不能同时和那个人研究它们——实际上，你被它们 阻塞 了。

这说明了包括 React 在内的 UI 库在目前通常是如何工作的。一旦它们开始渲染一次更新，它们不能中断包括创建新的 DOM 节点和运行组件中代码在内的工作。我们称这种方法为“阻塞渲染”。

在 Concurrent 模式中，渲染不是阻塞的。它是可中断的。这改善了用户体验。它同时解锁了以前不可能的新功能。在我们查看[下一个章节](#)的具体例子之前，我们将对新功能做一个高级的概述。

可中断渲染

考虑一个可过滤的产品列表。你是否曾在一个列表筛选器中输入过，且每一次输入都感觉到并不流畅？一些更新产品列表的工作是不可避免的，例如创建新的 DOM 节点或者浏览器执行布局。然而，我们 何时 以及 如何 处理这项工作起着很大的作用。

解决卡顿的一种常见方法是对输入进行“防抖”处理。防抖时，我们只在用户停止输入之后更新列表。然而，令人沮丧的是，在我们键入的时候不会进行更新。作为一种替代，我们可以对输入进行“节流”，并以一定的最大频率更新列表。但是在功率较低的设备上，还是会发卡顿现象。无论防抖还是节流都不会提供最佳的用户体验。

产生卡顿的原因很简单：一旦渲染开始，就不能被终止。因此浏览器不能在按键结束后立即更新。无论 UI 库(如 React)在基准测试中表现得多么出色，只要它使用阻塞渲染，组件中总会有一定数量的工作导致卡顿。并且，通常没有简单的解决办法。

Concurrent 模式通过使渲染可中断来修复此基本限制。这意味着当用户按下另一个按键时，React 不需要阻塞浏览器更新文本输入。相反，它可以让浏览器绘制输入的更新，然后在内存中渲染更新后的列表。当渲染完成后，React 更新 DOM，并且变化会反映在屏幕上。

从概念上讲，你可以将它视为 React “在分支上”准备每一次更新。就像你可以放弃分支上的工作或者在它们之间切换一样，React 在 Concurrent 模式中可以中断一项正在执行的更新去做一些更重要的事情，然后再回到之前正在做的工作。这项技术也许会使你想起电子游戏中的**双重缓冲**。

Concurrent 模式减少了防抖和节流在 UI 中的需求。因为渲染是可以中断的，React 不需要人为地延迟工作以避免卡顿。它可以立即开始渲染，但是当需要保持应用响应时中断这项工作。

有意的加载顺序

我们之前说过 Concurrent 模式就像 React 工作“在分支上”。分支不仅对短期修复有用，对长期的功能开发也很有用。有时你可能会开发某项功能，但是在它达到一个“足够好的状态”以合并到 main 之前，往往需要好几周的时间。我们的版本控制比喻在这一方面同样适用于渲染。

想象一下，我们正在应用的两个屏幕之间导航。有时，我们可能没有加载足够的代码和数据在新屏幕上向用户展示“足够好”的加载状态。这样过渡到一个空白屏或者大型的轮播图会是一个不愉快的体验。然而，通常获取所需的代码和数据不会花费太长时间。**如果 React 可以在旧屏幕上多停留一段时间，并在展示新屏幕之前“跳过”“不够好的加载状态”，不是更好吗？**

虽然这在当前是可以实现的，但是协调起来会有些困难。在 Concurrent 模式中，这些功能是内置的。React 首先在内存中准备新屏幕 — 或者，用我们比喻的说法，“在不同的分支上”。所以 React 可以在更新 DOM 之前进行等待，以便加载更多内容。在 Concurrent 模式中，我们可以让 React 继续显示完全互动，带有内联加载指示器的旧屏幕。当新屏幕准备就绪之后，React 可以带我们跳转到新屏幕。

并发

让我们回顾一下上面的两个例子然后看一下 Concurrent 模式是如何将它们联合起来的。**在 Concurrent 模式中，React 可以同时更新多个状态** —— 就像分支可以让不同的团队成员独立地工作一样：

- 对于 CPU-bound 的更新 (例如创建新的 DOM 节点和运行组件中的代码)，并发意味着一个更急迫的更新可以“中断”已经开始的渲染。
- 对于 IO-bound 的更新 (例如从网络加载代码或数据)，并发意味着 React 甚至可以在全部数据到达之前就在内存中开始渲染，然后跳过令人不愉快的空白加载状态。

重要的是，你 使用 React 的方式是相同的。components, props, 和 state 等概念的基本工作方式是相同的。当你想更新屏幕，设置 state 即可。

React 使用一种启发式方法决定更新的“紧急性”，并且允许你用几行代码对其进行调整，以便你可以在每次交互中实现理想的用户体验。

将研究投入生产

围绕 Concurrent 模式有一个共同的主题。它的任务是帮助将人机交互研究的结果整合到真实的 UI 中。

例如，研究表明，在屏幕之间切换时显示过多的中间加载状态会使切换的速度 变慢。这就是为什么 Concurrent 模式在一个固定的“时间表”上显示新的加载状态，用于避免不愉快的和过多的更新。

类似的，我们从研究得知悬停和文本输入之类的交互需要在很短的时间内处理，而点击和页面转换可以等待稍长时间而不会感到迟缓。Concurrent 模式在内部使用不同的“优先级”，大致对应于人类感知研究中的交互类别。

专注于用户体验的团队有时会通过一次性解决方案来解决类似的问题。然而，这些解决方案难以维护所以很少能长期存活。使用 Concurrent 模式，我们的目标是将 UI 的研究结果纳入抽象本身，并提供使用它们的惯用方法。作为一个 UI 库，React 很适合这样做。

下一步

现在你已经知道 Concurrent 模式是什么了！

在之后的页面中，你将学习更多特定主题的详细信息：

- [Suspense 用于数据获取](#) 描述了一种在 React 组件中获取数据的新机制。
- [Concurrent UI Patterns](#) 展示了一些通过 Concurrent 模式和 Suspense 实现的 UI 模式。
- [采用 Concurrent 模式](#) 解释了如何在你的项目中尝试 Concurrent 模式。
- [Concurrent 模式的 API 索引](#) 记录了在实验性版本中可用的新 API。

Concurrent UI 模式 (试验阶段)

注意：

本章节所描述的功能还处于实验阶段，在稳定版本中尚不可用。它面向的人群是早期使用者以及好奇心较强的人。

本页面中许多信息现已过时，仅仅是为了存档而存在。欲了解最新信息，请参阅 [React 18 Alpha 版公告](#)。

在 React 18 发布前，我们将用稳定的文档替代此章节。

通常，当我们更新 state 的时候，我们会期望这些变化立刻反映到屏幕上。我们期望应用能够持续响应用户的输入，这是符合常理的。但是，有时我们会期望 **更新延迟响应在屏幕上**。

举个例子，假如我们从一个页面切换到另一个页面，但是在下一页中代码和数据还没有加载好，则会看到空白页，并显示着加载中的加载指示器，这会让人很不舒服。这种情况下我们可能更希望在前一个页面多停留一会儿。在 React 中实现这个功能在之前是很难做到的。Concurrent 模式提供了一系列的新工具使之成为可能。

- [Transition](#)
 - [用 Transition 包裹 setState](#)
 - [添加一个等待提示器](#)
 - [回顾更改](#)
 - [是在那里更新的？](#)
 - [很多场景可以使用 transition](#)
 - [把 Transition 融合到你应用的设计系统](#)
- [3个阶段](#)
 - [默认方式：Receded → Skeleton → Complete](#)
 - [期望方式：Pending → Skeleton → Complete](#)
 - [使用 `<Suspense>` 包裹惰性功能](#)
 - [Suspense 更新“列车”](#)
 - [延迟显示等待提示](#)
 - [回顾](#)
- [其他模式](#)
 - [根据优先级分割 state](#)
 - [延迟一个值](#)
 - [SuspenseList](#)
- [下一步](#)

Transition

我们先来回顾一下前一篇关于 [Suspense 用于数据获取](#) 文章中的 [这个示例](#)。

当我们点击 "Next" 按钮来切换激活的页面，现存的页面立刻消失了，然后我们看到整个页面只有一个加载提示。可以说这是一个“不受欢迎”的加载状态。**如果我们可以“跳过”这个过程，并且等到内容加载后再过渡到新的页面，效果会更好**

React 提供了一个新的内置的 `useTransition()` 的 Hook 可以实现这个设计。

我们通过 3 个步骤来实现它。

首先，要确保项目中正在使用 Concurrent 模式。我们会在稍后讨论如何 [采用 Concurrent 模式](#)，但是就现在而言，我们要让这个特性工作只要知道需要使用 `ReactDOM.createRoot()` 而非 `ReactDOM.render()` 就足够了：

```
const rootElement = document.getElementById("root");
// 进入 Concurrent 模式
ReactDOM.createRoot(rootElement).render(<App />);
```

接下来，我们需要增加一个从 React 引入 `useTransition` Hook 的 import：

```
import React, { useState, useTransition, Suspense } from "react";
```

最后，我们在 `App` 组件中使用它：

```
function App() {
  const [resource, setResource] = useState(initialResource);
  const [startTransition, isPending] = useTransition({
    timeoutMs: 3000
  });
  // ...
}
```

就这段代码而言，**它还什么都做不了**。我们需要使用这个 Hook 的返回值来配置我们的界面切换。`useTransition` 包含两个返回值：

- `startTransition` 类型为函数。我们用它来告诉 React 我们希望的延迟的是哪个 state 的更新。
- `isPending` 类型为 boolean。此变量在 React 中用于告知我们该转换是否正在进行。

接下来我们就会用到它们。

注意我们给 `useTransition` 传入了一个配置对象。此对象包含 `timeoutMs` 属性，该属性指定了**我们希望这个转换在多久之内完成**。通过传入了配置 `{timeoutMs: 3000}`，就等同于告诉 React “如果下一个页面需要 3 秒钟以上才能加载好，就展示加载指示器——但是在那之前，我们先显示前一个界面”。

用 Transition 包裹 setState

我们的 "Next" 按钮的点击事件处理器能够引起触发切换页面的 state 更新：

```
<button
  onClick={() => {
    const nextUserId = getNextId(resource.userId);
    setResource(fetchProfileData(nextUserId));
  }}
>
```

我们把这个 state 更新包裹在 `startTransition` 中。这就是我们通知 React 如果它会产生不受欢迎的加载中界面 **我们希望 React 延迟更新此 state**：

```
<button
  onClick={() => {
    startTransition(() => {
```

```

    const nextUserId = getNextId(resource.userId);
    setResource(fetchProfileData(nextUserId));
  });
}
>

```

在 CodeSandbox 中尝试

试试点击 "Next" 几下。注意它的体验已经很不一样了。当点击时，我们没有直接切换到一个空白的页面，而是在前一个页面停留了一段时间。当数据加载好的时候 React 会帮我们切换到新的界面。

如果我们把 API 接口的响应时间调整到 5 秒钟，[我们可以确认](#) React “放弃”并在 3 秒后转换到了新的页面。这是因为我们给 `useTransition()` 传入的配置 `{timeoutMs: 3000}`。假如，我们传入的是 `{timeoutMs: 60000}` 那么它会等上整整一分钟。

添加一个等待提示器

在 [我们前一个例子](#) 中还是有地方体验不友好。最好不要显示加载中。**但是如果没有这个过程提示的话体验会更糟糕！**当我们点击 "Next"按钮，什么都没有发生，就好像整个应用卡死一样。

调用 `useTransition()` 包含两个值返回值：`startTransition` 和 `isPending`。

```
const [startTransition, isPending] = useTransition({ timeoutMs: 3000 });
```

我们已经使用了 `startTransition` 来包裹 state 更新。现在我们要使用 `isPending` 了。React 提供了这个布尔值来告诉我们当前**我们是否正在等待界面切换完成**。我们会用它来指示是不是有什么事情正在发生：

```

return (
  <>
    <button
      disabled={isPending}
      onClick={() => {
        startTransition(() => {
          const nextUserId = getNextId(resource.userId);
          setResource(fetchProfileData(nextUserId));
        });
      }}
    >
      Next
    </button>
    {isPending ? " Loading..." : null}
    <ProfilePage resource={resource} />
  </>
);

```

在 CodeSandbox 中尝试

现在，这感觉好多了！当我们点击 Next 按钮的时候，它变得不可用，因为点击它很多次并没有意义。而且新增的“Loading...”提示让用户知道程序并没有卡住。

回顾更改

我们来回顾基于 [原始示例](#) 做出的所有更改：

```

function App() {
  const [resource, setResource] = useState(initialResource);
  const [startTransition, isPending] = useTransition({
    timeoutMs: 3000
  });
  return (
    <>
      <button
        disabled={isPending}
        onClick={() => {
          startTransition(() => {
            const nextUserId = getNextId(resource.userId);
            setResource(fetchProfileData(nextUserId));
          });
        }}
      >
        Next
      </button>
      {isPending ? " Loading..." : null}
      <ProfilePage resource={resource} />
    </>
  );
}

```

[在 CodeSandbox 中尝试](#)

我们只用了 7 行代码来实现这个切换：

- 我们引入了 `useTransition` Hook 并在更新 state 的组件中使用了它。
- 我们传入了 `{timeoutMs: 3000}` 使得前一个页面在屏幕上最多保持3秒钟。
- 我们把 state 更新包裹在 `startTransition` 中，以通知 React 可以延迟这个更新。
- 我们使用 `isPending` 来告诉用户界面切换的进展并禁用按钮。

最后的结果是，点击“Next”按钮不会立刻切换界面并展示加载中，而是停留在前一个界面并同步加载进度。

是在哪里更新的？

这并不是很难实现。但是，如果你已经开始思考这是如何工作的，它可能会让你感到费解。既然我们更新了 state，为什么我们不能立刻看到结果呢？而下一个 `<ProfilePage>` 又是在哪里渲染的呢？

很显然，两个“版本”的 `<ProfilePage>` 同时存在了。我们知道旧的存在是因为它在界面上，而且它还显示了一个进度提示。我们也知道新的存在于某个地方，因为它就是我们正在等待的那个界面！

但是同一个组件的两个版本的是如何同时存在的呢？

这原因就在于 Concurrent 模式本身。我们 [之前提到](#) 它有点像在“branch”上运行的一个 state 更新。或者我们可以想象成，当我们把 state 更新包裹在 `startTransition` 的时候会在“另一个宇宙中”开始渲染，就像科幻电影一样。我们并不能直接看到那个宇宙 -- 但是我们能够从那个宇宙探知一些事情正在发生的事情 (`isPending`)。当更新完成的时候，我们的“多个宇宙”合并成一个，我们在屏幕上看到最终的结果！

在 [示例](#) 中多练习一下，然后试着想象它正在发生。

当然，两个版本的树同时渲染只是个假象，正如所有程序同时在你电脑上运行的想法也同样是假象。操作系统会在不同的应用之间快速的切换。类似的，React 可以在不同版本的树上进行切换，一个是你屏幕上看到的那个版本，另一个是它“准备”接下来给你显示的版本。

一个 `useTransition` 这样的 API 可以让你专注于期望的用户体验，而不需要思考这些机制是如何实现的。仍然，想象 `startTransition` 包裹的更新是“在一个分支上”或者“在另一个世界”发生的，是个很有帮助的比喻。

很多场景可以使用 transition

正如我们从 [Suspense 简介](#) 中所了解，所有所需数据没有准备好的组件都可以“suspend”一段时间。我们可以从策略上用 `<Suspense>` 把树的不同部分圈起来处理，但这并不总是足够的。

我们回到 [第一个 Suspense 示例](#) 那时还是只有一个界面的。现在我们增加一个“Refresh”按钮，用来检查服务端的数据更新。

我们的第一次尝试大概看起来是这样的：

```
const initialResource = fetchUserAndPosts();

function ProfilePage() {
  const [resource, setResource] = useState(initialResource);

  function handleRefreshClick() {
    setResource(fetchUserAndPosts());
  }

  return (
    <Suspense fallback={<h1>Loading profile...</h1>}>
      <ProfileDetails resource={resource} />
      <button onClick={handleRefreshClick}>
        Refresh
      </button>
      <Suspense fallback={<h1>Loading posts...</h1>}>
        <ProfileTimeline resource={resource} />
      </Suspense>
    </Suspense>
  );
}
```

在 CodeSandbox 中尝试

在这个例子中，我们会在加载和每次点击“Refresh”按钮的时候开始数据获取。我们把 `fetchUserAndPosts()` 的结果放到 `state` 中，这样下级的组件可以从我们刚刚发起的请求中读取新的数据。

我们可以看到在[示例](#)中点击“Refresh”是可以正常工作的。`<ProfileDetails>` 和 `<ProfileTimeline>` 组件接收代表新数据的 `resource` prop，它会因为我们尚未得到服务端响应而“suspend”，所以我们看到了降级方案界面。当服务端响应加载完成，我们看到更新后的文章（我们的伪造接口每 3 秒增加一些文章）。

然而，这种交互体验极差。用户正在浏览页面，但是在与页面进行交互的时候，内容却被加载状态覆盖。这会让人感觉匪夷所思。**正如前面那样，要避免显示加载中，我们把 state 更新放到 transition 中：**

```
function ProfilePage() {
  const [startTransition, isPending] = useTransition({
    // Wait 10 seconds before fallback
```

```

    timeoutMs: 10000
  );
  const [resource, setResource] = useState(initialResource);

  function handleRefreshClick() {
    startTransition(() => {
      setResource(fetchProfileData());
    });
  }

  return (
    <Suspense fallback={<h1>Loading profile...</h1>}>
      <ProfileDetails resource={resource} />
      <button
        onClick={handleRefreshClick}
        disabled={isPending}
      >
        {isPending ? "Refreshing..." : "Refresh"}
      </button>
      <Suspense fallback={<h1>Loading posts...</h1>}>
        <ProfileTimeline resource={resource} />
      </Suspense>
    </Suspense>
  );
}

```

[在 CodeSandbox 中尝试](#)

这下感觉好多了！点击“Refresh”按钮不会再阻断页面浏览了。我们会看到有内容正在“内联”加载，并且当数据准备好，它就显示出来了。

把 `Transition` 融合到你应用的设计系统

`useTransition` 是非常常见的需求。几乎所有可能导致组件挂起的点击或交互操作都需要使用 `useTransition`，以避免意外隐藏用户正在交互的内容。

这可能会导致组件存在大量重复代码。这正是我们通常建议把 `useTransition` 融合到你应用的设计系统组件中的原因。例如，我们可以把 `transition` 逻辑抽取到我们自己的 `<Button>` 组件中：

```

function Button({ children, onClick }) {
  const [startTransition, isPending] = useTransition({
    timeoutMs: 10000
  });

  function handleClick() {
    startTransition(() => {
      onClick();
    });
  }

  const spinner = (
    // ...
  );

  return (
    <>
      <button
        onClick={handleClick}
        disabled={isPending}
      >

```

```

    >
      {children}
    </button>
    {isPending ? spinner : null}
  </>
);
}

```

在 CodeSandbox 中尝试

需要注意按钮并不关心我们会更新什么。它把发生在它 `onClick` 处理器过程中的任意 state 更新包装到一个 transition 中。这样我们的 `<Button>` 组件来管理 transition 的配置，而 `<ProfilePage>` 组件不在需要单独配置：

```

function ProfilePage() {
  const [resource, setResource] = useState(initialResource);

  function handleRefreshClick() {
    setResource(fetchProfileData());
  }

  return (
    <Suspense fallback={<h1>Loading profile...</h1>}>
      <ProfileDetails resource={resource} />
      <Button onClick={handleRefreshClick}>
        Refresh
      </Button>
      <Suspense fallback={<h1>Loading posts...</h1>}>
        <ProfileTimeline resource={resource} />
      </Suspense>
    </Suspense>
  );
}

```

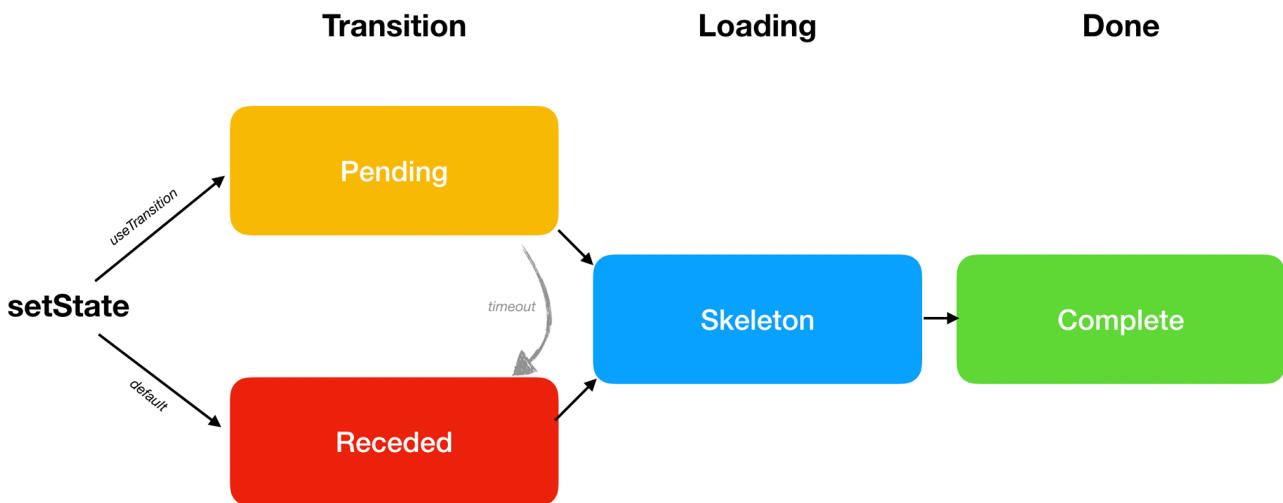
在 CodeSandbox 中尝试

当一个按钮点击的时候，它开启一个 transition 并在 transition 内部调用 `props.onClick()` —— 这会触发 `<ProfilePage>` 组件中的 `handleRefreshClick`。我们开始获取最新数据，但这并不会触发一个降级界面，因为我们正运行在 transition 中，并且 `useTransition` 调用时指定的 10 秒钟尚未达到。当一个 transition 等待的时候，这个按钮会内联显示加载中的提示。

我们现在可以看出 Concurrent 模式能够帮助我们在不牺牲组件的独立性和模块性的同时达成更好的用户体验。由 React 来协调 transition。

3 个阶段

至此我们已经讨论了更新时可能经历的所有不同的显示状态。在这一节中，我们会给它们命名并讨论它们之间的关联。



在最后，我们达到 **Complete** (完成) 状态。那是我们最终想要达到的状态。它代表着下一个界面完全渲染并且不再加载新数据的时刻。

但是在屏幕完全展示之前，我们可能需要加载一些数据或代码。当我们已经在下一个界面，但它的某些部分还在加载中，我们称此状态为 **Skeleton** (骨架)。

最后，有两种主要方式可以使我们进入骨架状态。我们会通过具体的示例详细描述它们之间的区别。

默认方式: Receded → Skeleton → Complete

打开[此示例](#)并点击“Open Profile”。你会陆续看到几种显示状态：

- **Receded (后退)**：第一秒，你会看到 `<h1>Loading the app...</h1>` 降级界面。
- **Skeleton**：你会看到 `<ProfilePage>` 组件中显示着 `<h2>Loading posts...</h2>`。
- **Complete**：你会看到 `<ProfilePage>` 组件不再显示降级界面。所有内容获取完毕。

我们如何区分 Receded 和 Skeleton 状态呢？它们之间的区别在于 **Receded** 感觉像是面向用户“向后退一步”，而 **Skeleton** 模式感觉像是在我们的进程中“向前走一步”来展示更多的内容。

在此示例中，我们从 `<HomePage>` 开始我们的旅程：

```

<Suspense fallback={...}>
  {/* previous screen */}
  <HomePage />
</Suspense>
  
```

点击之后，React 开始渲染下一个界面：

```

<Suspense fallback={...}>
  {/* next screen */}
  <ProfilePage>
    <ProfileDetails />
    <Suspense fallback={...}>
      <ProfileTimeline />
    </Suspense>
  </ProfilePage>
</Suspense>
  
```

```
</Suspense>
</ProfilePage>
</Suspense>
```

`<ProfileDetails>` 和 `<ProfileTimeline>` 都需要数据来渲染，所以它们将被 suspend：

```
<Suspense fallback={...}>
  {/* next screen */}
  <ProfilePage>
    <ProfileDetails /> {/* suspends! */}
    <Suspense fallback={<h2>Loading posts...</h2>}>
      <ProfileTimeline /> {/* suspends! */}
    </Suspense>
  </ProfilePage>
</Suspense>
```

当组件被 suspend 时，React 需要显示最近的那个降级界面。但是对 `<ProfileDetails>` 来说最近的降级界面就已经是最顶层了：

```
<Suspense fallback={
  // 我们现在看到这个降级界面是由 <ProfileDetails> 导致
  <h1>Loading the app...</h1>
}>
  {/* next screen */}
  <ProfilePage>
    <ProfileDetails /> {/* suspends! */}
    <Suspense fallback={...}>
      <ProfileTimeline />
    </Suspense>
  </ProfilePage>
</Suspense>
```

这就是当我们点击按钮之后，它感觉像是我们“后退了一步”。`<Suspense>` 范围本来显示的有用内容 (`<HomePage />`) 必须“后退”并显示降级界面 (`<h1>Loading the app...</h1>`)。我们称之为**Receded (后退)** 状态。

当我们加载了更多内容的时候，React 会重新尝试渲染，这时 `<ProfileDetails>` 能够成功渲染。最终，我们进入 **Skeleton** 状态。我们看到了尚未完全渲染的新的页面。

```
<Suspense fallback={...}>
  {/* 下一屏 */}
  <ProfilePage>
    <ProfileDetails />
    <Suspense fallback={
      // 我们看到此降级界面是由 <ProfileTimeline> 导致
      <h2>Loading posts...</h2>
    }>
      <ProfileTimeline /> {/* suspends! */}
    </Suspense>
  </ProfilePage>
</Suspense>
```

最终，它们也加载好了，然后我们达到 **Complete** 状态

这个剧情 (Receded → Skeleton → Complete) 是默认情况。但是 Receded 状态是非常不友好的，因为它“隐藏”了已经存在的信息。这正是 React 让我们通过 `useTransition` 进入另一个序列 (**Pending (等待)** → Skeleton → Complete) 的原因。

期望方式: Pending → Skeleton → Complete

当我们使用 `useTransition` 的时候, React 会让我们“停留”在前一个页面 -- 并在那显示一个进度提示。我们称它为一个 **Pending (暂停)** 状态。这种的体验会比 Receded 状态好很多, 因为我们已经显示的信息不会消失, 而且页面仍可以交互。

你可以对比这两个例子来体验其中的差异 :

- 默认方式: `Receded → Skeleton → Complete`
- 期望方式: Pending → Skeleton → Complete**

这两个例子唯一的不同就在于第一个使用的是普通 `<button>`, 而第二个使用的是我们使用 `useTransition` 定制的 `<Button>` 组件。

使用 `<Suspense>` 包裹惰性功能

打开 [这个例子](#)。当你点击一个按钮时, 你会先看到一个持续1秒的 Pending 状态再继续。这个 transition 体验很好而且流畅。

我们现在需要给详情页面增加一个新特性 -- 某人的趣闻列表

```
function ProfilePage({ resource }) {
  return (
    <>
      <ProfileDetails resource={resource} />
      <Suspense fallback={<h2>Loading posts...</h2>}>
        <ProfileTimeline resource={resource} />
      </Suspense>
      <ProfileTrivia resource={resource} />
    </>
  );
}

function ProfileTrivia({ resource }) {
  const trivia = resource.trivia.read();
  return (
    <>
      <h2>Fun Facts</h2>
      <ul>
        {trivia.map(fact => (
          <li key={fact.id}>{fact.text}</li>
        ))}
      </ul>
    </>
  );
}
```

在 [CodeSandbox](#) 中尝试

如果你现在点击 “Open Profile” 按钮, 你会发现哪里不对劲。它现在要等待整整 7 秒钟才能完成这个 transition ! 这是因为我们琐碎的 API 接口响应太慢。假设我们没有办法让这个接口变快。在这个约束条件下我们该如何提升用户体验呢 ?

如果我们不想在 Pending 状态等待太久, 我们第一直觉应该是调整 `useTransition` 中 `timeoutMs` 参数到一个更小的值, 比如 `3000`。你可以体验一下 [这个](#)。这样我们就不用在 Pending 状态长时间等待了, 但是这个时候我们还没有有用的内容给用户展示 !

还有一种简单的方法可以解决这个问题。与其把 `transition` 时间缩短，我们不如把慢组件从 `transition` “分离”出来，通过把它封装在 `<Suspense>` 中：

```
function ProfilePage({ resource }) {
  return (
    <>
      <ProfileDetails resource={resource} />
      <Suspense fallback={<h2>Loading posts...</h2>}>
        <ProfileTimeline resource={resource} />
      </Suspense>
      <Suspense fallback={<h2>Loading fun facts...</h2>}>
        <ProfileTrivia resource={resource} />
      </Suspense>
    </>
  );
}
```

[在 CodeSandbox 中尝试](#)

这揭示了一个重要的信息。React 总是倾向尽早的进入 `Skeleton` 状态。即使我们总是在 `transition` 中使用很长的超时时间设置，React 为了避免 `Receded` 状态也不会在 `Pending` 状态停留多余的时间。

如果有些特性并不是下个界面所必要的部分，用 `<Suspense>` 把它包裹起来使其惰性加载。这样就确保了我们能够尽可能快的给用户显示其他的内容，相反的，如果有些组件缺少了这个界面就不值得显示，例如我们例子中的 `<ProfileDetails>`，就不要用 `<Suspense>` 包裹它。这样 `transition` 就会“等待”直到它准备好。

Suspense 更新“列车”

当我们已经到达下一个界面，有的时候用来“解锁”不同 `<Suspense>` 区域的数据在短时间内陆续到达。例如，两个请求响应分别在 1000ms 和 1050ms 到达。如果你已经等待了1秒钟，再等 50ms 也并不容易察觉。这就是 React 在一个计划中显示 `<Suspense>` 区域的原因，就像一趟间隔指定时间到达的“列车”一样。这样就可以用一个较低的延迟换取展示给用户更少的布局变化和视觉变化次数。

你可以在 [这里](#) 查看这个示例。“文章”和“趣闻”响应相差了 100ms。但是 React 把他们的变化合并在一起并同时更新了他们的 `Suspense` 的区域。

延迟显示等待提示

当我们点击 `Button` 组件时它会立刻显示一个 `Pending` 状态提示：

```
function Button({ children, onClick }) {
  const [startTransition, isPending] = useTransition({
    timeoutMs: 10000
  });
  // ...

  return (
    <>
      <button onClick={handleClick} disabled={isPending}>
        {children}
      </button>
      {isPending ? spinner : null}
    </>
  );
}
```

```

    </>
  );
}

```

在 CodeSandbox 中尝试

这让用户知道有什么事情正在发生。但是，如果这个 transition 过程相对较短的时候（小于 500ms），它有可能过于分散注意力，并且使得 transition 本身感觉上更慢。

一个可能的方法就是延迟等待提示的显示：

```

.DelayedSpinner {
  animation: 0s linear 0.5s forwards makeVisible;
  visibility: hidden;
}

@keyframes makeVisible {
  to {
    visibility: visible;
  }
}

const spinner = (
  <span className="DelayedSpinner">
    {/* ... */}
  </span>
);

return (
  <>
    <button onClick={handleClick}>{children}</button>
    {isPending ? spinner : null}
  </>
);

```

在 CodeSandbox 中尝试

通过这个更改，即使我们进入了 Pending 状态，在 500ms 过去之前我们都不会给用户显示任何提示。这对于一些 API 响应较慢的情况算不上是很大的改进。但是在 API 响应快的情况下对比感受下 [使用前](#) 和 [使用后](#)。即使其他的代码并没有更改，通过不在延迟上吸引用户注意，隐藏掉“过快”的加载状态以达到提升感官体验。

回顾

到此我们所学的最重要的东西如下：

- 默认情况下，我们的加载顺序是 Receded → Skeleton → Complete.
- Receded 状态的体验并不友好，因为它会隐藏内容。
- 通过 `useTransition`，我们可以切换到首先进入 Pending 状态而非 Receded。这让我们可以在下一个界面准备好之前停留在前一个页面上。
- 如果我们不想因为一部分组件延迟整个 transition，我们可以把他们各自用 `<Suspense>` 区域包裹起来。
- 与其在每个不同组件中使用 `useTransition`，我们不如把它组织到系统设计中。

其他模式

Transition 大概是你能遇到的最常见的 Concurrent 模式了，但是还有一些其他的模式你有可能用得着。

根据优先级分割 state

当你设计 React 组件的时候，找到 state 的“极小表示法”通常是最好的方式。例如，与其在 state 中保存 `firstName`、`lastName` 和 `fullName`，不如只保存 `firstName` 和 `lastName` 这样通常会更好，然后在渲染时通过计算得到 `fullName`。这可以避免我们只更新了某个 state 却忘记了更新关联 state 所导致的错误。

然而，在 Concurrent 模式中有些情况下你会想要把一些数据冗余到不同的 state 变量中。请考虑这个小 translation 应用：

```
const initialQuery = "Hello, world";
const initialResource = fetchTranslation(initialQuery);

function App() {
  const [query, setQuery] = useState(initialQuery);
  const [resource, setResource] = useState(initialResource);

  function handleChange(e) {
    const value = e.target.value;
    setQuery(value);
    setResource(fetchTranslation(value));
  }

  return (
    <>
      <input
        value={query}
        onChange={handleChange}
      />
      <Suspense fallback={<p>Loading...</p>}>
        <Translation resource={resource} />
      </Suspense>
    </>
  );
}

function Translation({ resource }) {
  return (
    <p>
      <b>{resource.read()}</b>
    </p>
  );
}
```

[在 CodeSandbox 中尝试](#)

请注意当我们在输入框打字的时候，`<Translation>` 组件式如何 suspend 的，并且我们会在得到新的结果之前看到 `<p>Loading...</p>` 这个降级界面。这并不理想。当我们在获取下一个翻译的同时如果我们可以多看一会上一个翻译效果应该会更好。

事实上，如果我们打开控制台，我们会看到一则警告：

Warning: App triggered a user-blocking update that suspended.
 # Warning: App 触发了一个会 suspend 的 user-blocking 更新。

The fix is to split the update into multiple parts: a user-blocking update to provide immediate feedback, and another update that triggers the bulk of the changes.

修复方法是把更新分离到不同的部分：一个用于提供直接反馈的 user-blocking 更新，和一个用于触发主体变化的更新。

Refer to the documentation for `useTransition` to learn how to implement this pattern.

参考 `useTransition` 的文档来了解如何实现这个模式。

正如我们之前提到的，如果一个 state 更新会导致一个组件 suspend，那么这个 state 更新就应该用 transition 包裹起来。我们来把 `useTransition` 添加到我们的组件中：

```
function App() {
  const [query, setQuery] = useState(initialQuery);
  const [resource, setResource] = useState(initialResource);
  const [startTransition, isPending] = useTransition({
    timeoutMs: 5000
  });

  function handleChange(e) {
    const value = e.target.value;
    startTransition(() => {
      setQuery(value);
      setResource(fetchTranslation(value));
    });
  }

  // ...
}
```

在 CodeSandbox 中尝试

现在尝试往输入框里敲字吧。有点不对劲！输入框的更新非常慢。

我们解决了第一个问题（没有使用 transition 的 suspend）。但是因为这个 transition，我们的 state 无法立刻更新，而且无法“驱动”一个受控的输入框！

这个问题的解决方法是把 **state 分离到两个不同的部分**：一个“高优先级”的直接更新的部分，一个“低优先级的”等待 transition 的部分。

在我们的例子中，我们已经有两个 state 变量了。输入框文本在 `query` 变量中，和我们从中读取翻译的 `resource` 变量。我们希望 `query` state 的变化可以立刻发生，但是对 `resource` 的变化（例如获取一个新的翻译）应当触发一个 transition。

所以正确的解决办法是把（不会 suspend 的）`setQuery` 放到 transition 外面，但保持（会 suspend 的）`setResource` 仍在 transition 里面。

```
function handleChange(e) {
  const value = e.target.value;

  // Outside the transition (urgent)
```

```

   setQuery(value);

    startTransition(() => {
      // Inside the transition (may be delayed)
      setResource(fetchTranslation(value));
    });
}

```

在 CodeSandbox 中尝试

通过这个更改，它可以正常工作了。我们可以直接在输入框敲字，翻译会在稍后“跟上”我们所输入的内容。

延迟一个值

默认情况下，React 总是渲染一个一致的 UI。思考下面这段代码：

```

<>
  <ProfileDetails user={user} />
  <ProfileTimeline user={user} />
</>

```

React 保证不论什么时候你去看屏幕上的这两个组件，他们所反映的数据来自同一个 `user`。如果一个不同的 `user` 因为 state 更新传递下来，你会发现他们同时变化。你无法通过录屏找到某一帧显示着不同的 `user`。（你要是真的发现了，请提交一个BUG！）

这在绝大多数情况下是合理的。不一致的 UI 会让人困惑并误导用户。（例如，如果一个通讯软件的发送按钮和对话内容界面当前选择的会话“不一致”的话是非常糟糕的。）

然而，有些时候故意引入不一致可能是有帮助的。我们可以像上面那样把 state “分离”开，但 React 也提供了一个内置的 Hook 来做这件事：

```

import { useDeferredValue } from 'react';

const deferredValue = useDeferredValue(value, {
  timeoutMs: 5000
});

```

要演示这个特性，我们要用到 [详情页切换示例](#)。点击“Next”按钮并注意它是如何使用1秒钟完成 transition 的。

假设我们获取用户详情是非常快的，只需要 300 毫秒。现在，因为我们要等待用户信息和文章列表并保持详情页显示的一致性，我们等待了整整1秒。但是如果我只是希望详情能更快的显示出来呢？

如果你希望牺牲一致性，我们可以通过给拖延我们 **transition** 的组件传递可能过时的数据来实现。那正是 `useDeferredValue()` 可以帮我们做的事情：

```

function ProfilePage({ resource }) {
  const deferredResource = useDeferredValue(resource, {
    timeoutMs: 1000
  });
  return (
    <Suspense fallback={<h1>Loading profile...</h1>}>
      <ProfileDetails resource={resource} />
      <Suspense fallback={<h1>Loading posts...</h1>}>
        <ProfileTimeline
          resource={deferredResource}

```

```

        isStale={deferredResource !== resource}
      />
    </Suspense>
  </Suspense>
);
}

function ProfileTimeline({ isStale, resource }) {
  const posts = resource.posts.read();
  return (
    <ul style={{ opacity: isStale ? 0.7 : 1 }}>
      {posts.map(post => (
        <li key={post.id}>{post.text}</li>
      ))}
    </ul>
  );
}

```

在 CodeSandbox 中尝试

我们所做的权衡是，`<ProfileTimeline>` 会和其他组件不一致并很可能显示一个过时的内容。多点几次“Next”按钮，你就会发现这个问题。但是也正因如此，我们才能把 transition 的时间从 1000ms 降到 300ms。

这到底是是不是一个合理的权衡取决于具体情况。但这是个很方便的工具，尤其是在界面切换的时候内容变化并不明显的情况下，或者用户可能根本不会注意到他看的是一秒前版本的旧数据的情况。

值得注意的是 `useDeferredValue` 并不仅仅在获取数据的时候有用。它在更新组件树的工作量过大导致交互（例如：在输入框输入内容）卡顿的情况也是有用的。正如我们可以“延迟”一个花费长时间请求的值（并且显示之前的值而不影响其他组件的更新），我们也可以把它用在组件树需要花费较长时间更新的情况下。

举个例子，请考虑像这样的一个可筛选列表：

```

function App() {
  const [text, setText] = useState("hello");

  function handleChange(e) {
    setText(e.target.value);
  }

  return (
    <div className="App">
      <label>
        Type into the input:{" "}
        <input value={text} onChange={handleChange} />
      </label>
      ...
      <MySlowList text={text} />
    </div>
  );
}

```

在 CodeSandbox 中尝试

在这个例子中，`<MySlowList>` 中的每个项都有一个人为添加的延迟 -- 每个项会延迟渲染进程几毫秒。我们永远也不会在真实的应用中这样做，但是这是帮助我们模拟在一个已经没有优化余地的深层嵌套的组件树中会发生的事情。

我们可以看到往输入框敲内容是如何导致卡顿的。现在我们把 `useDeferredValue` 加进去：

```

function App() {
  const [text, setText] = useState("hello");
  const deferredText = useDeferredValue(text, {
    timeoutMs: 5000
  });

  function handleChange(e) {
    setText(e.target.value);
  }

  return (
    <div className="App">
      <label>
        Type into the input:{" "}
        <input value={text} onChange={handleChange} />
      </label>
      ...
      <MySlowList text={deferredText} />
    </div>
  );
}

```

在 CodeSandbox 中尝试

现在输入已经很少卡顿了 -- 尽管我们是通过延迟显示结果来实现这一点的。

这和 debouncing 有什么区别？我们的例子有一个固定的人为延迟（80个项，每项延迟 3ms），所以它总是会延迟，不论我们的电脑有多快。然而，`useDeferredValue` 的值只会在渲染耗费时间的情况下“滞后”。React 并不会加入一点多余的延迟。在一个更实际的工作负荷，你可以预期这个滞后会根据用户的设备而不同。在较快的机器上，滞后会更少或者根本不存在，在较慢的机器上，它会变得更明显。不论哪种情况，应用都会保持可响应。这就是此机制优于 debouncing 或 throttling 的地方，它们总是会引入最小延迟而且不可避免的会在渲染的时候阻塞进程。

尽管在响应性上有所提升，这个例子还并不是很说服力，因为对于这个用例 Concurrent 模式缺少了一些关键优化。但是，看到像 `useDeferredValue`（或 `useTransition`）这样的特性不论是在我们等待网络还是等待计算工作完成的情况都有用还是很有趣的。

SuspenseList

`<SuspenseList>` 是有关组织加载状态的最后一个模式了。

请考虑这个例子：

```

function ProfilePage({ resource }) {
  return (
    <>
      <ProfileDetails resource={resource} />
      <Suspense fallback={<h2>Loading posts...</h2>}>
        <ProfileTimeline resource={resource} />
      </Suspense>
      <Suspense fallback={<h2>Loading fun facts...</h2>}>
        <ProfileTrivia resource={resource} />
      </Suspense>
    </>
  );
}

```

在 CodeSandbox 中尝试

在这个例子中 API 调用的时长是随机的。如果你持续的刷新，你会发现有的时候文章列表会先到达，有的时候“趣闻”会先到达。

这带来了一个问题。如果趣闻先到达了，我们会发现趣闻展示在文章列表的降级界面 `<h2>Loading posts...</h2>`。我们可能会先开始阅读这些，但是稍后文章列表的响应到达，把所有的趣闻推到下面。这感觉很不好。

其中一种解决办法是我们通过把他们放在同一个 `Suspense` 边界中：

```
<Suspense fallback={<h2>Loading posts and fun facts...</h2>}>
  <ProfileTimeline resource={resource} />
  <ProfileTrivia resource={resource} />
</Suspense>
```

在 CodeSandbox 中尝试

这个办法的问题在于现在我们总是要等待这两个数据都获取到之后。但是，如果是文章列表先到达，我们就不需要延迟显示它们。当趣闻后到达的时候，因为他们本身就在文章列表下方所以他们并不会导致布局抖动。

另一种解决办法是，比如通过一种特殊的方式组织 Promise，当我们需要从树中多个不同组件中加载 state 的时候会变得越来越难以实现。

要解决这个问题，我们要用到 `SuspenseList`：

```
import { SuspenseList } from 'react';
```

`<SuspenseList>` 协调它下面的最接近的 `<Suspense>` 节点的“展开顺序”：

```
function ProfilePage({ resource }) {
  return (
    <SuspenseList revealOrder="forwards">
      <ProfileDetails resource={resource} />
      <Suspense fallback={<h2>Loading posts...</h2>}>
        <ProfileTimeline resource={resource} />
      </Suspense>
      <Suspense fallback={<h2>Loading fun facts...</h2>}>
        <ProfileTrivia resource={resource} />
      </Suspense>
    </SuspenseList>
  );
}
```

在 CodeSandbox 中尝试

这个 `revealOrder="forwards"` 配置表示这个列表中最接近的 `<Suspense>` 只会根据在树中的显示顺序来“展开”它们的内容 -- 即使它们的数据在不同的顺序到达。`<SuspenseList>` 还有其他有趣的模式：尝试把 `"forwards"` 换成 `"backwards"` 或 `"together"` 并观察效果。

你可以利用 `tail` prop 来控制同时显示多少个加载状态。如果我们制定 `tail="collapsed"`，我们只能看到最多一个降级界面。你可以在 [这里](#) 体验一下。

请记住和 React 中的其他东西一样 `<SuspenseList>` 也是可以组合的。例如，你可以做一个 `<SuspenseList> table` 中放着 `<SuspenseList> row` 的表格。

下一步

Concurrent 模式提供了一个强大的 UI 编程模型和一系列的新的可组合的指令集来帮助你构建愉快的用户体验。

这是通过多年的调查和开发的结果，但它尚未完结。在 [采用 Concurrent 模式](#) 中，我们会讲如何使用它以及它的效果。

Concurrent 模式 API 参考(实验版)

注意：

本章节所描述的功能还处于实验阶段，在稳定版本中尚不可用。它面向的人群是早期使用者以及好奇心较强的人。

本页面中许多信息现已过时，仅仅是为了存档而存在。欲了解最新信息，请参阅 [React 18 Alpha 版公告](#)。

在 React 18 发布前，我们将用稳定的文档替代此章节。

本章节为 [Concurrent 模式](#) 的 React API 参考。如果你想找使用指南，请查阅 [Concurrent UI 模式](#)。

注意：这是社区的预览版，并不是最终的稳定版本。这些 API 将来可能会发生变化。请自行承担风险！

- [启用 Concurrent 模式](#)
 - `createRoot`
- [Suspense](#)
 - `Suspense`
 - `SuspenseList`
 - `useTransition`
 - `useDeferredValue`

启用 Concurrent 模式

`createRoot {#createroot}`

```
ReactDOM.createRoot(rootNode).render(<App />);
```

使用上述代码替换 `ReactDOM.render(<App />, rootNode)` 并启用 Concurrent 模式。

欲了解有关 Concurrent 模式的更多信息，请查阅 [Concurrent 模式文档](#)

Suspense API

`Suspense {#suspensecomponent}`

```
<Suspense fallback={<h1>加载中...</h1>}>
  <ProfilePhoto />
  <ProfileDetails />
</Suspense>
```

`Suspense` 让你的组件在渲染之前进行“等待”，并在等待时显示 `fallback` 的内容。

在这个示例中，`ProfileDetails` 正在等待异步 API 调用来获取某些数据。在等待 `ProfileDetails` 和 `ProfilePhoto` 时，我们将显示 `加载中...` 的 fallback。请注意，在 `<Suspense>` 中的所有子组件都加载之前，我们将继续显示这个 fallback。

`Suspense` 接受两个 props：

- **fallback** 接受一个加载指示器。这个 fallback 在 `Suspense` 所有子组件完成渲染之前将会一直显示。
- **unstable_avoidThisFallback** 接受一个布尔值。它告诉 React 是否在初始加载时“跳过”显示这个边界，这个 API 可能在后续版本中删除。

`<SuspenseList> {#suspenselist}`

```
<SuspenseList revealOrder="forwards">
  <Suspense fallback={['加载中...']}>
    <ProfilePicture id={1} />
  </Suspense>
  <Suspense fallback={['加载中...']}>
    <ProfilePicture id={2} />
  </Suspense>
  <Suspense fallback={['加载中...']}>
    <ProfilePicture id={3} />
  </Suspense>
  ...
</SuspenseList>
```

`SuspenseList` 通过编排向用户显示这些组件的顺序，来帮助协调许多可以挂起的组件。

当多个组件需要获取数据时，这些数据可能会以不可预知的顺序到达。不过，如果你将这些项目包装在 `SuspenseList` 中，React 将不会在列表中显示这个项目，直到它之前的项目已经显示（此行为可调整）。

`SuspenseList` 接受两个 props：

- **revealOrder (forwards, backwards, together)** 定义了 `SuspenseList` 子组件应该显示的顺序。
 - `together` 在所有的子组件都准备好了的时候显示它们，而不是一个接着一个显示。
- **tail (collapsed, hidden)** 指定如何显示 `SuspenseList` 中未加载的项目。
 - 默认情况下，`SuspenseList` 将显示列表中的所有 fallback。
 - `collapsed` 仅显示列表中下一个 fallback。
 - `hidden` 未加载的项目不显示任何信息。

请注意，`SuspenseList` 只对其下方最近的 `Suspense` 和 `SuspenseList` 组件进行操作。它不会搜索深度超过一级的边界。不过，可以将多个 `SuspenseList` 组件相互嵌套来构建栅格。

`useTransition {#usetransition}`

```
const SUSPENSE_CONFIG = { timeoutMs: 2000 };
const [startTransition, isPending] = useTransition(SUSPENSE_CONFIG);
```

`useTransition` 允许组件在切换到下一个界面之前等待内容加载，从而避免不必要的加载状态。它还允许组件将速度较慢的数据获取更新推迟到随后渲染，以便能够立即渲染更重要的更新。

`useTransition` hook 返回两个值的数组。

- `startTransition` 是一个接受回调的函数。我们用它来告诉 React 需要推迟的 state。
- `isPending` 是一个布尔值。这是 React 通知我们是否正在等待过渡的完成的方式。

如果某个 `state` 更新导致组件挂起，则该 `state` 更新应包装在 `transition` 中

```
const SUSPENSE_CONFIG = { timeoutMs: 2000 };

function App() {
  const [resource, setResource] = useState(initialResource);
  const [startTransition, isPending] = useTransition(SUSPENSE_CONFIG);
  return (
    <>
      <button
        disabled={isPending}
        onClick={() => {
          startTransition(() => {
            const nextUserId = getNextId(resource.userId);
            setResource(fetchProfileData(nextUserId));
          });
        }}
      >
        Next
      </button>
      {isPending ? "加载中..." : null}
      <Suspense fallback={<Spinner />}>
        <ProfilePage resource={resource} />
      </Suspense>
    </>
  );
}
```

在这段代码中，我们使用 `startTransition` 包装了我们的数据获取。这使我们可以立即开始获取用户资料的数据，同时推迟下一个用户资料页面以及其关联的 `Spinner` 的渲染 2 秒钟 (`timeoutMs` 中显示的时间)。

`isPending` 布尔值让 React 知道我们的组件正在切换，因此我们可以通过在之前的用户资料页面上显示一些加载文本来让用户知道这一点。

深入了解 `transition`，可以阅读 [Concurrent UI 模式](#)。

useTransition 配置

```
const SUSPENSE_CONFIG = { timeoutMs: 2000 };
```

`useTransition` 接受带有 `timeoutMs` 的可选的 **Suspense** 配置。此超时（毫秒）告诉 React 在显示下一个状态（上例中为新的用户资料页面）之前等待多长时间。

注意：我们建议你在不同的模块之间共享 **Suspense** 配置。

useDeferredValue {#usedeferredvalue}

```
const deferredValue = useDeferredValue(value, { timeoutMs: 2000 });
```

返回一个延迟响应的值，该值可能“延后”的最长时间为 `timeoutMs`。

这通常用于在具有基于用户输入立即渲染的内容，以及需要等待数据获取的内容时，保持接口的可响应性。

文本输入框是个不错的示例。

```
function App() {
  const [text, setText] = useState("hello");
  const deferredText = useDeferredValue(text, { timeoutMs: 2000 });

  return (
    <div className="App">
      {/* 保持将当前文本传递给 input */}
      <input value={text} onChange={handleChange} />
      ...
      {/* 但在必要时可以将列表“延后” */}
      <MySlowList text={deferredText} />
    </div>
  );
}
```

这让我们可以立即显示 `input` 的新文本，从而感觉到网页的响应。同时，`MySlowList` “延后” 2 秒，根据 `timeoutMs`，更新之前，允许它在后台渲染当前文本。

深入了解延迟值，可以阅读 [Concurrent UI 模式](#)。

useDeferredValue 配置

```
const SUSPENSE_CONFIG = { timeoutMs: 2000 };
```

`useDeferredValue` 所接受的配置参数 **Suspense** 可选，该参数包含 `timeoutMs` 字段。此超时（以毫秒为单位）表示延迟的值允许延后多长时间。

当网络和设备允许时，React 始终会尝试使用较短的延迟。

用于数据获取的 Suspense(试验阶段)

注意：

本章节所描述的功能还处于实验阶段，在稳定版本中尚不可用。它面向的人群是早期使用者以及好奇心较强的人。

本页面中许多信息现已过时，仅仅是为了存档而存在。欲了解最新信息，请参阅 [React 18 Alpha 版公告](#)。

在 React 18 发布前，我们将用稳定的文档替代此章节。

React 16.6 新增了 `<Suspense>` 组件，让你可以“等待”目标代码加载，并且可以直接指定一个加载的界面（像是个 spinner），让它在用户等待的时候显示：

```
const ProfilePage = React.lazy(() => import('./ProfilePage')); // 懒加载

// 在 ProfilePage 组件处于加载阶段时显示一个 spinner
<Suspense fallback={<Spinner />}>
  <ProfilePage />
</Suspense>
```

用于数据获取的 Suspense 是一个新特性，你可以使用 `<Suspense>` 以声明的方式来“等待”任何内容，包括数据。本文重点介绍它在数据获取的用例，它也可以用于等待图像、脚本或其他异步的操作。

- [何为 Suspense？](#)
 - [什么不是 Suspense](#)
 - [Suspense 可以做什么](#)
- [在实践中使用 Suspense](#)
 - [如果我不使用 Relay，怎么办？](#)
 - [致库作者](#)
- [传统实现方法 vs Suspense](#)
 - [方法 1：Fetch-on-render（渲染之后获取数据，不使用 Suspense）](#)
 - [方法 2：Fetch-then-render（接收到全部数据之后渲染，不使用 Suspense）](#)
 - [方法 3：Render-as-you-fetch（获取数据之后渲染，使用 Suspense）](#)
- [尽早开始获取数据](#)
 - [我们仍在寻求方法中](#)
- [Suspense 和 Race Conditions](#)
 - [涉及 useEffect 的 Race Conditions](#)
 - [涉及 componentDidUpdate 的 Race Conditions](#)
 - [Race Conditions 问题](#)
 - [借助 Suspense 消除 Race Condition](#)
- [错误处理](#)
- [下一步](#)

何为 Suspense?

Suspense 让组件“等待”某个异步操作，直到该异步操作结束即可渲染。在下面[例子](#)中，两个组件都会等待异步 API 的返回值：

```
const resource = fetchProfileData();

function ProfilePage() {
  return (
    <Suspense fallback={<h1>Loading profile...</h1>}>
      <ProfileDetails />
      <Suspense fallback={<h1>Loading posts...</h1>}>
        <ProfileTimeline />
      </Suspense>
    </Suspense>
  );
}

function ProfileDetails() {
  // 尝试读取用户信息，尽管该数据可能尚未加载
  const user = resource.user.read();
  return <h1>{user.name}</h1>;
}

function ProfileTimeline() {
  // 尝试读取博文信息，尽管该部分数据可能尚未加载
  const posts = resource.posts.read();
  return (
    <ul>
      {posts.map(post => (
        <li key={post.id}>{post.text}</li>
      ))}
    </ul>
  );
}
```

[在 CodeSandbox 中尝试](#)

上面的 demo 只是个示意。别担心看不懂代码。我们后面会详细说明这部分代码的运作方式。需要记住的是，Suspense 其实更像是一种机制，而 demo 中那些具体的 API，如 `fetchProfileData()` 或者 `resource.posts.read()`，这些 API 本身并不重要。不过，如果你还是对它们很好奇，可以在这个 [demo sandbox](#) 中找到它们的定义。

Suspense 不是一个数据请求的库，而是一个机制。这个机制是用来给数据请求库向 React 通信说明某个组件正在读取的数据当前仍不可用。通信之后，React 可以继续等待数据的返回并更新 UI。在 Facebook，我们用了 Relay 和它的[集成 Suspense 新功能](#)。我们期望其他的库，如 Apollo，也能支持类似的集成。

从长远来看，我们想让 Suspense 成为组件读取异步数据的主要方式——无论数据来自何方。

什么不是 Suspense

Suspense 和当下其他解决异步问题的方法存在明显差异，因而，第一次接触 Suspense 容易让人产生误解。下面我们阐述下常见的误解：

- **它不是数据获取的一种实现。** 它并不假定你使用 GraphQL、REST，或者任何其他特定的数据格式、库、数据传输方式、协议。

- 它不是一个可以直接用于数据获取的客户端。你不能用 Suspense 来“替代”`fetch` 或者 Relay。不过你可以使用集成 Suspense 的库（比如说，[新的 Relay API](#)）。
- 它不使数据获取与视图层代码耦合。它协助编排加载状态在 UI 中的显示，但它并不将你的网络逻辑捆绑到 React 组件。

Suspense 可以做什么

说了那么多，Suspense 到底有什么用呢？对于这个问题，我们可以从不同的角度来回答：

- 它能让数据获取库与 React 紧密整合。如果一个数据请求库实现了对 Suspense 的支持，那么，在 React 中使用 Suspense 将会是自然不过的事。
- 它能让你有针对性地安排加载状态的展示。虽然它不干涉数据怎样获取，但它可以让你对应用的视图加载顺序有更大的控制权。
- 它能够消除 **race conditions**。即便是用上 `await`，异步代码还是很容易出错。相比之下，Suspense 更给人同步读取数据的感觉——假定数据已经加载完毕。

在实践中使用 Suspense

在 Facebook 中，我们目前只在生产环境使用集成了 Suspense 的 Relay。[如果你正在找一份实用指南来上手 Suspense，可以看这份 Relay 指南！](#)指南中写明了当前运行在我们在生产环境中的可用模式。

本文所有演示代码均使用“伪”API 实现，而不是 Relay。我们这样做的目的是想让代码本身更易懂些，让不熟悉 GraphQL 的读者也能看懂代码。也正因为示例代码使用“伪 API”，示例代码本身并不是在应用中使用 Suspense 的“正确方式”。可以说，本文是从概念上出发，目的是帮你了解为什么 Suspense 是以特定方式运行，以及 Suspense 解决了哪些问题这两件事情。

如果我不使用 Relay，怎么办？

如果你当下并不使用 Relay，那么你暂时无法在应用中试用 Suspense。因为迄今为止，在实现了 Suspense 的库中，Relay 是我们唯一在生产环境测试过，且对它的运作有把握的一个库。

在接下来的几个月里，许多库将会实现它们各自支持 Suspense 的 API。[如果你倾向于等到技术更加稳定之后才开始学习，那大概率你会先不看这部分文档，等到 Suspense 的生态更成熟之后再回来学习。](#)

如果你有兴趣的话，也可以自己开发，然后将你对 Suspense 的实现集成到某个数据请求库中。

致库作者

我们很期待看到社区中其他库对 Suspense 进行试验。对于数据请求库的作者，有一件重要的事情需要你们引起注意。

尽管实现对 Suspense 的支持从技术上是可行的，Suspense 当前并不作为在组件渲染的时候开始获取数据的方式。反而，它让组件表达出它们在正在“等待”已经发出获取行为的数据。[使用 Concurrent 模式和 Suspense 来构建优秀的用户体验一文说明了这一点的重要性，以及如何在实践中实现这个模式。](#)

除非你有现成的解决方法来避免瀑布（waterfall）问题，我们建议采用支持在渲染之前就能先获取数据的 API。关于实现这类 API 的具体例子，你可以查看 [Relay Suspense API](#) 实现预加载的方式。对于这方面的信息，我们当前给出的和过去给出的并不完全一致。因为 Suspense 用于数据获取还处于试验阶段，我们的建议会随着我们对 Suspense 在生产环境中的使用习得和对瀑布问题的理解，而发生变化。

传统实现方式 vs Suspense

我们可以完全不提及当前主流的数据获取方式，只介绍 Suspense。但这样做的话，以下 3 件事情：Suspense 解决了什么问题、为什么这些问题值得处理、以及 Suspense 和其他现存方法的不同，会更难理解。

因此，我们把 Suspense 看作是一系列解决方法的下一步逻辑演化。从这个角度对其展开介绍：

- **Fetch-on-render (渲染之后获取数据，如：在 `useEffect` 中 `fetch`)**：先开始渲染组件，每个完成渲染的组件都可能在它们的 effects 或者生命周期函数中获取数据。这种方式经常导致“瀑布”问题。
- **Fetch-then-render (接收到全部数据之后渲染，如：不使用 Suspense 的 Relay)**：先尽早获取下一屏需要的所有数据，数据准备好后，渲染新的屏幕。但在数据拿到之前，我们什么事也做不了。
- **Render-as-you-fetch (获取数据之后渲染，如：使用了 Suspense 的 Relay)**：先尽早获取下一屏需要的所有数据，然后立刻渲染新的屏幕——在网络响应可用之前就开始。在接收到数据的过程中，React 迭代地渲染需要数据的组件，直到渲染完所有内容为止。

注意

这部分流程经过了简化处理，在实际应用中，真正被采用的方法通常是由不同方法混合而成。但是，我们接下来还是会逐一审视这些方法，因为这样可以让我们更好理解它们各自的优劣。

为了对比这 3 个方法，我们分别用它们实现一个 profile 页面。

方法 1: Fetch-on-render (渲染之后获取数据, 不使用 Suspense)

目前 React 应用中常用的数据获取方式是使用 effect：

```
// 在函数组件中：
useEffect(() => {
  fetchSomething();
}, []);

// 或者，在 class 组件里：
componentDidMount() {
  fetchSomething();
}
```

我们称这种方法为“fetch-on-render”（渲染之后获取数据），因为数据的获取是发生在组件被渲染到屏幕之后。这种方法会导致“瀑布”的问题。

仔细看下面的 `<ProfilePage>` 和 `<ProfileTimeline>` 组件：

```
function ProfilePage() {
  const [user, setUser] = useState(null);

  useEffect(() => {
    fetchUser().then(u => setUser(u));
  }, []);

  if (user === null) {
    return <p>Loading profile...</p>;
  }
}
```

```

}
return (
  <>
  <h1>{user.name}</h1>
  <ProfileTimeline />
</>
);
}

function ProfileTimeline() {
  const [posts, setPosts] = useState(null);

  useEffect(() => {
    fetchPosts().then(p => setPosts(p));
  }, []);

  if (posts === null) {
    return <h2>Loading posts...</h2>;
  }
  return (
    <ul>
      {posts.map(post => (
        <li key={post.id}>{post.text}</li>
      ))}
    </ul>
  );
}

```

在 [CodeSandbox](#) 中尝试

如果你运行上面的代码，你会发现 console 打印如下信息：

1. We start fetching user details (我们开始获取用户信息)
2. We wait... (我们处于等待中)
3. We finish fetching user details (我们接收完所有的用户信息)
4. We start fetching posts (我们开始获取博文数据)
5. We wait... (我们处于等待中)
6. We finish fetching posts (我们接收完所有的博文数据)

假设获取用户信息需要 3 秒，那么在这个方法中，我们只能在 3 秒之后，才开始获取博文数据。这就是上面提到的“瀑布”问题：本该并行发出的请求无意中被串行发送出去。

在渲染之后再获取数据是引发“瀑布”问题的常见原因。虽然这种情况下的“瀑布”问题可以被解决，但随着项目代码的增多，开发者更倾向于选用其他不会引发这个问题的数据获取方法。

方法 2:Fetch-then-render(接收到全部数据之后渲染, 不使用 Suspense)

通过提供更集中化的方式来实现数据获取，库可以避免“瀑布”问题。比如说，Relay 是通过把组件所需的数据转移到可静态分析的 *fragments* 上，*fragments* 随后会被整合进一个单一的请求。

在本文中，我们不假定读者了解 Relay，因而我们不会在方法 2 的示例代码中使用它。我们做的，是手动把获取数据的方法合并到一起，来模拟 Relay 的行为：

```

function fetchProfileData() {
  return Promise.all([
    fetchUser(),

```

```

    fetchPosts()
  ]).then(([user, posts]) => {
    return {user, posts};
  })
}

```

在下面例子中，`<ProfilePage>` 等待两个并行发出的请求：

```

// 尽早开始获取数据
const promise = fetchProfileData();

function ProfilePage() {
  const [user, setUser] = useState(null);
  const [posts, setPosts] = useState(null);

  useEffect(() => {
    promise.then(data => {
      setUser(data.user);
      setPosts(data.posts);
    });
  }, []);

  if (user === null) {
    return <p>Loading profile...</p>;
  }
  return (
    <>
      <h1>{user.name}</h1>
      <ProfileTimeline posts={posts} />
    </>
  );
}

// 子组件不再触发数据请求
function ProfileTimeline({ posts }) {
  if (posts === null) {
    return <h2>Loading posts...</h2>;
  }
  return (
    <ul>
      {posts.map(post => (
        <li key={post.id}>{post.text}</li>
      ))}
    </ul>
  );
}

```

[在 CodeSandbox 中尝试](#)

在方法 2 中，`console` 打印的信息变成这样：

1. We start fetching user details (我们开始获取用户信息)
2. We start fetching posts (我们开始获取博文数据)
3. We wait... (我们处于等待中)
4. We finish fetching user details (我们接收完所有的用户信息)
5. We finish fetching posts (我们接收完所有的博文数据)

这里，我们解决了之前出现的网络“瀑布”问题，却意外引出另外一个问题。我们在 `fetchProfileData` 方法里利用 `Promise.all()` 来等待所有数据，这就导致了，即便我们先接收完用户信息的数据，我们也不能先渲染 `ProfileDetails` 这个组件，还得等到博文信息也接收完才行。在这个方法中，我们必须等到两份数据都接收完毕。

当然，这个例子的问题是可以解决的。我们可以去掉 `Promise.all()`，改用分别等待两个 Promise 的方式来解决。但随着数据和组件树复杂度的增加，这个方法的缺点会逐渐显现出来。如果数据树中出现部分数据的缺失或者过时，则很难写出健壮可靠的组件。因此，在拿到新屏幕所需的全部数据之后，再去渲染页面通常是一个比较现实的选择。

方法 3: Render-as-you-fetch(获取数据之后渲染, 使用 Suspense)

在上面方法 2 中，我们是在调用 `setState` 之前就开始获取数据：

1. 开始获取数据
2. 结束获取数据
3. 开始渲染

有了 `Suspense`，我们依然可以先获取数据，而且可以给上面流程的 2、3 步骤调换顺序：

1. 开始获取数据
2. 开始渲染
3. 结束获取数据

有了 `Suspense`，我们不必等到数据全部返回才开始渲染。实际上，我们是一发送网络请求，就马上开始渲染：

```
// 这不是一个 Promise。这是一个支持 Suspense 的特殊对象。
const resource = fetchProfileData();

function ProfilePage() {
  return (
    <Suspense fallback={<h1>Loading profile...</h1>}>
      <ProfileDetails />
      <Suspense fallback={<h1>Loading posts...</h1>}>
        <ProfileTimeline />
      </Suspense>
    </Suspense>
  );
}

function ProfileDetails() {
  // 尝试读取用户信息，尽管信息可能未加载完毕
  const user = resource.user.read();
  return <h1>{user.name}</h1>;
}

function ProfileTimeline() {
  // 尝试读取博文数据，尽管数据可能未加载完毕
  const posts = resource.posts.read();
  return (
    <ul>
      {posts.map(post => (
        <li key={post.id}>{post.text}</li>
      ))}
    </ul>
  );
}
```

在 CodeSandbox 中尝试

以下是方法 3 中当我们渲染 `<ProfilePage>` 时会发生的事情：

1. 我们一开始就通过 `fetchProfileData()` 发出请求。这个方法返回给我们一个特殊的对象“resource”，而不是一个 `Promise`。在现实的案例中，这个对象是由 Relay 通过集成了 `Suspense` 来提供的。
2. React 尝试渲染 `<ProfilePage>`。该组件返回两个子组件：`<ProfileDetails>` 和 `<ProfileTimeline>`。
3. React 尝试渲染 `<ProfileDetails>`。该组件调用了 `resource.user.read()`，但因为读取的数据还没被获取完毕，所以组件会处于一个“挂起”的状态。React 会跳过这个组件，继续渲染组件树中的其他组件。
4. React 尝试渲染 `<ProfileTimeline>`。该组件调用了 `resource.posts.read()`，和上面一样，数据还没获取完毕，所以这个组件也是处在“挂起”的状态。React 同样跳过这个组件，去渲染组件树中的其他组件。
5. 组件树中已经没有其他组件需要渲染了。因为 `<ProfileDetails>` 组件处于“挂起”状态，React 则是显示出距其上游最近的 `<Suspense>` `fallback`：`<h1>Loading profile...</h1>`。渲染到这里就结束了。

这里的 `resource` 对象表示尚未存在但最终将被加载的数据。当我们调用 `read()` 的时候，我们要么获取数据，要么组件处于“挂起”状态。

随着更多数据的到来，React 将尝试重新渲染，并且每次都可能渲染出更加完整的组件树。当 `resource.user` 的数据获取完毕之后，`<ProfileDetails>` 组件就能被顺利渲染出来，这时，我们就不再需要展示 `<h1>Loading profile...</h1>` 这个 `fallback` 了。当我们拿到全部数据之后，所有的 `fallbacks` 就都可以不展示了。

这意味着一个有趣的事，即使我们使用 GraphQL 客户端来收集单个请求中需要的所有数据，流式响应也可以使我们尽早显示更多的内容。在数据获取时 (`render-as-we-fetch`) (而不是全部数据获取后) 渲染，因此，如果 `user` 在响应中比 `posts` 出现得更早，我们则可以在响应结束之前“解锁”外层的 `<Suspense>` 边界。我们之前并没有意识到这一点，即便是 `fetch-then-render` (接收到全部数据之后渲染) 这个解决方案，在数据获取和渲染之间也有“瀑布”问题。`Suspense` 没有这个“瀑布”问题，像 Relay 这样的库就利用了这个优势。

请注意，我们是如何在组件中去掉 `if (...) "is loading"` 这个检查的。这不仅删除了样板代码，还简化了代码设计快速转变的流程。举个例子，如果我们想同时“弹出”`<ProfileDetails>` 组件和 `<ProfileTimeline>` 组件，只需删除两者之间的 `<suspense>`。或者，我们可以通过给它们各自的 `<Suspense>` 来让两者彼此独立。通过 `Suspense`，我们可以更改加载状态的粒度并控制顺序，而无需调整代码。

尽早开始获取数据

如果你正在开发数据请求库，对于“获取数据之后渲染”这个方法，你会想知道一件至关重要的事情：**我们是在渲染之前就进行数据获取**。可以仔细观察下面代码：

```
// 一早就开始数据获取，在渲染之前！
const resource = fetchProfileData();

// ...

function ProfileDetails() {
  // 尝试读取用户信息
  const user = resource.user.read();
  return <h1>{user.name}</h1>;
}
```

在 CodeSandbox 中尝试

请注意，此示例中的 `read()` 调用尚未开始。它只是试图读取已经获取的数据。在创建具有 Suspense 的敏捷应用程序时，这种差异非常重要。我们并不想把数据获取推迟到组件渲染之后。因此，作为数据请求库的作者，你可以将 `resource` 对象设计成在数据开始请求之前无法被获取，来实现数据请求先发生于组件渲染。本文中所有使用“伪 API”的演示都实现了对请求和渲染的顺序控制。

你可能认为演示代码中“在最顶层”获取数据的操作不切实际。如果我们想跳转到另一个人的 profile 页面怎么办？我们可能希望根据 `props` 获取数据。则解决问题的方法是：**在事件处理函数中开始获取数据**。下面是在不同 profile 页面间导航的简单示例：

```
// 开始获取数据，越快越好
const initialResource = fetchProfileData(0);

function App() {
  const [resource, setResource] = useState(initialResource);
  return (
    <>
      <button onClick={() => {
        const nextUserId = getNextId(resource.userId);
        // 再次获取数据：用户点击时
        setResource(fetchProfileData(nextUserId));
      }}>
        Next
      </button>
      <ProfilePage resource={resource} />
    </>
  );
}
```

[在 CodeSandbox 中尝试](#)

通过这种方法，我们可以**并行获取代码和数据**。在页面之间导航时，我们不必等待页面上的代码加载就可以开始加载页面数据。我们可以同时开始获取代码和数据（单击链接时），从而提供更好的用户体验。

接下来需解决的问题是：在渲染下个页面之前，我们怎么知道要获取什么数据”。对此解决方法有多种（例如，将数据请求集成到路由解决方案附近）。如果你正在开发数据请求库，那么[使用 Concurrent 模式和 Suspense 来构建优秀的用户体验](#)将深入探讨了如何解决此问题及其重要性。

我们仍在寻求方法中

Suspense 本身作为一个机制而言，它灵活可变并且没有太多的限制。而产品的代码需要足够多的限制来保障代码中不会有“瀑布”问题。关于如何提供保障这一点，目前是有不同的实现方式。当下，我们仍在探索以下问题：

- 提前请求数据可能很困难。我们可以轻松避免瀑布问题吗？
- 当我们获取页面数据时，API是否应该包含数据以便从该页面立即转换？
- 响应的有效期是多长？缓存应该是全局的还是本地的？谁管理缓存？
- 可以不通过插入 `read()`，让代理协助表示延迟加载的 APIs 吗？
- 对于任意给定的 Suspense 数据，GraphQL 查询的替代物是什么？

对于这些问题，Relay 有自己的答案。当然，解决这些问题的方法不止一种，我们很期待看到即将出现在 React 社区的新想法

Suspense 和 Race Conditions

Race Conditions 是由于对代码运行顺序的错误假设而导致的 bug。使用生命周期方法（如：`useEffect` Hook 和类的 `componentDidUpdate` 方法）获取数据经常会导致这种情况。Suspense 在这里很有用，让我们看看如何实现。

为了说明问题，我们增加一个顶层组件 `<App>` 来渲染 `<ProfilePage>`，并放置一个可以在不同的 **profile** 页面之间切换的按钮：

```
function getNextId(id) {
  // ...
}

function App() {
  const [id, setId] = useState(0);
  return (
    <>
      <button onClick={() => setId(getNextId(id))}>
        Next
      </button>
      <ProfilePage id={id} />
    </>
  );
}
```

让我们比较一下不同的数据获取方法如何实现这个需求。

涉及 `useEffect` 的 Race Conditions

首先，我们将尝试使用原始的“在 effect 中获取数据”示例，重写在 `<ProfilePage>` props 中传递的 `id` 参数，以传给 `fetchUser(id)` 和 `fetchPosts(id)`：

```
function ProfilePage({ id }) {
  const [user, setUser] = useState(null);

  useEffect(() => {
    fetchUser(id).then(u => setUser(u));
  }, [id]);

  if (user === null) {
    return <p>Loading profile...</p>;
  }
  return (
    <>
      <h1>{user.name}</h1>
      <ProfileTimeline id={id} />
    </>
  );
}

function ProfileTimeline({ id }) {
  const [posts, setPosts] = useState(null);

  useEffect(() => {
    fetchPosts(id).then(p => setPosts(p));
  }, [id]);

  if (posts === null) {
```

```

        return <h2>Loading posts...</h2>;
    }
    return (
      <ul>
        {posts.map(post => (
          <li key={post.id}>{post.text}</li>
        )));
      </ul>
    );
}

```

[在 CodeSandbox 中尝试](#)

请注意，我们将 effect 的依赖从 `[]` 更改为 `[id]` ——因为我们希望每次 `id` 改变时都重新运行 effect。如果不这样做，我们将无法再次获取到新的数据。

如果我们尝试运行此代码，一开始看起来运行得很好。但是，如果我们将“伪 API”实现的延迟时间随机化，快速点击“Next”按钮，就会发现控制台日志中有些问题。**有时，在把 profile 页面切换成别的 ID 后，旧的 profile 的请求会“返回”——这种情况下，其他 ID 用的旧响应会覆盖新的 state。**

这个问题是可以解决的（通过在 effect 里头配置 `cleanup` 函数来过滤、或者取消过期请求），但它依然是个反直觉的问题，且难以检测。

涉及 componentDidUpdate 的 Race Conditions

有的人可能认为 race conditions 这个问题只跟 `useEffect` 有关系，或者只和 Hooks 有关。如果我们把上面代码改用 class 组件来实现，或者用 `async / await` 来写，可能就能避开这个问题？

先一起试试看：

```

class ProfilePage extends React.Component {
  state = {
    user: null,
  };
  componentDidMount() {
    this.fetchData(this.props.id);
  }
  componentDidUpdate(prevProps) {
    if (prevProps.id !== this.props.id) {
      this.fetchData(this.props.id);
    }
  }
  async fetchData(id) {
    const user = await fetchUser(id);
    this.setState({ user });
  }
  render() {
    const { id } = this.props;
    const { user } = this.state;
    if (user === null) {
      return <p>Loading profile...</p>;
    }
    return (
      <>
        <h1>{user.name}</h1>
        <ProfileTimeline id={id} />
      </>
    );
  }
}

```

```

    );
}

class ProfileTimeline extends React.Component {
  state = {
    posts: null,
  };
  componentDidMount() {
    this.fetchData(this.props.id);
  }
  componentDidUpdate(prevProps) {
    if (prevProps.id !== this.props.id) {
      this.fetchData(this.props.id);
    }
  }
  async fetchData(id) {
    const posts = await fetchPosts(id);
    this.setState({ posts });
  }
  render() {
    const { posts } = this.state;
    if (posts === null) {
      return <h2>Loading posts...</h2>;
    }
    return (
      <ul>
        {posts.map(post => (
          <li key={post.id}>{post.text}</li>
        ))}
      </ul>
    );
  }
}

```

[在 CodeSandbox 中尝试](#)

上面代码看似简单易读，实则暗含同样的问题。

所以很不幸，无论是改用 class 组件，还是改用 `async / await` 都没能解决 race conditions。上面代码和用 Hooks 一样都有问题，问题的源头也一样。

Race Conditions 问题

React 组件有它们自己的“生命周期”。组件可能在任意时间点接收到 props 或者更新 state。然而，每一个异步请求同样也有自己的“生命周期”。异步请求的生命周期开始于我们发出请求，结束于我们收到响应报文。这里我们面临的问题是，如何在这两类生命周期之间及时进行“同步”。这个问题很是棘手。

借助 Suspense 消除 Race Condition

我们再来重写上面代码，但这次我们只用 Suspense 来写：

```

const initialResource = fetchProfileData(0);

function App() {
  const [resource, setResource] = useState(initialResource);
  return (
    <>

```

```

    <button onClick={() => {
      const nextUserId = getNextId(resource.userId);
      setResource(fetchProfileData(nextUserId));
    }}>
      Next
    </button>
    <ProfilePage resource={resource} />
  </>
);
}

function ProfilePage({ resource }) {
  return (
    <Suspense fallback={<h1>Loading profile...</h1>}>
      <ProfileDetails resource={resource} />
      <Suspense fallback={<h1>Loading posts...</h1>}>
        <ProfileTimeline resource={resource} />
      </Suspense>
    </Suspense>
  );
}

function ProfileDetails({ resource }) {
  const user = resource.user.read();
  return <h1>{user.name}</h1>;
}

function ProfileTimeline({ resource }) {
  const posts = resource.posts.read();
  return (
    <ul>
      {posts.map(post => (
        <li key={post.id}>{post.text}</li>
      ))}
    </ul>
  );
}

```

在 CodeSandbox 中尝试

在这个用上 Suspense 的示例中，我们只需要获取一个数据 `resource`，所以我们把它提到最外层，作为顶层变量。考虑到我们有多个 resources，我们把这个变量放入 `<App>` 组件的 state。

```

const initialResource = fetchProfileData(0);

function App() {
  const [resource, setResource] = useState(initialResource);

```

当我们点击“Next”按钮，`<App>` 组件便开始发出获取下一个 profile 的请求，并把请求返回的对象下传给 `<ProfilePage>` 组件。

```

<>
  <button onClick={() => {
    const nextUserId = getNextId(resource.userId);
    setResource(fetchProfileData(nextUserId));
  }}>
    Next
  </button>

```

```

</button>
<ProfilePage resource={resource} />
</>

```

再次申明，要注意这里我们并不是等到响应报文被接收之后，才去更新 state，而是反过来：我们发出请求之后，马上就开始更新 state（外加开始渲染）。一旦收到响应，React 就把可用的数据用到 `<Suspense>` 组件里头。

和之前的两版一样，这版代码的可读性也很强，但它和之前的有所不同，Suspense 的这版不会有 race conditions 问题。你可能好奇为什么不会有，这是因为，相比之前的两版，在 Suspense 的这版里，我们不需要太怎么考虑时机这个事情。在第一版有 race conditions 问题的代码中，state 需要在准确的时机设置，否则就会出错。然而在 Suspense 版本里，我们是都是在获取数据之后，立马就设置 state——因此大大降低出错的概率。

错误处理

每当使用 Promises，大概率我们会用 `catch()` 来做错误处理。但当我们用 Suspense 时，我们不等待 Promises 就直接开始渲染，这时 `catch()` 就不适用了。这种情况下，错误处理该怎么进行呢？

在 Suspense 中，获取数据时抛出的错误和组件渲染时的报错处理方式一样——你可以在需要的层级渲染一个[错误边界](#)组件来“捕捉”层级下面的所有报错信息。

首先，给我们的项目定义一个错误边界组件：

```

// 目前，错误边界组件只支持通过 class 组件定义。
class ErrorBoundary extends React.Component {
  state = { hasError: false, error: null };
  static getDerivedStateFromError(error) {
    return {
      hasError: true,
      error
    };
  }
  render() {
    if (this.state.hasError) {
      return this.props.fallback;
    }
    return this.props.children;
  }
}

```

其次，我们将它放到组件树中任意我们想要的地方来捕捉错误：

```

function ProfilePage() {
  return (
    <Suspense fallback={<h1>Loading profile...</h1>}>
      <ProfileDetails />
      <ErrorBoundary fallback={<h2>Could not fetch posts.</h2>}>
        <Suspense fallback={<h1>Loading posts...</h1>}>
          <ProfileTimeline />
        </Suspense>
      </ErrorBoundary>
    </Suspense>
  );
}

```

[在 CodeSandbox 中尝试](#)

上面代码中的错误边界组件既能捕捉渲染过程的报错，也能捕捉 Suspense 里头数据获取的报错。理论上，我们在组件树中插入多少个错误边界组件都是可以的，但这并不是推荐的做法，错误边界组件的位置最好是深思熟虑之后再确定。

下一步

到这里，我们已经介绍完当 Suspense 用于数据获取时的基本内容。更重要的是，我们现在对 Suspense 有自己运作方式的原因，以及它是如何在数据获取这个领域中发挥自己的作用这两点有更好的理解。

Suspense 本身解答了一些问题，但同时它也引出一些新的问题：

- 如果部分组件处于“挂起”状态，整个应用会卡死吗？该怎么避免这个问题？
- 如果我们不想在目标组件的上层，而想在其他地方展示 spinner，可以实现吗？
- 如果我们想有计划地在一个短的时间内展示不同的 UI，能够实现吗？
- 除了展示个 spinner，我们能添加额外的视觉效果吗？像是给现有界面加上蒙层之类的？
- 在[最后一个 Suspense 的代码示例中](#)，为什么在点击了“Next”按钮之后，会报出警告？

对于上述问题的解答，我们将交由下一章节 [Concurrent UI 模式](#) 处理。

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` 或者 [条件运算符](#) 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。

观察这两个组件：

```
function UserGreeting(props) {
  return <h1>Welcome back!</h1>;
}

function GuestGreeting(props) {
  return <h1>Please sign up.</h1>;
}
```

再创建一个 `Greeting` 组件，它会根据用户是否登录来决定显示上面的哪一个组件。

```
function Greeting(props) {
  const isLoggedIn = props.isLoggedIn;
  if (isLoggedIn) {
    return <UserGreeting />;
  }
  return <GuestGreeting />;
}

const root = ReactDOM.createRoot(document.getElementById('root'));
// Try changing to isLoggedIn={true}:
root.render(<Greeting isLoggedIn={false} />);
```

[在 CodePen 上尝试](#)

这个示例根据 `isLoggedIn` 的值来渲染不同的问候语。

元素变量

你可以使用变量来储存元素。它可以帮助你有条件地渲染组件的一部分，而其他的渲染部分并不会因此而改变。

观察这两个组件，它们分别代表了注销和登录按钮：

```
function LoginButton(props) {
  return (
    <button onClick={props.onClick}>
      Login
    </button>
  );
}

function LogoutButton(props) {
  return (
    <button onClick={props.onClick}>
      Logout
    </button>
  );
}
```

```

        </button>
    );
}

```

在下面的示例中，我们将创建一个名叫 `LoginControl` 的有状态的组件。

它将根据当前的状态来渲染 `<LoginButton />` 或者 `<LogoutButton />`。同时它还会渲染上一个示例中的 `<Greeting />`。

```

class LoginControl extends React.Component {
  constructor(props) {
    super(props);
    this.handleLoginClick = this.handleLoginClick.bind(this);
    this.handleLogoutClick = this.handleLogoutClick.bind(this);
    this.state = {isLoggedIn: false};
  }

  handleLoginClick() {
    this.setState({isLoggedIn: true});
  }

  handleLogoutClick() {
    this.setState({isLoggedIn: false});
  }

  render() {
    const isLoggedIn = this.state.isLoggedIn;
    let button;

    if (isLoggedIn) {
      button = <LogoutButton onClick={this.handleLogoutClick} />;
    } else {
      button = <LoginButton onClick={this.handleLoginClick} />;
    }

    return (
      <div>
        <Greeting isLoggedIn={isLoggedIn} />
        {button}
      </div>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<LoginControl />);

```

[在 CodePen 上尝试](#)

声明一个变量并使用 `if` 语句进行条件渲染是不错的方式，但有时你可能会想使用更为简洁的语法。接下来，我们将介绍几种在 JSX 中内联条件渲染的方法。

与运算符 `&&`

通过花括号包裹代码，你可以在 JSX 中嵌入表达式。这也包括 JavaScript 中的逻辑与 (`&&`) 运算符。它可以很方便地进行元素的条件渲染：

```

function Mailbox(props) {
  const unreadMessages = props.unreadMessages;
  return (
    <div>
      <h1>Hello!</h1>
      {unreadMessages.length > 0 &&
        <h2>
          You have {unreadMessages.length} unread messages.
        </h2>
      }
    </div>
  );
}

const messages = ['React', 'Re: React', 'Re:Re: React'];

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Mailbox unreadMessages={messages} />);

```

[在 CodePen 上尝试](#)

之所以能这样做，是因为在 JavaScript 中，`true && expression` 总是会返回 `expression`，而 `false && expression` 总是会返回 `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。

请注意，`falsy 表达式` 会使 `&&` 后面的元素被跳过，但会返回 falsy 表达式的值。在下面示例中，`render` 方法的返回值是 `<div>0</div>`。

```

render() {
  const count = 0;
  return (
    <div>
      {count && <h1>Messages: {count}</h1>}
    </div>
  );
}

```

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`。

在下面这个示例中，我们用它来条件渲染一小段文本

```

render() {
  const isLoggedIn = this.state.isLoggedIn;
  return (
    <div>
      The user is <b>{isLoggedIn ? 'currently' : 'not'}</b> logged in.
    </div>
  );
}

```

同样的，它也可以用于较为复杂的表达式中，虽然看起来不是很直观：

```

render() {
  const isLoggedIn = this.state.isLoggedIn;
  return (
    <div>
      {isLoggedIn
        ? <LogoutButton onClick={this.handleLogoutClick} />
        : <LoginButton onClick={this.handleLoginClick} />
      }
    </div>
  );
}

```

就像在 JavaScript 中一样，你可以根据团队的习惯来选择可读性更高的代码风格。需要注意的是，如果条件变得过于复杂，那你应该考虑如何[提取组件](#)。

阻止组件渲染

在极少数情况下，你可能希望能隐藏组件，即使它已经被其他组件渲染。若要完成此操作，你可以让 `render` 方法直接返回 `null`，而不进行任何渲染。

下面的示例中，`<WarningBanner />` 会根据 prop 中 `warn` 的值来进行条件渲染。如果 `warn` 的值是 `false`，那么组件则不会渲染：

```

function WarningBanner(props) {
  if (!props.warn) {
    return null;
  }

  return (
    <div className="warning">
      Warning!
    </div>
  );
}

class Page extends React.Component {
  constructor(props) {
    super(props);
    this.state = {showWarning: true};
    this.handleToggleClick = this.handleToggleClick.bind(this);
  }

  handleToggleClick() {
    this.setState(state => ({
      showWarning: !state.showWarning
    }));
  }

  render() {
    return (
      <div>
        <WarningBanner warn={this.state.showWarning} />
        <button onClick={this.handleToggleClick}>
          {this.state.showWarning ? 'Hide' : 'Show'}
        </button>
      </div>
    );
  }
}

```

```
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Page />);
```

[在 CodePen 上尝试](#)

在组件的 `render` 方法中返回 `null` 并不会影响组件的生命周期。例如，上面这个示例中，`componentDidUpdate` 依然会被调用。

[Go to TOC](#)

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。

在一个典型的 React 应用中，数据是通过 props 属性自上而下（由父及子）进行传递的，但此种用法对于某些类型的属性而言是极其繁琐的（例如：地区偏好，UI 主题），这些属性是应用程序中许多组件都需要的。Context 提供了一种在组件之间共享此类值的方式，而不必显式地通过组件树的逐层传递 props。

- [何时使用 Context](#)
- [使用 Context 之前的考虑](#)
- [API](#)
 - [React.createContext](#)
 - [Context.Provider](#)
 - [Class.contextType](#)
 - [Context.Consumer](#)
 - [Context.displayName](#)
- [示例](#)
 - [动态 Context](#)
 - [在嵌套组件中更新 Context](#)
 - [使用多个 Context](#)
- [注意事项](#)
- [废弃的 API](#)

何时使用 Context

Context 设计目的是为了共享那些对于一个组件树而言是“全局”的数据，例如当前认证的用户、主题或首选语言。举个例子，在下面的代码中，我们通过一个“theme”属性手动调整一个按钮组件的样式：

```
embed:context/motivation-problem.js
```

使用 context，我们可以避免通过中间元素传递 props：

```
embed:context/motivation-solution.js
```

使用 Context 之前的考虑

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）有时候是一个比 context 更好的解决方案。

比如，考虑这样一个 `Page` 组件，它层层向下传递 `user` 和 `avatarSize` 属性，从而让深度嵌套的 `Link` 和 `Avatar` 组件可以读取到这些属性：

```
<Page user={user} avatarSize={avatarSize} />
// ... 渲染出 ...
<PageLayout user={user} avatarSize={avatarSize} />
// ... 渲染出 ...
<NavigationBar user={user} avatarSize={avatarSize} />
// ... 渲染出 ...
<Link href={user.permalink}>
  <Avatar user={user} size={avatarSize} />
</Link>
```

如果在最后只有 `Avatar` 组件真的需要 `user` 和 `avatarSize`, 那么层层传递这两个 props 就显得非常冗余。而且一旦 `Avatar` 组件需要更多来自顶层组件的 props, 你还得在中间层级一个一个加上去, 这将会变得非常麻烦。

一种 **无需 context** 的解决方案是将 `Avatar` 组件自身传递下去, 因为中间组件无需知道 `user` 或者 `avatarSize` 等 props :

```
function Page(props) {
  const user = props.user;
  const userLink = (
    <Link href={user.permalink}>
      <Avatar user={user} size={props.avatarSize} />
    </Link>
  );
  return <PageLayout userLink={userLink} />;
}

// 现在, 我们有这样的组件:
<Page user={user} avatarSize={avatarSize} />
// ... 渲染出 ...
<PageLayout userLink={...} />
// ... 渲染出 ...
<NavigationBar userLink={...} />
// ... 渲染出 ...
{props.userLink}
```

这种变化下, 只有最顶部的 `Page` 组件需要知道 `Link` 和 `Avatar` 组件是如何使用 `user` 和 `avatarSize` 的。

这种对组件的控制反转减少了在你的应用中要传递的 props 数量, 这在很多场景下会使得你的代码更加干净, 使你对根组件有更多的把控。但是, 这并不适用于每一个场景 : 这种将逻辑提升到组件树的更高层次来处理, 会使得这些高层组件变得更复杂, 并且会强行将低层组件适应这样的形式, 这可能不会是你想要的。

而且你的组件并不限制于接收单个子组件。你可能会传递多个子组件, 甚至会为这些子组件 (children) 封装多个单独的“接口 (slots)”, [正如这里的文档所列举的](#)

```
function Page(props) {
  const user = props.user;
  const content = <Feed user={user} />;
  const topBar = (
    <NavigationBar>
      <Link href={user.permalink}>
        <Avatar user={user} size={props.avatarSize} />
      </Link>
    </NavigationBar>
  );
  return (
    <PageLayout
      topBar={topBar}
```

```
    content={content}
  />
);
}
```

这种模式足够覆盖很多场景了，在这些场景下你需要将子组件和直接关联的父组件解耦。如果子组件需要在渲染前和父组件进行一些交流，你可以进一步使用 `render props`。

但是，有的时候在组件树中很多不同层级的组件需要访问同样的一批数据。Context 能让你将这些数据向组件树下所有的组件进行“广播”，所有的组件都能访问到这些数据，也能访问到后续的数据更新。使用 `context` 的通用的场景包括管理当前的 locale, theme, 或者一些缓存数据，这比替代方案要简单的多。

API

`React.createContext` [">#reactcreatecontext](#)

```
const MyContext = React.createContext(defaultValue);
```

创建一个 Context 对象。当 React 渲染一个订阅了这个 Context 对象的组件，这个组件会从组件树中离自身最近的那个匹配的 `Provider` 中读取到当前的 context 值。

只有当组件所处的树中没有匹配到 Provider 时，其 `defaultValue` 参数才会生效。此默认值有助于在不使用 Provider 包装组件的情况下对组件进行测试。注意：将 `undefined` 传递给 Provider 的 `value` 时，消费组件的 `defaultValue` 不会生效。

`Context.Provider` [">#contextprovider](#)

```
<MyContext.Provider value={/* 某个值 */}>
```

每个 Context 对象都会返回一个 Provider React 组件，它允许消费组件订阅 context 的变化。

Provider 接收一个 `value` 属性，传递给消费组件。一个 Provider 可以和多个消费组件有对应关系。多个 Provider 也可以嵌套使用，里层的会覆盖外层的数据。

当 Provider 的 `value` 值发生变化时，它内部的所有消费组件都会重新渲染。从 Provider 到其内部 consumer 组件（包括 `.contextType` 和 `useContext`）的传播不受制于 `shouldComponentUpdate` 函数，因此当 consumer 组件在其祖先组件跳过更新的情况下也能更新。

通过新旧值检测来确定变化，使用了与 `Object.is` 相同的算法。

注意

当传递对象给 `value` 时，检测变化的方式会导致一些问题：详见[注意事项](#)。

Class.contextType {#classcontexttype}

```
class MyClass extends React.Component {
  componentDidMount() {
    let value = this.context;
    /* 在组件挂载完成后，使用 MyContext 组件的值来执行一些有副作用的操作 */
  }
  componentDidUpdate() {
    let value = this.context;
    /* ... */
  }
  componentWillUnmount() {
    let value = this.context;
    /* ... */
  }
  render() {
    let value = this.context;
    /* 基于 MyContext 组件的值进行渲染 */
  }
}
MyClass.contextType = MyContext;
```

挂载在 class 上的 `contextType` 属性可以赋值为由 `React.createContext()` 创建的 Context 对象。此属性可以让你使用 `this.context` 来获取最近 Context 上的值。你可以在任何生命周期中访问到它，包括 `render` 函数中。

注意：

你只通过该 API 订阅单一 context。如果你想订阅多个，阅读[使用多个 Context](#)章节

如果你正在使用实验性的 `public class fields` 语法，你可以使用 `static` 这个类属性来初始化你的 `contextType`。

```
class MyClass extends React.Component {
  static contextType = MyContext;
  render() {
    let value = this.context;
    /* 基于这个值进行渲染工作 */
  }
}
```

Context.Consumer {#contextconsumer}

```
<MyContext.Consumer>
  {value => /* 基于 context 值进行渲染 */}
</MyContext.Consumer>
```

一个 React 组件可以订阅 context 的变更，此组件可以让你在[函数式组件](#)中可以订阅 context。

这种方法需要一个[函数作为子元素 \(function as a child\)](#)。这个函数接收当前的 context 值，并返回一个 React 节点。传递给函数的 `value` 值等价于组件树上方离这个 context 最近的 Provider 提供的 `value` 值。如果没有对应的 Provider，`value` 参数等同于传递给 `createContext()` 的 `defaultValue`。

注意

想要了解更多关于“函数作为子元素 (function as a child)”模式，详见 [render props](#)。

`Context.displayName` {#contextdisplayname}

context 对象接受一个名为 `displayName` 的 property，类型为字符串。React DevTools 使用该字符串来确定 context 要显示的内容。

示例，下述组件在 DevTools 中将显示为 MyDisplayName：

```
const MyContext = React.createContext(/* some value */);
MyContext.displayName = 'MyDisplayName';

<MyContext.Provider> // "MyDisplayName.Provider" 在 DevTools 中
<MyContext.Consumer> // "MyDisplayName.Consumer" 在 DevTools 中
```

示例

动态 Context

一个更加复杂的方案是对上面的 theme 例子使用动态值 (dynamic values)：

theme-context.js [embed:context/theme-detailed-theme-context.js](#)

themed-button.js [embed:context/theme-detailed-themed-button.js](#)

app.js [embed:context/theme-detailed-app.js](#)

在嵌套组件中更新 Context

从一个在组件树中嵌套很深的组件中更新 context 是很有必要的。在这种场景下，你可以通过 context 传递一个函数，使得 consumers 组件更新 context：

theme-context.js [embed:context/updating-nested-context-context.js](#)

theme-toggler-button.js [embed:context/updating-nested-context-theme-toggler-button.js](#)

app.js [embed:context/updating-nested-context-app.js](#)

消费多个 Context

为了确保 context 快速进行重渲染，React 需要使每一个 consumers 组件的 context 在组件树中成为一个单独的节点。

[embed:context/multiple-contexts.js](#)

如果两个或者更多的 context 值经常被一起使用，那你可能要考虑一下另外创建你自己的渲染组件，以提供这些值。

注意事项

因为 context 会根据引用标识来决定何时进行渲染（本质上是 `value` 属性值的浅比较），所以这里可能存在一些陷阱，当 provider 的父组件进行重渲染时，可能会在 consumers 组件中触发意外的渲染。举个例子，当每一次 Provider 重渲染时，由于 `value` 属性总是被赋值为新的对象，以下的代码会重新渲染下面所有的 consumers 组件：

```
embed:context/reference-caveats-problem.js
```

为了防止这种情况，将 `value` 状态提升到父节点的 `state` 里：

```
embed:context/reference-caveats-solution.js
```

过时的 API

注意

先前 React 使用实验性的 context API 运行，旧的 API 将会在所有 16.x 版本中得到支持，但用到它的应用应该迁移到新版本。过时的 API 将在未来的 React 版本中被移除。阅读[过时的 context 文档](#)了解更多。

[Go to TOC](#)

创建新的 React 应用

使用集成的工具链，以实现最佳的用户和开发人员体验。

本页将介绍一些流行的 React 工具链，它们有助于完成如下任务：

- 扩展文件和组件的规模。
- 使用来自 npm 的第三方库。
- 尽早发现常见错误。
- 在开发中实时编辑 CSS 和 JS。
- 优化生产输出。

本页推荐的工具链**无需配置即可开始使用**。

你可能不需要工具链

如果你没有碰到上述的问题，或者还不习惯使用 JavaScript 工具，可以考虑把 React 作为普通的 `<script>` 标记添加到 [HTML 页面上](#)，以及[使用可选的 JSX](#)。

这也是将 React 集成到现有网站最简单的方式。如果你认为更大的工具链有所帮助，可以随时添加！

推荐的工具链

React 团队主要推荐这些解决方案：

- 如果你是在**学习 React 或创建一个新的单页应用**，请使用 [Create React App](#)。
- 如果你是在**用 Node.js 构建服务端渲染的网站**，试试 [Next.js](#)。
- 如果你是在**构建内容主导的静态网站**，试试 [Gatsby](#)。
- 如果你是在**打造组件库或将 React 集成到现有代码仓库**，尝试[更灵活的工具链](#)。

Create React App

[Create React App](#) 是一个用于**学习 React** 的舒适环境，也是用 React **创建新的单页应用**的最佳方式。

它会配置你的开发环境，以便使你能够使用最新的 JavaScript 特性，提供良好的开发体验，并为生产环境优化你的应用程序。你需要在你的机器上安装 [Node >= 14.0.0](#) 和 [npm >= 5.6](#)。要创建项目，请执行：

```
npx create-react-app my-app  
cd my-app  
npm start
```

注意

第一行的 `npx` 不是拼写错误 —— 它是 [npm 5.2+](#) 附带的 package 运行工具。

Create React App 不会处理后端逻辑或操纵数据库；它只是创建一个前端构建流水线 (build pipeline)，所以你可以使用它来配合任何你想使用的后端。它在内部使用 [Babel](#) 和 [webpack](#)，但你无需了解它们的任何细节。

当你准备好部署到生产环境时，执行 `npm run build` 会在 `build` 文件夹内生成你应用的优化版本。你能从[它的 README](#) 和[用户指南](#)了解 Create React App 的更多信息。

Next.js

[Next.js](#) 是一个流行的、轻量级的框架，用于配合 React 打造静态化和服务端渲染应用。它包括开箱即用的样式和路由方案，并且假定你使用 [Node.js](#) 作为服务器环境。

从 [Next.js 的官方指南](#)了解更多。

Gatsby

[Gatsby](#) 是用 React 创建静态网站的最佳方式。它让你能使用 React 组件，但输出预渲染的 HTML 和 CSS 以保证最快的加载速度。

从 Gatsby 的[官方指南](#)和[入门示例集](#)了解更多。

更灵活的工具链

以下工具链为 React 提供更多更具灵活性的方案。推荐给更有经验的使用者：

- [Neutrino](#) 把 [webpack](#) 的强大功能和简单预设结合在一起。并且包括了 [React 应用](#)和 [React 组件](#)的预设。
- [Nx](#) 是针对全栈 monorepo 的开发工具包，其内置了 React, Next.js, Express 等。
- [Parcel](#) 是一个快速的、零配置的网页应用打包器，并且可以[搭配 React 一起工作](#)。
- [Razzle](#) 是一个无需配置的服务端渲染框架，但它提供了比 Next.js 更多的灵活性。

从头开始打造工具链

一组 JavaScript 构建工具链通常由这些组成：

- 一个 **package 管理器**，比如 [Yarn](#) 或 [npm](#)。它能让你充分利用庞大的第三方 package 的生态系统，并且轻松地安装或更新它们。
- 一个**打包器**，比如 [webpack](#) 或 [Parcel](#)。它能让你编写模块化代码，并将它们组合在一起成为小的 package，以优化加载时间。
- 一个**编译器**，例如 [Babel](#)。它能让你编写的新版本 JavaScript 代码，在旧版浏览器中依然能够工作。

如果你倾向于从头开始打造你自己的 JavaScript 工具链，可以[查看这个指南](#)，它重新创建了一些 Create React App 的功能。

别忘了确保你自定义的工具链对生产环境进行了正确配置。

[Go to TOC](#)

跨源资源共享错误

注意：

以下部分仅适应于开发模式下的 React。生产模式下的异常处理由常规的 try/catch 语句完成。

在[开发模式](#)下,React 使用全局事件 `错误` 处理来保留在浏览器开发者工具的 "异常时暂停" 行为。它还会将错误记录到开发者控制台.

如果错误由一个[不同的源](#)抛出, 浏览器将会掩盖它的细节, React也不能记录原始的错误信息。这是一个由浏览器采取的安全预防措施来避免敏感信息的泄露。

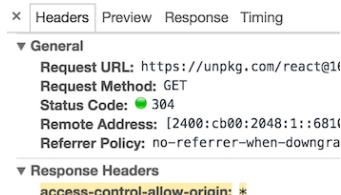
你可以抛出一个同源策略的错误来简化开发/调试模式过程。以下是一些常见的引起跨源资源共享的原因以及处理他们的方法。

CDN

当从一个 CDN 加载 React (或者其他可能抛出错误的库), 在你的 `<script>` 标签中添加 `crossorigin` 属性：

```
<script crossorigin src="..."></script>
```

并且确保 CDN 以 `Access-Control-Allow-Origin: *` HTTP 请求头应答：



`Access-Control-Allow-Origin: *`

Webpack

源码映射

一些 JavaScript 打包器可能在开发过程中以 `eval` 包装应用代码。(例如, Webpack 会用这个如果设置 `devtool` 的值任意包含"eval")。这个可能会引起被视为跨源错误。

如果你使用 Webpack, 我们推荐在开发中使用 `cheap-module-source-map` 设置来避免这个问题。

代码拆分

如果你的应用被拆分成多个包, 这些包可能会使用 JSONP被加载。这些包会被视为跨源资源, 从而可能会引发错误。

为了处理这个错误，在标签 `<script>` 中为 `crossorigin` 属性添加 `crossOriginLoading` 设置，从而为 JSONP 生成请求。

设计原则

编写该文档的目的是，使开发者更易于了解我们如何决策 React（应该做哪些，不应该做哪些），以及我们的发展理念。我们非常欢迎来自社区的贡献，但如若违背这些理念，实非我们所愿。

注意：

文章描述了 React 自身的设计原则，而非 React 组件或应用，阅读者需要对 React 有深入的理解。

如需 React 的入门文档，查看 [React 哲学](#)。

组合

组件之间的组合是 React 的重要特征。不同开发者写的组件应该可以一起正常执行。给一个组件添加功能，而不会对整个代码库造成涟漪似的变化，这对我们很重要。

比如，应该可以在不影响任何使用它的组件的情况下，将一些内部 state 引入该组件。类似的，在必要的情况下可以在任何组件里添加一些初始化和销毁的代码。

在组件中使用 state 或者生命周期函数没什么“不好”。跟所有强大的特性一样，应该适度使用它们，我们并不打算移除他们。相反，我们认为他们是 React 之所以好用的一部分。我们未来也许会启用[更多函数模式](#)，但内部 state 和生命周期函数都会在里面。

人们常常认为组件“只是函数”，但在我们看来，组件要好用的话，需要的不止这些。在 React 中，组件描述了任何可组合的行为，包含渲染、生命周期和 state。一些类似 [Relay](#) 的外部库给组件带来了其他增强功能，比如描述数据之间的依赖关系。有可能这些做法会以某种形式回到 React 中。

共用抽象

一般而言我们[拒绝添加](#)一些开发者可以实现的特性。我们不想因为无用的库代码使得大家的应用变的累赘，然而也有特例。

比如，如果 React 不支持内部 state 或者生命周期函数，大家会为此创建自己的抽象。当有多种抽象竞争的时候，React 不能强制使用或利用这些抽象中的任何一个。React 必须选择最基本的共同点。

这就是我们增加 React 特性的原因。如果我们发现很多组件以不兼容或者不高效的方式实现了某些特性，我们会倾向在 React 中实现它。我们轻易不这样做，只有我们非常确定提高抽象层级有助于整个生态系统时我们才会这样做。State、生命周期函数、跨浏览器事件的正规化都是很好的范例。

我们总是和社区一起商议这样的优化提议。你可以在 React 问题跟踪的 “[big picture](#)” 标签上找到一些这样的讨论。

应急方案

React 是务实的，Facebook 的产品需求驱使它这样。尽管 React 受一些目前还非主流的编程思想比如函数式编程的影响，这个项目的一个明确目标是广泛地接触具有不同技能和经验的开发者。

如果要废弃某个我们不喜欢的模式，我们有责任在废弃之前，考虑所有已知的用例并且教育社区这些模式的[替代做法](#)。如果某些有助于开发应用的模式很难以声明式的方式表达，我们会提供一个[命令式的 API](#)。如果我们发现很多应用中必要的模式我们找不到一个完美的API，我们会[提供一个临时欠佳的 API](#)，只要以后可以移除它并且方便后续的优化。

稳定性

我们重视 API 的稳定性。在 Facebook，我们有超过 5 万多个组件在使用 React。很多其他公司，包括 Twitter 和 Airbnb 也是 React 的重度用户。这就是我们一般不愿意变更公共 API 或行为的原因。

然而我们认为“没有变更”的这种稳定性被高估了，它很快会演变成停滞不前。我们偏向这样理解稳定性：“在生产环境大规模使用，当需要变更时有一个明确的（自动化更好）迁移路径。”。

当我们废弃一个模式时，我们会研究它在 Facebook 内部的使用情况并添加废弃警告。这样我们可以评估变更的影响范围。有时觉得时机太早了，我们就不作变更。我们需要更策略化的思考来让代码库准备好这样的变更。

如果我们自信地认为这次改动破坏性不是非常大，而且迁移策略对所有用户场景都可行，我们会在开源社区发布废弃警告。我们和很多 Facebook 以外的 React 用户联系紧密，而且我们会监控流行的开源项目并指导他们解决这些废弃警告。

只考虑 Facebook 里的 React 的代码库大小，成功的内部迁移往往是个好兆头：其他公司迁移也不会有困难。然而时不时会有人提出我们没有考虑到的场景，我们会给他们一个紧急出路，或者反思我们的做法。

没有充分的理由我们不会废弃任何特性。我们意识到有时废弃警告会让开发者沮丧，但我们添加废弃警告是因为它们为社区中的很多人认为有价值的优化和新特性扫除了障碍。

例如，我们在 React 15.2.0 中添加了一个[未知 DOM 属性的警告](#)，很多项目因此受到影响。然而修复这些警告非常重要，因为我们在 React 中新增支持[自定义属性](#)。以往的每一次废弃警告都有像这样的考量在里面。

我们添加废弃警告时会保留到当前大版本，在[下一次大版本中改变行为](#)。如果这其中涉及到大量重复性的人力工作，我们会发布一个[代码更改](#)脚本自动化大部分的改动。Codemods 使我们能够在庞大的代码库中继续前行，我们也推荐大家使用。

你可以在 [react-codemod](#) 仓库中找到我们发布的 codemods。

互操作性

我们很看重与已有系统的互操作性和渐进式的采用。Facebook 有一个庞大的非 React 的代码库。Facebook 的网站混合使用了名为 XHP 的服务端组件系统、React 之前的内部 UI 库和 React。任何产品团队能够[开始在小功能上使用 React](#)而不是重写他们的代码，这对我们很重要。

这就是 React 提供应急方案的原因：让不同的模型协同工作，让不同的 UI 库协同工作。你可以把一个已有的命令式 UI 封装成声明式的组件，反之亦然。这对渐进采用而言至关重要。

调度

即便你的组件以 function 的方式声明，在 React 中你也并不会直接调用它们。每个组件返回一个该渲染什么的描述，该描述会包含开发者写的组件如 `<LikeButton>` 和 平台特定的组件如 `<div>`。由 React 决定在未来的某个时间点展开 `<LikeButton>`，并根据组件的渲染结果递归地把这些变更实际应用到 UI 树上。

虽然只是微小的区别，但这样做意义重大。因为你不需要调用组件方法而是让 React 调用它，这意味着如果必要 React 可以延迟调用。在 React 当前的实现中，React 在单个 tick 周期中递归地走完这棵树，然后调用整个更新后树的渲染方法。但是以后 React 可能会[延迟一些更新操作来防止掉帧](#)。

这在 React 的设计中很常见。有一些流行的库实现了“push”模式，即当新数据到达时再计算。然而 React 坚持“pull”模式，即计算可以延迟到必要时再执行。

React 不是一个常规的数据处理库，它是开发用户界面的库。我们认为 React 在一个应用中的位置很独特，它知道当前哪些计算当前是相关的，哪些不是。

如果不在当前屏幕，我们可以延迟执行相关逻辑。如果数据到达的速度快过帧速，我们可以合并、批量更新。我们优先执行用户交互（例如按钮点击形成的动画）的工作，延后执行相对不那么重要的后台工作（例如渲染刚从网络上下载的新内容），从而避免掉帧。

要清楚我们现在还没有利用调度。然而，我们之所以偏好自己控制调度以及异步 `setState()`，是因为拥有了选择的自由度。

如果我们允许用户使用在一些变体的[函数反应式编程](#)范式中常见的“推”模式直接拼接视图，我们将会很难获得调度的控制权。

React 的一个关键目标是在把控制权转交给 React 之前执行的用户代码量最少。这确保 React 保持调度的能力，并根据它所知道的 UI 的情况把工作切分成小块处理。

在团队内部有个笑话，React 本该叫做“调度”因为 React 不想变得完全“反应式（reactive）”。

开发者体验

提供好的开发者体验对我们很重要。

例如，我们维护了[React DevTools](#)，它让大家在 Chrome 和 Firefox 浏览器中可以检查 React 组件树。我们听说它大幅提高了 Facebook 的工程师和社区的生产效率。

我们还提供了对开发有所帮助的开发者警告。比如你以浏览器不理解的方式嵌套标签或者你在编写 API 时常见的错别字，React 会警告你。开发者警告和相关的检查导致了 React 的开发版本比生产版本速度慢。

Facebook 内部的使用模式帮助我们了解常见的错误有哪些，以及如何提前预防。当我们添加新特性时，我们尝试去预测可能会发生的错误并发出警告。

我们一直在寻找提高开发者体验的方法。我们很乐意听取大家的建议，接受大家的贡献，把开发者体验做的更好。

调试

当发生错误时，你有一些线索可以追溯到代码库中的具体代码，这很重要。在 React 中，`props` 和 `state` 就是线索。

当你看到屏幕上出现错误时，你可以打开 React 开发者工具，找到负责渲染的组件，检查它的 `props` 和 `state` 是否正确。如果正确，就能大概知道问题可能出在组件的 `render()` 方法，或者某个被 `render()` 调用的方法。问题显而易见。

如果 `state` 发生错误，你就知道问题在于这个文件中的某个 `setState()` 调用，定位和修复也相对简单，因为在单个文件中 `setState()` 调用次数很少。

如果 `props` 出现错误，你可以在检查器中沿着树向上排查，查找到第一个因为向下传递了错误的 `props` 而“污染了这口井”的组件。

能够以当前的 `props` 和 `state` 这种形式追溯到其产生的任何 UI，这种能力对 React 来说非常重要。`state` 不会包裹在闭包和组合子中，并且在 React 中可以直接获取，这是 React 一个非常明确的设计目标。

尽管 UI 是动态变化的，但是我们认为 state 和 props 的同步执行的 `render()` 函数使得调试变成了无聊但是有限的步骤，而不是瞎猜。我们在 React 里保留了这个限制，即使它使调试在某些场景下变得更加困难，比如复杂的动画。

配置

我们发现全局的运行时配置会造成很多问题。

比如我们时常收到这样的请求，实现类似于 `React.configure(options)` 或者 `React.register(component)` 的方法。然而这样做会出现很多问题，而我们对此并没有很好的解决方案。

如果有人在第三方组件库中调用了这样的方法怎么办？如果一个 React 应用内嵌了另一个 React 应用，而且他们需要的配置不兼容怎么办？一个第三方组件怎么申明它需要某个特定配置呢？我们认为全局的配置与组合搭配效果不好。既然组合是 React 的核心，那么我们不在代码里提供全局的配置。

然而我们在构建层面提供了一些全局的配置。比如我们提供了独立的开发和生产环境的构建。我们后面也许会[增加一个分析构建](#)。我们对考虑使用其他构建参数保持开放态度。

DOM 之外

我们认为 React 的价值在于它使我们写出更少 bug 的组件，而且组件很方便组合。React 当初的渲染目标是 DOM，但 [React Native](#) 对 Facebook 和社区来说同样重要。

React 的一个重要设计约束是要渲染引擎无关。这在内部呈现增加了一些开销，另一方面，对内核的任何优化将对所有平台都有益。

单一的编程模型使我们能够围绕产品形成工程团队，而不是围绕平台。目前来看这样的取舍对我们来说很值得。

实现

相比优雅的实现，我们更想提供尽可能优雅的 API。现实是不完美的，如果能让用户无需写这些代码，在合适的范围内，我们偏向把丑陋的代码写在库里。我们评估新代码时，我们看中的是正确地实现、高性能并且能够带来良好的开发体验，优雅是第二位的。

我们宁可写无聊的代码也不写要聪明的代码。代码是一次性的而且经常变更，所以[除非极度必要，不引入新的内部抽象](#)很重要。很方便移动、改动和删除的啰嗦的代码比过早抽象且难以变更的优雅的代码更好。

针对工具优化

React 一些常用的 API 名字很冗长。比如，我们采用 `componentDidMount()` 而非 `didMount()` 或者 `onMount()`。这是[有意为之的](#)，目的是使得和 React 的交互点具有高可见性。

在像 Facebook 这样庞大的代码库中，能够搜索某些特定 API 的使用很重要。我们重视清晰冗长的名字，特别是在一些需要保守使用的特性上。比如，`dangerouslySetInnerHTML` 在代码评审中就很容易被发现。

针对搜索优化很重要，因为我们依赖 [codemods](#) 做不兼容的变更。我们希望非常容易、安全地在代码库中应用大量自动化变更，独特冗长的名字帮助了我们。类似地，独特的命名使得编写自定义 React 用法的[提示规则](#)变得很容易，无需担心潜在的错误匹配。

[JSX](#) 也类似这样。尽管 React 不强制使用 JSX，在 Facebook 我们大量使用 JSX，因为它既好看又实用。

在我们的代码库中，JSX 给和 React 元素树打交道的工具提供了明确的提示。这使得构建时优化成为可能，比如[提升常量元素](#)、安全地进行代码提示、codemod 内部组件用法、在代码警告中[包含 JSX 源码定位](#)。

内部测试

我们全力解决社区提出的问题。但我们可能会优先处理在 Facebook 内部遇到的*同样的*问题。也许这很反直觉，但是我们认为这是社区相信 React 的主要原因。

内部的重度使用使我们坚信 React 不会凭空消失。Facebook 创建了 React 是来解决它的问题的。React 给 Facebook 带来了现实的商业价值，并且在很多产品中使用。在内部测试意味着我们的目光保持敏锐，有着前进的重点方向。

这不代表我们会忽视社区提出的问题。比如，即便我们内部并不依赖他们，我们仍增加了 React 对 [web components](#) 和 [SVG](#) 的支持。我们倾听着大家的痛点，并全力解决他们。是社区使 React 变得与众不同，所以我们很荣幸可以回报社区。

在 Facebook 发布了很多开源的项目之后，我们学到了让大家都开心的做法会导致项目没有重点，成长不起来。相反，我们发现选择一小部分群体，重点关注如何使他们开心带来了积极的净效应。这正是 React 的做法，到目前为止解决 Facebook 产品团队的问题很好的传递到了开源社区。

这种做法的不好之处是有时对于 Facebook 团队无需关心的事，比如“起步”的体验，我们不能够给予足够的重视。我们已经觉察到了，我们正在着手思考如何做才能使社区里的每个人都受益，不重蹈以前开源项目的覆辙。

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。

注意

错误边界无法捕获以下场景中产生的错误：

- 事件处理 ([了解更多](#))
- 异步代码（例如 `setTimeout` 或 `requestAnimationFrame` 回调函数）
- 服务端渲染
- 它自身抛出来的错误（并非它的子组件）

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。

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    // 更新 state 使下一次渲染能够显示降级后的 UI
    return { hasError: true };
  }

  componentDidCatch(error, errorInfo) {
    // 你同样可以将错误日志上报给服务器
    logErrorToMyService(error, errorInfo);
  }

  render() {
    if (this.state.hasError) {
      // 你可以自定义降级后的 UI 并渲染
      return <h1>Something went wrong.</h1>;
    }
  }
}
```

```

    return this.props.children;
}
}

```

然后你可以将它作为一个常规组件去使用：

```

<ErrorBoundary>
  <MyWidget />
</ErrorBoundary>

```

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。

在线演示

查看 [定义和使用错误边界的示例](#)。

错误边界应该放置在哪？

错误边界的粒度由你来决定，可以将其包装在最顶层的路由组件并为用户展示一个“Something went wrong”的错误信息，就像服务端框架经常处理崩溃一样。你也可以将单独的部件包装在错误边界以保护应用其他部分不崩溃。

未捕获错误(Uncaught Errors)的新行为

这一改变具有重要意义，**自 React 16 起，任何未被错误边界捕获的错误将会导致整个 React 组件树被卸载**。

我们对这一决定有过一些争论，但根据我们的经验，把一个错误的 UI 留在那比完全移除它要更糟糕。例如，在类似 Messenger 的产品中，把一个异常的 UI 展示给用户可能会导致用户将信息错发给别人。同样，对于支付类应用而言，显示错误的金额也比不呈现任何内容更糟糕。

此变化意味着当你迁移到 React 16 时，你可能会发现一些已存在你应用中但未曾注意到的崩溃。增加错误边界能够让你在应用发生异常时提供更好的用户体验。

例如，Facebook Messenger 将侧边栏、信息面板、聊天记录以及信息输入框包装在单独的错误边界中。如果其中的某些 UI 组件崩溃，其余部分仍然能够交互。

我们也鼓励使用 JS 错误报告服务（或自行构建），这样你能了解关于生产环境中出现的未捕获异常，并将其修复。

组件栈追踪

在开发环境下，React 16 会把渲染期间发生的所有错误打印到控制台，即使该应用意外的将这些错误掩盖。除了错误信息和 JavaScript 栈外，React 16 还提供了组件栈追踪。现在你可以准确地查看发生在组件树内的错误信息：

```
▶ React caught an error thrown by BuggyCounter. You should fix this error in your code. react-dom.development.js:7708
React will try to recreate this component tree from scratch using the error boundary you provided, ErrorBoundary.

Error: I crashed!

The error is located at:
in BuggyCounter (created by App)
in ErrorBoundary (created by App)
in div (created by App)
in App
```

你也可以在组件栈追踪中查看文件名和行号，这一功能在 [Create React App](#) 项目中默认开启：

```
▶ React caught an error thrown by BuggyCounter. You should fix this error in your code. react-dom.development.js:7708
React will try to recreate this component tree from scratch using the error boundary you provided, ErrorBoundary.

Error: I crashed!

The error is located at:
in BuggyCounter (at App.js:26)
in ErrorBoundary (at App.js:21)
in div (at App.js:8)
in App (at index.js:5)
```

如果你没有使用 Create React App，可以手动将[该插件](#)添加到你的 Babel 配置中。注意它仅用于开发环境，在生产环境必须将其禁用。

注意

组件名称在栈追踪中的显示依赖于 `Function.name` 属性。如果你想要支持尚未提供该功能的旧版浏览器和设备（例如 IE 11），考虑在你的打包（bundled）应用程序中包含一个 `Function.name` 的 polyfill，如 `function.name-polyfill`。或者，你可以在所有组件上显式设置 `displayName` 属性。

关于 `try/catch` ?

`try / catch` 很棒但它仅能用于命令式代码（imperative code）：

```
try {
  showButton();
} catch (error) {
  // ...
}
```

然而，React 组件是声明式的并且具体指出什么需要被渲染：

`<Button />`

错误边界保留了 React 的声明性质，其行为符合你的预期。例如，即使一个错误发生在 `componentDidUpdate` 方法中，并且由某一个深层组件树的 `setState` 引起，其仍然能够冒泡到最近的错误边界。

关于事件处理器

错误边界无法捕获事件处理器内部的错误。

React 不需要错误边界来捕获事件处理器中的错误。与 `render` 方法和生命周期方法不同，事件处理器不会在渲染期间触发。因此，如果它们抛出异常，React 仍然能够知道需要在屏幕上显示什么。

如果你需要在事件处理器内部捕获错误，使用普通的 JavaScript `try` / `catch` 语句：

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = { error: null };
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    try {
      // 执行操作，如有错误则会抛出
    } catch (error) {
      this.setState({ error });
    }
  }

  render() {
    if (this.state.error) {
      return <h1>Caught an error.</h1>
    }
    return <button onClick={this.handleClick}>Click Me</button>
  }
}
```

请注意上述例子只是演示了普通的 JavaScript 行为，并没有使用错误边界。

自 React 15 的命名更改

React 15 中有一个支持有限的错误边界方法 `unstable_handleError`。此方法不再起作用，同时自 React 16 beta 发布起你需要在代码中将其修改为 `componentDidCatch`。

对此，我们已提供了一个 [codemod](#) 来帮助你自动迁移你的代码。

[Go to TOC](#)

Error Decoder

In the minified production build of React, we avoid sending down full error messages in order to reduce the number of bytes sent over the wire.

We highly recommend using the development build locally when debugging your app since it tracks additional debug info and provides helpful warnings about potential problems in your apps, but if you encounter an exception while using the production build, this page will reassemble the original text of the error.

AJAX and APIs

如何在 React 中发起 AJAX 请求？

在 React 开发中，你能使用任何你喜欢的 AJAX 库，比如社区比较流行的 [Axios](#), [jQuery AJAX](#)，或者是浏览器内置的 `window.fetch`。

应该在 React 组件的哪个生命周期函数中发起 AJAX 请求？

我们推荐你在 `componentDidMount` 这个生命周期函数中发起 AJAX 请求。这样做你可以拿到 AJAX 请求返回的数据并通过 `setState` 来更新组件。

示例：使用 AJAX 请求结果去改变组件内部 state

下面这个组件演示了如何在 `componentDidMount` 中发起 AJAX 请求去更新组件的 state。

示例 API 返回如下的 JSON 对象：

```
{
  "items": [
    { "id": 1, "name": "Apples", "price": "$2" },
    { "id": 2, "name": "Peaches", "price": "$5" }
  ]
}
```

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      error: null,
      isLoading: false,
      items: []
    };
  }

  componentDidMount() {
    fetch("https://api.example.com/items")
      .then(res => res.json())
      .then(
        (result) => {
          this.setState({
            isLoading: true,
            items: result.items
          });
        },
        // 注意：需要在此处处理错误
        // 而不是使用 catch() 去捕获错误
        // 因为使用 catch 去捕获异常会掩盖掉组件本身可能产生的 bug
      (error) => {
        this.setState({
          isLoading: true,
          error
        });
      }
    );
  }
}
```

```

}

render() {
  const { error, isLoading, items } = this.state;
  if (error) {
    return <div>Error: {error.message}</div>;
  } else if (!isLoading) {
    return <div>Loading...</div>;
  } else {
    return (
      <ul>
        {items.map(item => (
          <li key={item.id}>
            {item.name} {item.price}
          </li>
        ))}
      </ul>
    );
  }
}

```

这里与 Hook 的实现相同：

```

function MyComponent() {
  const [error, setError] = useState(null);
  const [isLoading, setIsLoaded] = useState(false);
  const [items, setItems] = useState([]);

  // 注意：空的依赖数组 []
  // 表示这个 useEffect
  // 与 componentDidMount() 类似，只运行一次
  useEffect(() => {
    fetch("https://api.example.com/items")
      .then(res => res.json())
      .then(
        (result) => {
          setIsLoaded(true);
          setItems(result);
        },
        // 注意：需要在此处处理错误
        // 而不是使用 catch() 去捕获错误
        // 因为使用 catch 去捕获异常会掩盖掉组件本身可能产生的 bug
        (error) => {
          setIsLoaded(true);
          setError(error);
        }
      )
    }, [])

  if (error) {
    return <div>Error: {error.message}</div>;
  } else if (!isLoading) {
    return <div>Loading...</div>;
  } else {
    return (
      <ul>
        {items.map(item => (
          <li key={item.id}>
            {item.name} {item.price}
          </li>
        ))}
      </ul>
    );
  }
}

```

```
)});  
  </ul>  
};  
};
```

Babel, JSX 及构建过程

必须在 React 中使用 JSX 吗？

不是必须的！请查阅 “[不使用 JSX](#)” 以了解更多信息。

必须在 React 中使用 ES6 (+) 吗？

不是必须的！请查阅 “[不使用 ES6](#)” 以了解更多信息。

怎样才能在 JSX 中编写注释？

```
<div>
  {/* 注释写在这里 */}
  Hello, {name}!
</div>
```

```
<div>
  /* 多行注释
   * 也同样有效。
   */
  Hello, {name}!
</div>
```

传递函数给组件

如何将事件处理器(比如 onClick)传递给组件?

可以将事件处理器和其他函数作为 props 传递给子组件：

```
<button onClick={this.handleClick}>
```

如果需要在事件处理器中访问父组件，还需要为该函数绑定组件实例（参见下文）。

如何为函数绑定组件实例？

有以下几种方式可以确保函数可以访问组件属性，比如 `this.props` 和 `this.state`，这取决于使用的语法和构建步骤。

在构造函数中绑定 (ES2015)

```
class Foo extends Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick() {
    console.log('Click happened');
  }
  render() {
    return <button onClick={this.handleClick}>Click Me</button>;
  }
}
```

Class Properties (ES2022)

```
class Foo extends Component {
  handleClick = () => {
    console.log('Click happened');
  };
  render() {
    return <button onClick={this.handleClick}>Click Me</button>;
  }
}
```

在 Render 中的绑定

```
class Foo extends Component {
  handleClick() {
    console.log('Click happened');
  }
  render() {
    return <button onClick={this.handleClick.bind(this)}>Click Me</button>;
  }
}
```

注意：

在 render 方法中使用 `Function.prototype.bind` 会在每次组件渲染时创建一个新的函数，可能会影响性能（参见下文）。

在 Render 中使用箭头函数

```
class Foo extends Component {
  handleClick() {
    console.log('Click happened');
  }
  render() {
    return <button onClick={() => this.handleClick()}>Click Me</button>;
  }
}
```

注意：

在 render 方法中使用箭头函数也会在每次组件渲染时创建一个新的函数，这会破坏基于恒等比较的性能优化。

可以在 render 方法中使用箭头函数吗？

一般来说是可以的，并且使用箭头函数是向回调函数传递参数的最简单的办法。

但是如果遇到了性能问题，一定要进行优化！

为什么绑定是必要的？

在 JavaScript 中，以下两种写法是不等价的：

```
obj.method();
```

```
var method = obj.method;
method();
```

`bind` 方法确保了第二种写法与第一种写法相同。

使用 React，通常只需要绑定传递给其他组件的方法。例如，`<button onClick={this.handleClick}>` 是在传递 `this.handleClick`，所以需要绑定它。但是，没有必要绑定 `render` 方法或生命周期方法：我们并没有将它们传递给其他的组件。

[Yehuda Katz 的文章](#) 详细解释了什么是绑定，以及函数在 JavaScript 中怎么起作用。

为什么我的函数每次组件渲染时都会被调用？

确保你在传递一个函数给组件时，没有调用这个函数：

```
render() {
  // Wrong: handleClick is called instead of passed as a reference!
  return <button onClick={this.handleClick()}>Click Me</button>
}
```

正确做法是，传递函数本身（不带括号）：

```
render() {
  // Correct: handleClick is passed as a reference!
  return <button onClick={this.handleClick}>Click Me</button>
}
```

如何传递参数给事件处理器或回调？

可以使用箭头函数包裹事件处理器，并传递参数：

```
<button onClick={() => this.handleClick(id)} />
```

以上代码和调用 `.bind` 是等价的：

```
<button onClick={this.handleClick.bind(this, id)} />
```

示例：通过箭头函数传递参数

```
const A = 65 // ASCII character code

class Alphabet extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      justClicked: null,
      letters: Array.from({length: 26}, (_, i) => String.fromCharCode(A + i))
    };
  }
  handleClick(letter) {
    this.setState({ justClicked: letter });
  }
  render() {
    return (
      <div>
        Just clicked: {this.state.justClicked}
        <ul>
          {this.state.letters.map(letter =>
            <li key={letter} onClick={() => this.handleClick(letter)}>
              {letter}
            </li>
          )}
        </ul>
      </div>
    );
  }
}
```

示例：通过 `data-attributes` 传递参数

同样的，也可以使用 DOM API 来存储事件处理器需要的数据。如果需要优化大量元素或使用依赖于 `React.PureComponent` 相等性检查的渲染树，请考虑使用此方法。

```
const A = 65 // ASCII character code

class Alphabet extends React.Component {
  constructor(props) {
```

```

super(props);
this.handleClick = this.handleClick.bind(this);
this.state = {
  justClicked: null,
  letters: Array.from({length: 26}, (_, i) => String.fromCharCode(A + i))
};

handleClick(e) {
  this.setState({
    justClicked: e.target.dataset.letter
  });
}

render() {
  return (
    <div>
      Just clicked: {this.state.justClicked}
      <ul>
        {this.state.letters.map(letter =>
          <li key={letter} data-letter={letter} onClick={this.handleClick}>
            {letter}
          </li>
        )}
      </ul>
    </div>
  )
}
}

```

怎样阻止函数被调用太快或者太多次？

如果你有一个 `onClick` 或者 `onScroll` 这样的事件处理器，想要阻止回调被触发的太快，那么可以限制执行回调的速度，可以通过以下几种方式做到这点：

- **节流**：基于时间的频率来进行抽样更改（例如 `_.throttle`）
- **防抖**：一段时间的不活动之后发布更改（例如 `_.debounce`）
- **`requestAnimationFrame` 节流**：基于 `requestAnimationFrame` 的抽样更改（例如 `raf-schd`）

可以看这个比较 `throttle` 和 `debounce` 的[可视化页面](#)

注意：

`_.debounce`、`_.throttle` 和 `raf-schd` 都提供了一个 `cancel` 方法来取消延迟回调。你需要在 `componentWillUnmount` 中调用该方法，或者对代码进行检查来保证在延迟函数有效期间内组件始终挂载。

节流

节流阻止函数在给定时间窗口内被调不能超过一次。下面这个例子会节流“click”事件处理器，使其每秒钟的只能调用一次。

```

import throttle from 'lodash.throttle';

class LoadMoreButton extends React.Component {
  constructor(props) {

```

```

super(props);
this.handleClick = this.handleClick.bind(this);
this.handleClickThrottled = throttle(this.handleClick, 1000);
}

componentWillUnmount() {
  this.handleClickThrottled.cancel();
}

render() {
  return <button onClick={this.handleClickThrottled}>Load More</button>;
}

handleClick() {
  this.props.loadMore();
}
}

```

防抖

防抖确保函数不会在上一次被调用之后一定量的时间内被执行。当必须进行一些费时的计算来响应快速派发的事件时（比如鼠标滚动或键盘事件时），防抖是非常有用的。下面这个例子以 250ms 的延迟来改变文本输入。

```

import debounce from 'lodash.debounce';

class Searchbox extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.emitChangeDebounced = debounce(this.emitChange, 250);
  }

  componentWillUnmount() {
    this.emitChangeDebounced.cancel();
  }

  render() {
    return (
      <input
        type="text"
        onChange={this.handleChange}
        placeholder="Search..."
        defaultValue={this.props.value}
      />
    );
  }

  handleChange(e) {
    this.emitChangeDebounced(e.target.value);
  }

  emitChange(value) {
    this.props.onChange(value);
  }
}

```

requestAnimationFrame 节流

`requestAnimationFrame` 是在浏览器中排队等待执行的一种方法，它可以在呈现性能的最佳时间执行。一个函数被 `requestAnimationFrame` 放入队列后将会在下一帧触发。浏览器会努力确保每秒 60 帧 (60fps)。然而，如果浏览器无法确保，那么自然会限制每秒的帧数。例如，某个设备可能只能处理每秒 30 帧，所以每秒只能得到 30 帧。使用 `requestAnimationFrame` 来节流是一种有用的技术，它可以防止在一秒中进行 60 帧以上的更新。如果一秒钟内完成 100 次更新，则会为浏览器带来额外的负担，而用户却无法感知到这些工作。

注意：

使用这个方法时只能获取某一帧中最后发布的值。也可以在 [MDN](#) 中看优化的示例。

```
import rafSchedule from 'raf-schd';

class ScrollListener extends React.Component {
  constructor(props) {
    super(props);

    this.handleScroll = this.handleScroll.bind(this);

    // Create a new function to schedule updates.
    this.scheduleUpdate = rafSchedule(
      point => this.props.onScroll(point)
    );
  }

  handleScroll(e) {
    // When we receive a scroll event, schedule an update.
    // If we receive many updates within a frame, we'll only publish the latest
    // value.
    this.scheduleUpdate({ x: e.clientX, y: e.clientY });
  }

  componentWillUnmount() {
    // Cancel any pending updates since we're unmounting.
    this.scheduleUpdate.cancel();
  }

  render() {
    return (
      <div
        style={{ overflow: 'scroll' }}
        onScroll={this.handleScroll}
      >
        
      </div>
    );
  }
}
```

测试速率限制

在测试速率限制的代码是否正确工作的时候，如果可以（对动画或操作）进行快进将会很有帮助。如果正在使用 `jest`，那么可以使用 `mock timers` 来快进。如果正在使用 `requestAnimationFrame` 节流，那么就会发现 `raf-stub` 是一个控制动画帧的十分有用的工具。

[Go to TOC](#)

Virtual DOM 及内核

什么是 Virtual DOM？

Virtual DOM 是一种编程概念。在这个概念里，UI 以一种理想化的，或者说“虚拟的”表现形式被保存于内存中，并通过如 ReactDOM 等类库使之与“真实的”DOM 同步。这一过程叫做[协调](#)。

这种方式赋予了 React 声明式的 API：您告诉 React 希望让 UI 是什么状态，React 就确保 DOM 匹配该状态。这使您可以从属性操作、事件处理和手动 DOM 更新这些在构建应用程序时必要的操作中解放出来。

与其将“Virtual DOM”视为一种技术，不如说它是一种模式，人们提到它时经常是要表达不同的东西。在 React 的世界里，术语“Virtual DOM”通常与[React 元素](#)关联在一起，因为它们都是代表了用户界面的对象。而 React 也使用一个名为“fibers”的内部对象来存放组件树的附加信息。上述二者也被认为是 React 中“Virtual DOM”实现的一部分。

Shadow DOM 和 Virtual DOM 是一回事吗？

不，他们不一样。Shadow DOM 是一种浏览器技术，主要用于在 web 组件中封装变量和 CSS。Virtual DOM 则是一种由 Javascript 类库基于浏览器 API 实现的概念。

什么是“React Fiber”？

Fiber 是 React 16 中新的协调引擎。它的主要目的是使 Virtual DOM 可以进行增量式渲染。[了解更多](#)。

组件状态

setState 实际做了什么？

`setState()` 会对一个组件的 `state` 对象安排一次更新。当 `state` 改变了，该组件就会重新渲染。

state 和 props 之间的区别是什么？

`props` ("properties" 的缩写) 和 `state` 都是普通的 JavaScript 对象。它们都是用来保存信息的，这些信息可以控制组件的渲染输出，而它们的一个重要的不同点就是：`props` 是传递给组件的（类似于函数的形参），而 `state` 是在组件内被组件自己管理的（类似于在一个函数内声明的变量）。

下面是一些不错的资源，可以用来进一步了解使用 `props` 或 `state` 的最佳时机：

- [Props vs State](#)
- [ReactJS: Props vs. State](#)

为什么 setState 给了我一个错误的值？

在 React 中，`this.props` 和 `this.state` 都代表着已经被渲染了的值，即当前屏幕上显示的值。

调用 `setState` 其实是异步的——不要指望在调用 `setState` 之后，`this.state` 会立即映射为新的值。如果你需要基于当前的 `state` 来计算出新的值，那你应该传递一个函数，而不是一个对象（详情见下文）。

代码不会像预期那样运行的示例：

```
incrementCount() {
  // 注意：这样 *不会* 像预期的那样工作。
  this.setState({count: this.state.count + 1});
}

handleSomething() {
  // 假设 `this.state.count` 从 0 开始。
  this.incrementCount();
  this.incrementCount();
  this.incrementCount();
  // 当 React 重新渲染该组件时，`this.state.count` 会变为 1，而不是你期望的 3。

  // 这是因为上面的 `incrementCount()` 函数是从 `this.state.count` 中读取数据的，
  // 但是 React 不会更新 `this.state.count`，直到该组件被重新渲染。
  // 所以最终 `incrementCount()` 每次读取 `this.state.count` 的值都是 0，并将它设为 1。

  // 问题的修复参见下面的说明。
}
```

参见下面的说明来修复这个问题。

我应该如何更新那些依赖于当前的 state 的 state 呢？

给 `setState` 传递一个函数，而不是一个对象，就可以确保每次的调用都是使用最新版的 `state`（见下面的说明）。

给 `setState` 传递一个对象与传递一个函数的区别是什么？

传递一个函数可以让你在函数内访问到当前的 `state` 的值。因为 `setState` 的调用是分批的，所以你可以链式地进行更新，并确保它们是一个建立在另一个之上的，这样才不会发生冲突：

```
incrementCount() {
  this.setState((state) => {
    // 重要：在更新的时候读取 `state`，而不是 `this.state`。
    return {count: state.count + 1}
  });
}

handleSomething() {
  // 假设 `this.state.count` 从 0 开始。
  this.incrementCount();
  this.incrementCount();
  this.incrementCount();

  // 如果你现在在这里读取 `this.state.count`，它还是会为 0。
  // 但是，当 React 重新渲染该组件时，它会变为 3。
}
```

[学习更多有关 `setState` 的内容](#)

`setState` 什么时候是异步的？

目前，在事件处理函数内部的 `setState` 是异步的。

例如，如果 `Parent` 和 `Child` 在同一个 `click` 事件中都调用了 `setState`，这样就可以确保 `Child` 不会被重新渲染两次。取而代之的是，React 会将该 `state` “冲洗”到浏览器事件结束的时候，再统一地进行更新。这种机制可以在大型应用中得到很好的性能提升。

这只是一个实现的细节，所以请不要直接依赖于这种机制。在以后的版本当中，React 会在更多的情况下静默地使用 `state` 的批更新机制。

为什么 React 不同步地更新 `this.state`？

如前面章节解释的那样，在开始重新渲染之前，React 会有意地进行“等待”，直到所有在组件的事件处理函数内调用的 `setState()` 完成之后。这样可以通过避免不必要的重新渲染来提升性能。

但是，你可能还是会想，为什么 React 不能立即更新 `this.state`，而不对组件进行重新渲染呢。

主要有两个原因：

- 这样会破坏掉 `props` 和 `state` 之间的一致性，造成一些难以 debug 的问题。
- 这样会让一些我们正在实现的新功能变得无法实现。

这个 [GitHub 评论](#) 深入了该特殊示例。

我应该使用一个像 Redux 或 MobX 那样的 `state` 管理库吗？

[或许需要。](#)

在添加额外的库之前，最好先了解清楚 React 能干什么。你也可以只使用 React 来构建出一个比较复杂的应用。

react

[Go to TOC](#)

项目文件结构

是否有一种推荐的方式来组织 React 的项目文件结构呢？

React 对如何将文件放入文件夹中没有意见。也就是说，你可以参考使用生态系统中一些常见的组织项目文件结构的方式。

按功能或路由组织

组织项目文件结构的一种常见方法是将 CSS、JS 和测试文件一起按照功能或路由进行组织。

```
common/
  Avatar.js
  Avatar.css
  APIUtils.js
  APIUtils.test.js
feed/
  index.js
  Feed.js
  Feed.css
  FeedStory.js
  FeedStory.test.js
  FeedAPI.js
profile/
  index.js
  Profile.js
  ProfileHeader.js
  ProfileHeader.css
  ProfileAPI.js
```

一个“功能”的定义因人而异，你可以自行选择其粒度。如果你想不出如何设计顶层目录，则可以通过向产品用户询问这个产品所包含的主要部分，并将反馈的想法用作设计蓝本。

按文件类型组织

另一种组织项目文件结构的常用方法是将类似的文件组织在一起，例如：

```
api/
  APIUtils.js
  APIUtils.test.js
  ProfileAPI.js
  UserAPI.js
components/
  Avatar.js
  Avatar.css
  Feed.js
  Feed.css
  FeedStory.js
  FeedStory.test.js
  Profile.js
  ProfileHeader.js
  ProfileHeader.css
```

一些人还喜欢更进一步，他们根据组件在应用程序中的角色将组件文件组织到不同的目录中去。例如，[原子设计](#)就是一种基于此原理的设计方法。请记住，将这些方法作为有用的示例而非必须严格遵守的规则，通常会更富成效。

避免多层嵌套

JavaScript 项目中的深层目录嵌套会带来许多痛点。在编写相对路径导入，或是在文件移动后更新这些导入将变得更加困难。除非你有非常令人信服的理由来使用深层目录嵌套，否则请考虑将单个项目中的目录嵌套控制在最多三到四个层级内。当然，这只是一个建议，它可能与你的项目无关。

不要过度思考

如果你刚刚开始一个项目，[不要花超过五分钟](#)在选择项目文件组织结构上。选择上述任何方式（或提出自己的方式）并开始编写代码！因为，在你编写了一些真正的代码之后，你将很有可能会重新考虑它。

如果您感觉完全卡住，请先将所有文件保存在同一个文件夹中。它最终会变得足够大，以至于让你想要将其中一些文件拆分出去。到那时，你将有足够的知识去区分你最频繁编辑的文件。通常，将经常一起变化的文件组织在一起是个好主意。这个原则被称为“colocation”。

随着项目规模的扩大，人们通常会在实践中混搭使用上述这些方式。因此，在开始时选择“正确”的那个方式并不是很重要。

样式与 CSS

如何为组件添加 CSS 的 class？

传递一个字符串作为 `className` 属性：

```
render() {
  return <span className="menu navigation-menu">Menu</span>
}
```

CSS 的 class 依赖组件的 props 或 state 的情况很常见：

```
render() {
  let className = 'menu';
  if (this.props.isActive) {
    className += ' menu-active';
  }
  return <span className={className}>Menu</span>
}
```

提示

如果你经常发现自己写类似这样的代码，`classnames` 的 package 可以简化这一过程。

可以使用行内样式吗？

可以，在[此处](#)查看关于样式的文档。

行内样式不好（bad）吗？

从性能角度来说，CSS 的 class 通常比行内样式更好。

什么是 CSS-in-JS？

“CSS-in-JS”是指一种模式，其中 CSS 由 JavaScript 生成而不是在外部文件中定义。

注意此功能并不是 React 的一部分，而是由第三方库提供。React 对样式如何定义并没有明确态度；如果存在疑惑，比较好的方式是和平时一样，在一个单独的 `*.css` 文件定义你的样式，并且通过 `className` 指定它们。

可以在 React 中实现动画效果吗？

React 可以被用来实现强大的动画效果。参见 [React Transition Group](#)、[React Motion](#) 以及 [React Spring](#) 等示例。

[Go to TOC](#)

版本号规则

React 遵循语义化版本 (semver) 原则。

也就是说，若当前版本号为 **x.y.z**，则：

- 当出现严重 bug 并修复时，我们会通过修改 **z** 来发布一个修订版本（如：15.6.2 至 15.6.3）。
- 当发布新功能或修复非严重 bug 时，我们会通过修改 **y** 来发布一个次要版本（如：15.6.2 至 15.7.0）。
- 当发布破坏性更新时，我们会通过修改 **x** 来发布一个主版本（如：15.6.2 至 16.0.0）。

主版本也可能包含新功能，任何一个版本都可能包含问题修复。

次要版本是最常见的版本发布类型。

此版本控制策略不适用于 Next 或 Experimental channels 中的预览版。[了解更多关于预览版信息](#)。

不兼容的改动

不兼容的改动会对所有人造成不便，因此我们会尽可能地减少主版本的发布次数。例如：React 15 在 2016 年 4 月发布，React 16 在 2017 年 9 月发布，React 17 在 2020 年 10 月发布。

实际上，我们会在次版本中发布新功能。次版本相比主版本更加有趣和吸引人，尽管它的名字没这么说。

对稳定性的承诺

在我们不断改变 React 的过程中，我们一直尝试着减少使用新功能的成本。如果情况允许，我们会保留旧的 API，哪怕是将它放在一个独立的 package 中。例如，[很多年前我们就不建议使用 mixins](#)，但我们仍然通过 `create-react-class` 提供对它的支持，使许多项目得以继续在稳定的、旧的代码中使用 mixins。

超过一百万开发者使用 React，维护着数百万个组件。仅 Facebook 代码库就有超过 50000 个 React 组件。这意味着我们需要让升级 React 版本尽可能的简单。如果我们做了很大的变动但没有提供升级方案，人们就会继续使用旧版本。我们在 Facebook 内部会测试这些升级 —— 如果不足 10 人的团队能够独自升级 50000 多个组件，那么任何使用 React 的人都可以轻松升级。在许多情况下，我们会编写[自动脚本](#)来帮助更新组件语法，并将这些脚本开源供所有人使用。

根据警告逐渐升级

开发环境的 React 包含许多有帮助的警告。我们会尽可能地添加一些警告以提示未来的不兼容改动。因此，如果你的应用使用的是最新版本而没有收到警告，那么它就会兼容下个主版本。基于此，你可以逐个组件地升级你的应用。

开发环境的警告不会影响你的应用，你的应用在开发环境和生产环境的表现是一致的 —— 唯一的区别是生产环境不会打印出这些警告，并且生产环境的性能更好。（如果你发现了其他的区别，请提交 issue。）

什么算是不兼容的改动？

通常来说，在对下列事情做变更时，我们**不会**升级主版本号：

- **开发环境的警告。** 由于这些警告不会影响生产环境的表现，我们可能会在同一个主版本中添加或修改警告。实际上这样做可以让我们更好地去提示即将到来的不兼容的改动。
- **以 `unstable_` 开头的 API。** 当一些功能还不够稳定时，这些 API 会作为试验性功能提供。通过以 `unstable_` 为前缀的方式发布这些 API，我们能够更快地迭代，更早地推出稳定的功能。
- **React 的 alpha 和 canary 版本。** 我们提供 alpha 版本的 React 是为了尽早测试新功能，同时我们需要根据 alpha 版本的反馈灵活地做出改动。如果你使用了这些版本，请注意这些 API 在稳定版本发布前可能会有变化。
- **没有文档的 API 和内部数据结构。** 如果你使用了如`__SECRET_INTERNALS_DO_NOT_USE_OR_YOU_WILL_BE_FIRED`、`__reactInternalInstance$uk43rzhitjg`之类的内部属性，我们不会向你承诺这些属性会一直有效。

这条规则的出发点是务实的：我们不想为你造成不便。如果我们因为这些改动而升级了主版本号，那么会出现更多的主版本，为社区制造更多的版本问题，我们也不能快速地去完善 React。

然而，如果上述列表中的改动会在社区中造成大范围的问题，我们仍会尽全力提供一个逐渐升级的方案。

如果发布了次要版本但没有引入新特性，那为何不发布修订版？

发布次要版本可能并不会引入新特性。[此规则在 semver 有提及](#)，如果在私有代码中引入了实质性的新特性或优化，则[次要版本]会增加。它可能包含修订版本的更改。

但是，这带来一个问题，即为什么这些发行版本没有发布为修订版。

对于 React（或其他软件）来说，任何的更改都具有意外破坏的风险。想象一下，发布了修复 bug 的版本，但偶然又引入了另一个 bug。这不仅会对开发人员带来麻烦，还会影响他们对修订版本的信心。如果发布的修订版本针对的是实际开发中极少出现的 bug，则更加令人遗憾。

我们在保证 React 发布的版本没有 bug 方面有着良好的记录，但是修订版发布具有更高标准，因为大多数开发者认为使用这些版本不会带来不利的问题。

基于以上原因，我们仅针对最严重的 bug 和安全漏洞发布修订版本。

如果某个版本包含不必要的更改（例如内部重构，对实现细节的修改，性能优化或较小 bug 的修正），即使没有新特性，我们也会更改次要版本。

表单

在 React 里，HTML 表单元素的工作方式和其他的 DOM 元素有些不同，这是因为表单元素通常会保持一些内部的 state。例如这个纯 HTML 表单只接受一个名称：

```
<form>
  <label>
    名字:
    <input type="text" name="name" />
  </label>
  <input type="submit" value="提交" />
</form>
```

此表单具有默认的 HTML 表单行为，即在用户提交表单后浏览到新页面。如果你在 React 中执行相同的代码，它依然有效。但大多数情况下，使用 JavaScript 函数可以很方便的处理表单的提交，同时还可以访问用户填写的表单数据。实现这种效果的标准方式是使用“受控组件”。

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()` 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”。

例如，如果我们想让前一个示例在提交时打印出名称，我们可以将表单写为受控组件：

```
class NameForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: ''};

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({value: event.target.value});
  }

  handleSubmit(event) {
    alert('提交的名字: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          名字:
          <input type="text" value={this.state.value} onChange={this.handleChange}>
        </label>
      </form>
    );
  }
}
```

```

        <input type="submit" value="提交" />
    </form>
);
}
}

```

[在 CodePen 上尝试](#)

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。由于 `handlechange` 在每次按键时都会执行并更新 React 的 state，因此显示的值将随着用户输入而更新。

对于受控组件来说，输入的值始终由 React 的 state 驱动。你也可以将 `value` 传递给其他 UI 元素，或者通过其他事件处理函数重置，但这意味着你需要编写更多的代码。

textarea 标签

在 HTML 中，`<textarea>` 元素通过其子元素定义其文本：

```

<textarea>
你好，这是在 text area 里的文本
</textarea>

```

而在 React 中，`<textarea>` 使用 `value` 属性代替。这样，可以使得使用 `<textarea>` 的表单和使用单行 `input` 的表单非常类似：

```

class EssayForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: '请撰写一篇关于你喜欢的 DOM 元素的文章。'
    };

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({value: event.target.value});
  }

  handleSubmit(event) {
    alert('提交的文章：' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          文章：
          <textarea value={this.state.value} onChange={this.handleChange}></textarea>
        </label>
        <input type="submit" value="提交" />
      </form>
    );
  }
}

```

请注意，`this.state.value` 初始化于构造函数中，因此文本区域默认有初值。

select 标签

在 HTML 中，`<select>` 创建下拉列表标签。例如，如下 HTML 创建了水果相关的下拉列表：

```
<select>
  <option value="grapefruit">葡萄柚</option>
  <option value="lime">酸橙</option>
  <option selected value="coconut">椰子</option>
  <option value="mango">芒果</option>
</select>
```

请注意，由于 `selected` 属性的缘故，椰子选项默认被选中。React 并不会使用 `selected` 属性，而是在根 `select` 标签上使用 `value` 属性。这在受控组件中更便捷，因为您只需要在根标签中更新它。例如：

```
class FlavorForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: 'coconut'};

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({value: event.target.value});
  }

  handleSubmit(event) {
    alert('你喜欢的风味是：' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          选择你喜欢的风味：
          <select value={this.state.value} onChange={this.handleChange}>
            <option value="grapefruit">葡萄柚</option>
            <option value="lime">酸橙</option>
            <option value="coconut">椰子</option>
            <option value="mango">芒果</option>
          </select>
        </label>
        <input type="submit" value="提交" />
      </form>
    );
  }
}
```

[在 CodePen 上尝试](#)

总的来说，这使得 `<input type="text">`，`<textarea>` 和 `<select>` 之类的标签都非常相似--它们都接受一个 `value` 属性，你可以使用它来实现受控组件。

注意

你可以将数组传递到 `value` 属性中，以支持在 `select` 标签中选择多个选项：

```
<select multiple={true} value={['B', 'C']}>
```

文件 input 标签

在 HTML 中，`<input type="file">` 允许用户从存储设备中选择一个或多个文件，将其上传到服务器，或通过使用 JavaScript 的 [File API](#) 进行控制。

```
<input type="file" />
```

因为它的 `value` 只读，所以它是 React 中的一个[非受控组件](#)。将与其他非受控组件[在后续文档中](#)一起讨论。

处理多个输入

当需要处理多个 `input` 元素时，我们可以给每个元素添加 `name` 属性，并让处理函数根据 `event.target.name` 的值选择要执行的操作。

例如：

```
class Reservation extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      isGoing: true,
      numberOfGuests: 2
    };
    this.handleInputChange = this.handleInputChange.bind(this);
  }

  handleInputChange(event) {
    const target = event.target;
    const value = target.type === 'checkbox' ? target.checked : target.value;
    const name = target.name;

    this.setState({
      [name]: value
    });
  }

  render() {
    return (
      <form>
        <label>
          参与：
          <input
            name="isGoing"
            type="checkbox"
            checked={this.state.isGoing}
            onChange={this.handleInputChange} />
        </label>
    );
  }
}
```

```

<br />
<label>
  来宾人数：
  <input
    name="numberOfGuests"
    type="number"
    value={this.state.numberOfGuests}
    onChange={this.handleInputChange} />
</label>
</form>
);
}
}

```

[在 CodePen 上尝试](#)

这里使用了 ES6 [计算属性名称](#) 的语法更新给定输入名称对应的 state 值：

例如：

```

this.setState({
  [name]: value
});

```

等同 ES5：

```

var partialState = {};
partialState[name] = value;
this.setState(partialState);

```

另外，由于 `setState()` 自动将部分 state 合并到当前 state，只需调用它更改部分 state 即可。

受控输入空值

在[受控组件](#)上指定 `value` 的 prop 会阻止用户更改输入。如果你指定了 `value`，但输入仍可编辑，则可能是你意外地将 `value` 设置为 `undefined` 或 `null`。

下面的代码演示了这一点。（输入最初被锁定，但在短时间延迟后变为可编辑。）

```

ReactDOM.createRoot(mountNode).render(<input value="hi" />);

setTimeout(function() {
  ReactDOM.createRoot(mountNode).render(<input value={null} />);
}, 1000);

```

受控组件的替代品

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](#)，这是实现输入表单的另一种方式。

成熟的解决方案

如果你想寻找包含验证、追踪访问字段以及处理表单提交的完整解决方案，使用 [Formik](#) 是不错的选择。然而，它也是建立在受控组件和管理 state 的基础之上 —— 所以不要忽视学习它们。

Refs 转发

Ref 转发是一项将 `ref` 自动地通过组件传递到其子组件的技巧。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。最常见的案例如下所述。

转发 refs 到 DOM 组件

考虑这个渲染原生 DOM 元素 `button` 的 `FancyButton` 组件：[embed:forwarding-refs/fancy-button-simple.js](#)

React 组件隐藏其实现细节，包括其渲染结果。其他使用 `FancyButton` 的组件通常不需要获取内部的 DOM 元素 `button` 的 `ref`。这很好，因为这防止组件过度依赖其他组件的 DOM 结构。

虽然这种封装对类似 `FeedStory` 或 `Comment` 这样的应用级组件是理想的，但其对 `FancyButton` 或 `MyTextInput` 这样的高可复用“叶”组件来说可能是不方便的。这些组件倾向于在整个应用中以一种类似常规 DOM `button` 和 `input` 的方式被使用，并且访问其 DOM 节点对管理焦点、选中或动画来说是不可避免的。

Ref 转发是一个可选特性，其允许某些组件接收 `ref`，并将其向下传递（换句话说，“转发”它）给子组件。

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`：

[embed:forwarding-refs/fancy-button-simple-ref.js](#)

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。

以下是对上述示例发生情况的逐步解释：

1. 我们通过调用 `React.createRef` 创建了一个 React `ref` 并将其赋值给 `ref` 变量。
2. 我们通过指定 `ref` 为 JSX 属性，将其向下传递给 `<FancyButton ref={ref}>`。
3. React 传递 `ref` 给 `forwardRef` 内函数 `(props, ref) => ...`，作为其第二个参数。
4. 我们向下转发该 `ref` 参数到 `<button ref={ref}>`，将其指定为 JSX 属性。
5. 当 `ref` 挂载完成，`ref.current` 将指向 `<button>` DOM 节点。

注意

第二个参数 `ref` 只在使用 `React.forwardRef` 定义组件时存在。常规函数和 class 组件不接收 `ref` 参数，且 `props` 中也不存在 `ref`。

Ref 转发不仅限于 DOM 组件，你也可以转发 refs 到 class 组件实例中。

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。

在高阶组件中转发 refs

这个技巧对高阶组件（也被称为 HOC）特别有用。让我们从一个输出组件 `props` 到控制台的 HOC 示例开始：

`embed:forwarding-refs/log-props-before.js`

“`logProps`” HOC 透传（pass through）所有 `props` 到其包裹的组件，所以渲染结果将是相同的。例如：我们可以使用该 HOC 记录所有传递到“`fancy button`”组件的 `props`：`embed:forwarding-refs/fancy-button.js`

下面的示例有一点需要注意：`refs` 将不会透传下去。这是因为 `ref` 不是 prop 属性。就像 `key` 一样，其被 React 进行了特殊处理。如果你对 HOC 添加 `ref`，该 `ref` 将引用最外层的容器组件，而不是被包裹的组件。

这意味着用于我们 `FancyButton` 组件的 `refs` 实际上将被挂载到 `LogProps` 组件：`embed:forwarding-refs/fancy-button-ref.js`

幸运的是，我们可以使用 `React.forwardRef` API 明确地将 `refs` 转发到内部的 `FancyButton` 组件。

`React.forwardRef` 接受一个渲染函数，其接收 `props` 和 `ref` 参数并返回一个 React 节点。例如：

`embed:forwarding-refs/log-props-after.js`

在 DevTools 中显示自定义名称

`React.forwardRef` 接受一个渲染函数。React DevTools 使用该函数来决定为 `ref` 转发组件显示的内容。

例如，以下组件将在 DevTools 中显示为“`ForwardRef`”：

`embed:forwarding-refs/wrapped-component.js`

如果你命名了渲染函数，DevTools 也将包含其名称（例如“`ForwardRef(myFunction)`”）：

`embed:forwarding-refs/wrapped-component-with-function-name.js`

你甚至可以设置函数的 `displayName` 属性来包含被包裹组件的名称：

`embed:forwarding-refs/customized-display-name.js`

Fragments

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。

```
render() {
  return (
    <React.Fragment>
      <ChildA />
      <ChildB />
      <ChildC />
    </React.Fragment>
  );
}
```

还有一种新的[短语法](#)可用于声明它们。

动机

一种常见模式是组件返回一个子元素列表。以此 React 代码片段为例：

```
class Table extends React.Component {
  render() {
    return (
      <table>
        <tr>
          <Columns />
        </tr>
      </table>
    );
  }
}
```

`<Columns />` 需要返回多个 `<td>` 元素以使渲染的 HTML 有效。如果在 `<Columns />` 的 `render()` 中使用了父 `div`，则生成的 HTML 将无效。

```
class Columns extends React.Component {
  render() {
    return (
      <div>
        <td>Hello</td>
        <td>World</td>
      </div>
    );
  }
}
```

得到一个 `<Table />` 输出：

```
<table>
  <tr>
    <div>
      <td>Hello</td>
      <td>World</td>
```

```
</div>
</tr>
</table>
```

Fragments 解决了这个问题。

用法

```
class Columns extends React.Component {
  render() {
    return (
      <React.Fragment>
        <td>Hello</td>
        <td>World</td>
      </React.Fragment>
    );
  }
}
```

这样可以正确的输出 `<Table />` :

```
<table>
  <tr>
    <td>Hello</td>
    <td>World</td>
  </tr>
</table>
```

短语法

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：

```
class Columns extends React.Component {
  render() {
    return (
      <>
        <td>Hello</td>
        <td>World</td>
      </>
    );
  }
}
```

你可以像使用其他任意元素一样使用 `<> </>`，但它并不支持 key 或属性。

带 key 的 Fragments

使用显式 `<React.Fragment>` 语法声明的片段可能具有 key。一个使用场景是将一个集合映射到一个 Fragments 数组 - 举个例子，创建一个描述列表：

```
function Glossary(props) {
  return (
    <dl>
      {props.items.map(item => (
        // 没有`key`，React 会发出一个关键警告
        <React.Fragment key={item.id}>
```

```
<dt>{item.term}</dt>
  <dd>{item.description}</dd>
</React.Fragment>
) );
</dl>
);
}
```

`key` 是唯一可以传递给 `Fragment` 的属性。未来我们可能会添加对其他属性的支持，例如事件。

在线 Demo

你可以在 [CodePen](#) 中尝试这个新的 JSX Fragment 语法。

[Go to TOC](#)

开始

- "download.html"
- "downloads.html"

这是一个 React 文档及相关资源的概览页面。

React 是一个用于构建用户界面的 JavaScript 库。你可以在[首页](#)或[教程](#)中学习什么是 React。

- [体验 React](#)
- [学习 React](#)
- [获取最新消息](#)
- [版本化文档](#)
- [找不到想要的？](#)

体验 React

React 从诞生之初就是可被逐步采用的，因而你可以按需引入或多或少的 **React 特性**。不管你是想体验下 React，用它给简单的 HTML 页面增加一点交互，还是要开始一个完全由 React 驱动的复杂应用，该章节内容里的链接都能帮你快速开始。

在线体验

如果你对体验 React 感兴趣，可以尝试在线代码编辑器。从 [CodePen](#), [CodeSandbox](#), 或者 [Stackblitz](#) 开始一个 React 版本的 Hello World 模板。

如果你喜欢使用自己的文本编辑器，也可以[下载这个 HTML 文件](#)，然后编辑文件内容，最后再用浏览器从本地文件系统打开文件，预览页面效果。注意：这个文件中包含一个低效率的运行时代码转换脚本，所以我们推荐仅在简单的演示项目中使用。

在网站中添加 React

你可以[立即在 HTML 文件中添加 React](#)，然后选择逐渐拓展它的应用范围，或只在一些动态小部件中使用它。

创建新的 React 应用

当你刚开始一个 React 应用时，[通过 HTML 的 script 标签引入 React](#) 依然是最好的选项，因为这能让你的项目立即启动。

但随着应用越来越大，你可能会需要更加集成化的安装方式。我们推荐了一些 JavaScript 工具链，它们适合大型应用。它们只需很少甚至零配置，就能让你充分利用丰富的 React 生态。[立即尝试](#)。

学习 React

学习 React 的人有着不同的知识背景和学习方式，不管你是理论派还是实践派，我们希望该章节内容对你有帮助。

- 如果你喜欢边做边学，请从[实践教程](#)开始。
- 如果你喜欢一步步学习概念，请从 [Hello World](#) 开始。

和所有新技术一样，React 也有一条学习曲线。但只要多多实践和保有一点点耐心，你终将掌握它。

第一个示例

[React 首页](#)包含了一些可在线编辑的小示例。即使你对 React 尚一无所知，也可以尝试修改代码，看看结果会怎样。

初学者的 React

如果你觉得 React 官方文档节奏太快，不太适应，可以先去看看这篇 [Tania Rascia 的 React 概览](#)。它以新手友好的方式详细介绍了最重要的 React 概念，看完这篇概览，再回来试试看官方文档吧！

设计师的 React

如果你有设计背景，[这些资源](#)是很好的开始。

JavaScript 资源

阅读 React 官方文档需要对使用 JavaScript 语言编程有一定基础。你不需要是一个 JavaScript 编程专家，但同时学习 React 和 JavaScript 编程将会比较困难。

我们推荐你浏览 [JavaScript 概览](#)来评估自己的知识水平。这将花费你大约 30 分钟到 1 个小时，但之后学习 React 你会感觉更加自信。

提示

当你对一些 JavaScript 知识感到疑惑时，[MDN](#) 和 [javascript.info](#) 是很棒的资源网站。你也可以去[社区支持论坛](#)寻求帮助。

实践教程

如果你喜欢边做边学，请参考我们的[实践教程](#)。在这篇教程中，我们用 React 构建了一个井字棋游戏。即使你对游戏开发并不感兴趣而想要跳过这部分，但还请给它一次机会。这篇教程中所涉及的技术是构建任何 React 应用的基础，掌握它们将让你对 React 有更深的理解。

一步步学习指南

如果你喜欢[一步步学习概念](#)，我们的[核心概念简介](#)是最好的开始。指南里的每一章节都基于前一章节已学的内容，这样一步步过来你不会错过任何知识。

React 哲学

很多用户认为阅读 [React 哲学](#)是他们最终对 React 恍然大悟的时刻。它可能是最老的 React 教程，但它依然极具价值。

推荐课程

相比官方文档，有时人们更喜欢第三方的书籍或视频课程。我们维护了一个[推荐资源清单](#)，其中有一些资源是免费的。

高级概念

一旦你已经熟知 React 的[主要概念](#)，并且已经在项目中小试牛刀，你可能会对更高级的话题感兴趣。该章节内容将会介绍一些强大但并不常用的 React 特性，比如：[context](#) 和 [refs](#)。

API 参考

当你想深入学习特定 React API 细节时，该章节将会变得非常有用。举个例子：[React.Component API 参考](#)能够介绍 [setState\(\)](#) 方法是如何工作的，以及不同生命周期方法的使用场景。

术语表与常见问题

[术语表](#)中包含了 React 文档中最常见术语的概览。常见问题章节致力于简短解答一些大家普遍遇到的问题，包括[发送 AJAX 请求](#)，[组件状态](#)以及[文件结构](#)等。

获取最新消息

[React 博客](#)是 React 团队发布更新的官方渠道。一切重要的信息，包括：更新日志，废弃通知等，都会首先在这里发布。

你也可以在 Twitter 上关注 [@reactjs 账号](#) 获取更新，但即使不这样做，仅通过官方博客你也不会错过任何必要的信息。

并非每一个 React 版本都值得我们在博客上发布文章，但你可以在 [React 代码仓库中的 CHANGELOG.md 文件](#)或[更新日志页面](#)找到每个版本的更新日志。

版本化文档

官方文档会和最新稳定版的 React 保持同步。从 React 16 开始，你可以在[历史版本页面](#)找到老版本的 React 文档。注意，老版本的文档是该版本发布时的快照，并不会持续地更新。

找不到想要的？

如果你找不到想要的内容或觉得文档某些地方让人疑惑，请[移步文档仓库提交 issue](#) 或在 Twitter 上提及 [@reactjs 账号](#)。我们期待你的反馈！

事件处理

React 元素的事件处理和 DOM 元素的很相似，但是有一点语法上的不同：

- 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。

例如，传统的 HTML：

```
<button onclick="activateLasers()">
  Activate Lasers
</button>
```

在 React 中略微不同：

```
<button onClick={activateLasers}>
  Activate Lasers
</button>
```

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。你必须显式地使用 `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：

```
<form onsubmit="console.log('You clicked submit.'); return false">
  <button type="submit">Submit</button>
</form>
```

在 React 中，可能是这样的：

```
function Form() {
  function handleSubmit(e) {
    e.preventDefault();
    console.log('You clicked submit.');
  }

  return (
    <form onSubmit={handleSubmit}>
      <button type="submit">Submit</button>
    </form>
  );
}
```

在这里，`e` 是一个合成事件。React 根据 [W3C 规范](#) 来定义这些合成事件，所以你不需要担心跨浏览器的兼容性问题。React 事件与原生事件不完全相同。如果想了解更多，请查看 [SyntheticEvent](#) 参考指南。

使用 React 时，你一般不需要使用 `addEventListener` 为已创建的 DOM 元素添加监听器。事实上，你只需要在该元素初始渲染的时候添加监听器即可。

当你使用 [ES6 class](#) 语法定义一个组件的时候，通常的做法是将事件处理函数声明为 class 中的方法。例如，下面的 `Toggle` 组件会渲染一个让用户切换开关状态的按钮：

```

class Toggle extends React.Component {
  constructor(props) {
    super(props);
    this.state = {isToggleOn: true};

    // 为了在回调中使用 `this`，这个绑定是必不可少的
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState(prevState => ({
      isToggleOn: !prevState.isToggleOn
    }));
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        {this.state.isToggleOn ? 'ON' : 'OFF'}
      </button>
    );
  }
}

```

在 CodePen 上尝试

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 `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](#)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。你可以使用 [public class fields 语法](#) to correctly bind callbacks:

```

class LoggingButton extends React.Component {
  // This syntax ensures `this` is bound within handleClick.
  handleClick = () => {
    console.log('this is:', this);
  };

  render() {
    return (
      <button onClick={this.handleClick}>
        Click me
      </button>
    );
  }
}

```

[Create React App](#) 默认启用此语法。

如果你没有使用 class fields 语法，你可以在回调中使用[箭头函数](#)：

```

class LoggingButton extends React.Component {
  handleClick() {
    console.log('this is:', this);
  }
}

```

```

render() {
  // 此语法确保 `handleClick` 内的 `this` 已被绑定。
  return (
    <button onClick={() => this.handleClick()}>
      Click me
    </button>
  );
}
}

```

此语法问题在于每次渲染 `LoggingButton` 时都会创建不同的回调函数。在大多数情况下，这没什么问题，但如果该回调函数作为 prop 传入子组件时，这些组件可能会进行额外的重新渲染。我们通常建议在构造器中绑定或使用 class fields 语法来避免这类性能问题。

向事件处理程序传递参数

在循环中，通常我们会为事件处理函数传递额外的参数。例如，若 `id` 是你要删除那一行的 ID，以下两种方式都可以向事件处理函数传递参数：

```

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

```

上述两种方式是等价的，分别通过 `箭头函数` 和 `Function.prototype.bind` 来实现。

在这两种情况下，React 的事件对象 `e` 会被作为第二个参数传递。如果通过箭头函数的方式，事件对象必须显式的进行传递，而通过 `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。

Hello World

最简易的 React 示例如下：

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

它将在页面上展示一个 "Hello, world!" 的标题。

[在 CodePen 上试试](#)

点击链接打开在线编辑器。随意更改内容，查看它们会怎样影响展示。本指南中的大多数页面都有像这样的可编辑的示例。

如何阅读本指南

在本指南中，我们将研究 React 应用程序的组成部分：元素和组件。一旦你掌握了它们，便可以使用这些可复用的小片段组成复杂的应用。

提示

本指南专为喜欢**逐步学习概念**的人士设计。如果你想边学边做，请查阅我们的[实用教程](#)。你会发现该指南与教程是相互补充的。

本文是 React 核心概念分步指南的第一章。你可以在侧边导航栏中找到所有章节的列表。如果你是通过移动设备阅读此内容，你可以通过点击屏幕右下角的按钮来查看导航栏。

本指南中的每一章节都以其前述章节中介绍的知识点为基础。**你可以按照侧边导航栏中显示的顺序阅读浏览“核心概念”的指南章节。以了解 React 的大部分内容。**例如，“[JSX 简介](#)”就是本文的下一章节。

预备知识

React 是一个 JavaScript 库，所以我们假设你对 JavaScript 语言已有基本的了解。**如果你对自己的基础不自信，我们推荐通过 [JavaScript 教程](#)检查你的基础知识储备水平**，使得你能够无压力的阅读本指南。这可能会花费你 30 分钟到 1 个小时的时间，但这样做的好处是你不会觉得同时在学习 React 和 JavaScript。

注意

本指南的示例中偶尔会使用一些 JavaScript 新语法。如果你在过去几年中并没有使用过 JavaScript，大多数情况下[这三点](#)应该能帮到你。

让我们开始吧！

继续向下滚动，你将在网站页脚右侧找到[下一篇指南](#)的链接。

[Go to TOC](#)

高阶组件

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。

HOC 在 React 的第三方库中很常见，例如 Redux 的 `connect` 和 Relay 的 `createFragmentContainer`。

在本文档中，我们将讨论为什么高阶组件有用，以及如何编写自己的 HOC 函数。

使用 HOC 解决横切关注点问题

注意

我们之前建议使用 mixins 用于解决横切关注点相关的问题。但我们已经意识到 mixins 会产生更多麻烦。[阅读更多](#) 以了解我们为什么要抛弃 mixins 以及如何转换现有组件。

组件是 React 中代码复用的基本单元。但你会发现某些模式并不适合传统组件。

例如，假设有一个 `CommentList` 组件，它订阅外部数据源，用以渲染评论列表：

```
class CommentList extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {
      // 假设 "DataSource" 是个全局范围内的数据源变量
      comments: DataSource.getComments()
    };
  }

  componentDidMount() {
    // 订阅更改
    DataSource.addChangeListener(this.handleChange);
  }

  componentWillUnmount() {
    // 清除订阅
    DataSource.removeChangeListener(this.handleChange);
  }

  handleChange() {
    // 当数据源更新时，更新组件状态
    this.setState({
      comments: DataSource.getComments()
    });
  }
}
```

```

    }

    render() {
      return (
        <div>
          {this.state.comments.map((comment) => (
            <Comment comment={comment} key={comment.id} />
          ))}
        </div>
      );
    }
}

```

稍后，编写了一个用于订阅单个博客帖子的组件，该帖子遵循类似的模式：

```

class BlogPost extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {
      blogPost: DataSource.getBlogPost(props.id)
    };
  }

  componentDidMount() {
    DataSource.addChangeListener(this.handleChange);
  }

  componentWillUnmount() {
    DataSource.removeChangeListener(this.handleChange);
  }

  handleChange() {
    this.setState({
      blogPost: DataSource.getBlogPost(this.props.id)
    });
  }

  render() {
    return <TextBlock text={this.state.blogPost} />;
  }
}

```

`CommentList` 和 `BlogPost` 不同 - 它们在 `DataSource` 上调用不同的方法，且渲染不同的结果。但它们的大部分实现都是一样的：

- 在挂载时，向 `DataSource` 添加一个更改侦听器。
- 在侦听器内部，当数据源发生变化时，调用 `setState`。
- 在卸载时，删除侦听器。

你可以想象，在一个大型应用程序中，这种订阅 `DataSource` 和调用 `setState` 的模式将一次又一次地发生。我们需要一个抽象，允许我们在一个地方定义这个逻辑，并在许多组件之间共享它。这正是高阶组件擅长的地方。

对于订阅了 `DataSource` 的组件，比如 `CommentList` 和 `BlogPost`，我们可以编写一个创建组件函数。该函数将接受一个子组件作为它的其中一个参数，该子组件将订阅数据作为 prop。让我们调用函数 `withSubscription`：

```

const CommentListWithSubscription = withSubscription(
  CommentList,
  (DataSource) => DataSource.getComments()
);

const BlogPostWithSubscription = withSubscription(
  BlogPost,
  (DataSource, props) => DataSource.getBlogPost(props.id)
);

```

第一个参数是被包装组件。第二个参数通过 `DataSource` 和当前的 `props` 返回我们需要的数据。

当渲染 `CommentListWithSubscription` 和 `BlogPostWithSubscription` 时，`CommentList` 和 `BlogPost` 将传递一个 `data prop`，其中包含从 `DataSource` 检索到的最新数据：

```

// 此函数接收一个组件...
function withSubscription(WrappedComponent, selectData) {
  // ...并返回另一个组件...
  return class extends React.Component {
    constructor(props) {
      super(props);
      this.handleChange = this.handleChange.bind(this);
      this.state = {
        data: selectData(DataSource, props)
      };
    }

    componentDidMount() {
      // ...负责订阅相关操作...
      DataSource.addChangeListener(this.handleChange);
    }

    componentWillUnmount() {
      DataSource.removeChangeListener(this.handleChange);
    }

    handleChange() {
      this.setState({
        data: selectData(DataSource, this.props)
      });
    }

    render() {
      // ... 并使用新数据渲染被包装的组件！
      // 请注意，我们可能还会传递其他属性
      return <WrappedComponent data={this.state.data} {...this.props} />;
    }
  };
}

```

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。

被包装组件接收来自容器组件的所有 `prop`，同时也接收一个新的用于 `render` 的 `data prop`。HOC 不需要关心数据的使用方式或原因，而被包装组件也不需要关心数据是怎么来的。

因为 `withSubscription` 是一个普通函数，你可以根据需要对参数进行增添或者删除。例如，您可能希望使 `data prop` 的名称可配置，以进一步将 HOC 与包装组件隔离开来。或者你可以接受一个配置 `shouldComponentUpdate` 的参数，或者一个配置数据源的参数。因为 HOC 可以控制组件的定义方式，这一切都变得有可能。

与普通组件一样，`withSubscription` 和包装组件之间的契约完全基于之间传递的 `props`。这种依赖方式使得替换 HOC 变得容易，只要它们为包装的组件提供相同的 `prop` 即可。例如你需要改用其他库来获取数据的时候，这一点就很有用。

不要改变原始组件，而应该使用组合。

不要试图在 HOC 中修改组件原型（或以其他方式改变它）。

```
function logProps(InputComponent) {
  InputComponent.prototype.componentDidUpdate = function(prevProps) {
    console.log('Current props: ', this.props);
    console.log('Previous props: ', prevProps);
  };
  // 返回原始的 input 组件，暗示它已经被修改。
  return InputComponent;
}

// 每次调用 logProps 时，增强组件都会有 log 输出。
const EnhancedComponent = logProps(InputComponent);
```

这样做会产生一些不良后果。其一是输入组件再也无法像 HOC 增强之前那样使用了。更严重的是，如果你再用另一个同样会修改 `componentDidUpdate` 的 HOC 增强它，那么前面的 HOC 就会失效！同时，这个 HOC 也无法应用于没有生命周期的函数组件。

修改传入组件的 HOC 是一种糟糕的抽象方式。调用者必须知道他们是如何实现的，以避免与其他 HOC 发生冲突。

HOC 不应该修改传入组件，而应该使用组合的方式，通过将组件包装在容器组件中实现功能：

```
function logProps(WrappedComponent) {
  return class extends React.Component {
    componentDidUpdate(prevProps) {
      console.log('Current props: ', this.props);
      console.log('Previous props: ', prevProps);
    }
    render() {
      // 将 input 组件包装在容器中，而不对其进行修改。Good!
      return <WrappedComponent {...this.props} />;
    }
  }
}
```

该 HOC 与上文中修改传入组件的 HOC 功能相同，同时避免了出现冲突的情况。它同样适用于 `class` 组件和函数组件。而且因为它是一个纯函数，它可以与其他 HOC 组合，甚至可以与其自身组合。

您可能已经注意到 HOC 与 **容器组件模式** 之间有相似之处。容器组件担任将高级和低级关注点分离的责任，由容器管理订阅和状态，并将 `prop` 传递给处理 UI 的组件。HOC 使用容器作为其实现的一部分，你可以将 HOC 视为参数化容器组件。

约定：将不相关的 `props` 传递给被包裹的组件

HOC 为组件添加特性。自身不应该大幅改变约定。HOC 返回的组件与原组件应保持类似的接口。

HOC 应该透传与自身无关的 props。大多数 HOC 都应该包含一个类似于下面的 render 方法：

```
render() {
  // 过滤掉非此 HOC 额外的 props，且不要进行透传
  const { extraProp, ...passThroughProps } = this.props;

  // 将 props 注入到被包装的组件中。
  // 通常为 state 的值或者实例方法。
  const injectedProp = someStateOrInstanceMethod;

  // 将 props 传递给被包装组件
  return (
    <WrappedComponent
      injectedProp={injectedProp}
      {...passThroughProps}
    />
  );
}
```

这种约定保证了 HOC 的灵活性以及可复用性。

约定：最大化可组合性

并不是所有的 HOC 都一样。有时候它仅接受一个参数，也就是被包裹的组件：

```
const NavbarWithRouter = withRouter(Navbar);
```

HOC 通常可以接收多个参数。比如在 Relay 中，HOC 额外接收了一个配置对象用于指定组件的数据依赖：

```
const CommentWithRelay = Relay.createContainer(Comment, config);
```

最常见的 HOC 签名如下：

```
// React Redux 的 `connect` 函数
const ConnectedComment = connect(commentSelector, commentActions)(CommentList);
```

刚刚发生了什么？！如果你把它分开，就会更容易看出发生了什么。

```
// connect 是一个函数，它的返回值为另外一个函数。
const enhance = connect(commentListSelector, commentListActions);
// 返回值为 HOC，它会返回已经连接 Redux store 的组件
const ConnectedComment = enhance(CommentList);
```

换句话说，`connect` 是一个返回高阶组件的高阶函数！

这种形式可能看起来令人困惑或不必要，但它有一个有用的属性。像 `connect` 函数返回的单参数 HOC 具有签名 `Component => Component`。输出类型与输入类型相同的函数很容易组合在一起。

```
// 不推荐如下写法...
const EnhancedComponent = withRouter(connect(commentSelector)(WrappedComponent))

// ... 建议编写组合工具函数
// compose(f, g, h) 等同于 (...args) => f(g(h(...args)))
const enhance = compose(
  // 这些都是单参数的 HOC
```

```

  withRouter,
  connect(commentSelector)
)
const EnhancedComponent = enhance(WrappedComponent)

```

(同样的属性也允许 `connect` 和其他 HOC 承担装饰器的角色，装饰器是一个实验性的 JavaScript 提案。)

许多第三方库都提供了 `compose` 工具函数，包括 `lodash`（比如 `lodash.flowRight`），`Redux` 和 `Ramda`。

约定：包装显示名称以便轻松调试

HOC 创建的容器组件会与任何其他组件一样，会显示在 `React Developer Tools` 中。为了方便调试，请选择一个显示名称，以表明它是 HOC 的产物。

最常见的方式是用 HOC 包住被包装组件的显示名称。比如高阶组件名为 `withSubscription`，并且被包装组件的显示名称为 `CommentList`，显示名称应该为 `WithSubscription(CommentList)`：

```

function withSubscription(WrappedComponent) {
  class WithSubscription extends React.Component {/* ... */}
  WithSubscription.displayName =
`WithSubscription(${getDisplayName(WrappedComponent)})`;
  return WithSubscription;
}

function getDisplayName(WrappedComponent) {
  return WrappedComponent.displayName || WrappedComponent.name || 'Component';
}

```

注意事项

高阶组件有一些需要注意的地方，对于 React 新手来说可能不容易发现。

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。

通常，你不需要考虑这点。但对 HOC 来说这一点很重要，因为这代表着你不应在组件的 `render` 方法中对一个组件应用 HOC：

```

render() {
  // 每次调用 render 函数都会创建一个新的 EnhancedComponent
  // EnhancedComponent1 !== EnhancedComponent2
  const EnhancedComponent = enhance(MyComponent);
  // 这将导致子树每次渲染都会进行卸载，和重新挂载的操作！
  return <EnhancedComponent />;
}

```

这不仅仅是性能问题 - 重新挂载组件会导致该组件及其所有子组件的状态丢失。

如果在组件之外创建 HOC，这样一来组件只会创建一次。因此，每次 `render` 时都会是同一个组件。一般来说，这跟你的预期表现是一致的。

在极少数情况下，你需要动态调用 HOC。你可以在组件的生命周期方法或其构造函数中进行调用。

务必复制静态方法

有时在 React 组件上定义静态方法很有用。例如，Relay 容器暴露了一个静态方法 `getFragment` 以方便组合 GraphQL 片段。

但是，当你将 HOC 应用于组件时，原始组件将使用容器组件进行包装。这意味着新组件没有原始组件的任何静态方法。

```
// 定义静态函数
WrappedComponent.staticMethod = function() {/*...*/}
// 现在使用 HOC
const EnhancedComponent = enhance(WrappedComponent);

// 增强组件没有 staticMethod
typeof EnhancedComponent.staticMethod === 'undefined' // true
```

为了解决这个问题，你可以在返回之前把这些方法拷贝到容器组件上：

```
function enhance(WrappedComponent) {
  class Enhance extends React.Component {/*...*/}
  // 必须准确知道应该拷贝哪些方法 :(
  Enhance.staticMethod = WrappedComponent.staticMethod;
  return Enhance;
}
```

但要这样做，你需要知道哪些方法应该被拷贝。你可以使用 `hoist-non-react-statics` 自动拷贝所有非 React 静态方法：

```
import hoistNonReactStatic from 'hoist-non-react-statics';
function enhance(WrappedComponent) {
  class Enhance extends React.Component {/*...*/}
  hoistNonReactStatic(Enhance, WrappedComponent);
  return Enhance;
}
```

除了导出组件，另一个可行的方案是再额外导出这个静态方法。

```
// 使用这种方式代替...
MyComponent.someFunction = someFunction;
export default MyComponent;

// ...单独导出该方法...
export { someFunction };

// ...并在要使用的组件中，import 它们
import MyComponent, { someFunction } from './MyComponent.js';
```

Refs 不会被传递

虽然高阶组件的约定是将所有 `props` 传递给被包装组件，但这对于 `refs` 并不适用。那是因为 `ref` 实际上并不是一个 prop - 就像 `key` 一样，它是由 React 专内处理的。如果将 `ref` 添加到 HOC 的返回组件中，则 `ref` 引用指向容器组件，而不是被包装组件。

这个问题的解决方案是通过使用 `React.forwardRef` API (React 16.3 中引入)。前往 [ref 转发章节了解更多](#)。

react

[Go to TOC](#)

自定义 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 React 特性。

通过自定义 Hook，可以将组件逻辑提取到可重用的函数中。

在我们学习[使用 Effect Hook](#)时，我们已经见过这个聊天程序中的组件，该组件用于显示好友的在线状态：

```
import React, { useState, useEffect } from 'react';

function FriendStatus(props) {
  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}
```

现在我们假设聊天应用中有一个联系人列表，当用户在线时需要把名字设置为绿色。我们可以把上面类似的逻辑复制并粘贴到 `FriendListItem` 组件中来，但这并不是理想的解决方案：

```
import React, { useState, useEffect } from 'react';

function FriendListItem(props) {
  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  return (
    <li style={{ color: isOnline ? 'green' : 'black' }}>
      {props.friend.name}
    </li>
  );
}
```

相反，我们希望在 `FriendStatus` 和 `FriendListItem` 之间共享逻辑。

目前为止，在 React 中有两种流行的方式来共享组件之间的状态逻辑：`render props` 和高阶组件，现在让我们来看看 `Hook` 是如何在让你不增加组件的情况下解决相同问题的。

提取自定义 Hook

当我们想在两个函数之间共享逻辑时，我们会把它提取到第三个函数中。而组件和 `Hook` 都是函数，所以也同样适用这种方式。

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`：

```
import { useState, useEffect } from 'react';

function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    ChatAPI.subscribeToFriendStatus(friendID, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(friendID, handleStatusChange);
    };
  });

  return isOnline;
}
```

此处并未包含任何新的内容——逻辑是从上述组件拷贝来的。与组件中一致，请确保只在自定义 `Hook` 的顶层无条件地调用其他 `Hook`。

与 React 组件不同的是，自定义 `Hook` 不需要具有特殊的标识。我们可以自由的决定它的参数是什么，以及它应该返回什么（如果需要的话）。换句话说，它就像一个正常的函数。但是它的名字应该始终以 `use` 开头，这样可以一眼看出其符合 [Hook 的规则](#)。

此处 `useFriendStatus` 的 `Hook` 目的是订阅某个好友的在线状态。这就是我们需要将 `friendID` 作为参数，并返回这位好友的在线状态的原因。

```
function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);

  // ...

  return isOnline;
}
```

现在让我们看看应该如何使用自定义 `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：

```
function FriendStatus(props) {
  const isOnline = useFriendStatus(props.friend.id);

  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}

function FriendListItem(props) {
  const isOnline = useFriendStatus(props.friend.id);

  return (
    <li style={{ color: isOnline ? 'green' : 'black' }}>
      {props.friend.name}
    </li>
  );
}
```

这段代码等价于原来的示例代码吗？等价，它的工作方式完全一样。如果你仔细观察，你会发现我们没有对其行为做任何的改变，我们只是将两个函数之间一些共同的代码提取到单独的函数中。**自定义 Hook 是一种自然遵循 Hook 设计的约定，而并不是 React 的特性。**

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](#)。

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[之前章节中了解到的一样](#)，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。

提示：在多个 Hook 之间传递信息

由于 Hook 本身就是函数，因此我们可以在它们之间传递信息。

我们将使用聊天程序中的另一个组件来说明这一点。这是一个聊天消息接收者的选择器，它会显示当前选定的好友是否在线：

```
const friendList = [
];

function ChatRecipientPicker() {
  const [recipientID, setRecipientID] = useState(1);
  const isRecipientOnline = useFriendStatus(recipientID);

  return (
    <ul>
      {friendList.map(friend => (
        <li key={friend.id}>
          {friend.name} {isRecipientOnline ? 'Online' : 'Offline'}
        </li>
      ))}
    </ul>
  );
}
```

```

    <>
      <Circle color={isRecipientOnline ? 'green' : 'red'} />
      <select
        value={recipientID}
        onChange={e => setRecipientID(Number(e.target.value))}>
        >
          {friendList.map(friend => (
            <option key={friend.id} value={friend.id}>
              {friend.name}
            </option>
          ))}
        </select>
      </>
    );
  }
}

```

我们将当前选择的好友 ID 保存在 `recipientID` 状态变量中，并在用户从 `<select>` 中选择其他好友时更新这个 state。

由于 `useState` 为我们提供了 `recipientID` 状态变量的最新值，因此我们可以将它作为参数传递给自定义的 `useFriendStatus` Hook：

```

const [recipientID, setRecipientID] = useState(1);
const isRecipientOnline = useFriendStatus(recipientID);

```

如此可以让我们知道当前选中的好友是否在线。当我们选择不同的好友并更新 `recipientID` 状态变量时，`useFriendStatus` Hook 将会取消订阅之前选中的好友，并订阅新选中的好友状态。

useYourImagination() {#useyourimagination}

自定义 Hook 解决了以前在 React 组件中无法灵活共享逻辑的问题。你可以创建涵盖各种场景的自定义 Hook，如表单处理、动画、订阅声明、计时器，甚至可能还有其他我们没想到的场景。更重要的是，创建自定义 Hook 就像使用 React 内置的功能一样简单。

尽量避免过早地增加抽象逻辑。既然函数组件能够做的更多，那么代码库中函数组件的代码行数可能会剧增。这属于正常现象——不必立即将它们拆分为 Hook。但我们仍鼓励你能通过自定义 Hook 寻找可能，以达到简化代码逻辑，解决组件杂乱无章的目的。

例如，有个复杂的组件，其中包含了大量以特殊的方式来管理的内部状态。`useState` 并不会使得集中更新逻辑变得容易，因此你可能更愿意使用 `redux` 中的 `reducer` 来编写。

```

function todosReducer(state, action) {
  switch (action.type) {
    case 'add':
      return [...state, {
        text: action.text,
        completed: false
      }];
    // ... other actions ...
    default:
      return state;
  }
}

```

Reducers 非常便于单独测试，且易于扩展，以表达复杂的更新逻辑。如有必要，您可以将它们分成更小的 reducer。但是，你可能还享受着 React 内部 state 带来的好处，或者可能根本不想安装其他库。

那么，为什么我们不编写一个 `useReducer` 的 Hook，使用 reducer 的方式来管理组件的内部 state 呢？其简化版本可能如下所示：

```
function useReducer(reducer, initialState) {
  const [state, setState] = useState(initialState);

  function dispatch(action) {
    const nextState = reducer(state, action);
    setState(nextState);
  }

  return [state, dispatch];
}
```

在组件中使用它，让 reducer 驱动它管理 state：

```
function Todos() {
  const [todos, dispatch] = useReducer(todosReducer, []);

  function handleAddClick(text) {
    dispatch({ type: 'add', text });
  }

  // ...
}
```

在复杂组件中使用 reducer 管理内部 state 的需求很常见，我们已经将 `useReducer` 的 Hook 内置到 React 中。你可以在 [Hook API 索引](#)中找到它使用，搭配其他内置的 Hook 一起使用。

使用 Effect 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 React 特性。

Effect Hook 可以让你在函数组件中执行副作用操作

```
import React, { useState, useEffect } from 'react';

function Example() {
  const [count, setCount] = useState(0);

  // Similar to componentDidMount and componentDidUpdate:
  useEffect(() => {
    // Update the document title using the browser API
    document.title = `You clicked ${count} times`;
  });

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

这段代码基于[上一章节中的计数器示例](#)进行修改，我们为计数器增加了一个小功能：将 document 的 title 设置为包含了点击次数的消息。

数据获取，设置订阅以及手动更改 React 组件中的 DOM 都属于副作用。不管你知不知道这些操作，或是“副作用”这个名字，应该都在组件中使用过它们。

提示

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。

在 React 组件中有两种常见副作用操作：需要清除的和不需要清除的。我们来更仔细地看一下他们之间的区别。

无需清除的 effect

有时候，我们只想在 **React 更新 DOM 之后运行一些额外的代码**。比如发送网络请求，手动变更 DOM，记录日志，这些都是常见的无需清除的操作。因为我们在执行完这些操作之后，就可以忽略他们了。让我们对比一下使用 class 和 Hook 都是怎么实现这些副作用的。

使用 class 的示例

在 React 的 class 组件中，`render` 函数是不应该有任何副作用的。一般来说，在这里执行操作太早了，我们基本上都希望在 React 更新 DOM 之后才执行我们的操作。

这就是为什么在 React class 中，我们把副作用操作放到 `componentDidMount` 和 `componentDidUpdate` 函数中。回到示例中，这是一个 React 计数器的 class 组件。它在 React 对 DOM 进行操作之后，立即更新了 `document` 的 `title` 属性

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  componentDidMount() {
    document.title = `You clicked ${this.state.count} times`;
  }

  componentDidUpdate() {
    document.title = `You clicked ${this.state.count} times`;
  }

  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Click me
        </button>
      </div>
    );
  }
}
```

注意，在这个 `class` 中，我们需要在两个生命周期函数中编写重复的代码。

这是因为很多情况下，我们希望在组件加载和更新时执行同样的操作。从概念上说，我们希望它在每次渲染之后执行——但 React 的 class 组件没有提供这样的方法。即使我们提取出一个方法，我们还是要在两个地方调用它。

现在让我们来看看如何使用 `useEffect` 执行相同的操作。

使用 Hook 的示例

我们在本章节开始时已经看到了这个示例，但让我们再仔细观察它：

```
import React, { useState, useEffect } from 'react';

function Example() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });
}
```

```

return (
  <div>
    <p>You clicked {count} times</p>
    <button onClick={() => setCount(count + 1)}>
      Click me
    </button>
  </div>
);
}

```

useEffect 做了什么？ 通过使用这个 Hook，你可以告诉 React 组件需要在渲染后执行某些操作。React 会保存你传递的函数（我们将它称之为“effect”），并在执行 DOM 更新之后调用它。在这个 effect 中，我们设置了 document 的 title 属性，不过我们也可以执行数据获取或调用其他命令式的 API。

为什么在组件内部调用 useEffect？ 将 `useEffect` 放在组件内部让我们可以在 effect 中直接访问 `count` state 变量（或其他 props）。我们不需要特殊的 API 来读取它——它已经保存在函数作用域中。Hook 使用了 JavaScript 的闭包机制，而不用在 JavaScript 已经提供了解决方案的情况下，还引入特定的 React API。

useEffect 会在每次渲染后都执行吗？ 是的，默认情况下，它在第一次渲染之后和每次更新之后都会执行。（我们稍后会谈到如何控制它。）你可能会更容易接受 effect 发生在“渲染之后”这种概念，不用再去考虑“挂载”还是“更新”。React 保证了每次运行 effect 的同时，DOM 都已经更新完毕。

详细说明

现在我们已经对 effect 有了大致了解，下面这些代码应该不难看懂了：

```

function Example() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });
}

```

我们声明了 `count` state 变量，并告诉 React 我们需要使用 effect。紧接着传递函数给 `useEffect` Hook。此函数就是我们的 effect。然后使用 `document.title` 浏览器 API 设置 `document` 的 title。我们可以在 effect 中获取到最新的 `count` 值，因为他在函数的作用域内。当 React 渲染组件时，会保存已使用的 effect，并在更新完 DOM 后执行它。这个过程在每次渲染时都会发生，包括首次渲染。

经验丰富的 JavaScript 开发人员可能会注意到，传递给 `useEffect` 的函数在每次渲染中都会有所不同，这是刻意为之的。事实上这正是我们可以在 effect 中获取最新的 `count` 的值，而不用担心其过期的原因。每次我们重新渲染，都会生成新的 effect，替换掉之前的。某种意义上讲，effect 更像是渲染结果的一部分——每个 effect “属于”一次特定的渲染。我们将在[本章节后续部分](#)更清楚地了解这样做的意义。

提示

与 `componentDidMount` 或 `componentDidUpdate` 不同，使用 `useEffect` 调度的 effect 不会阻塞浏览器更新屏幕，这让你的应用看起来响应更快。大多数情况下，effect 不需要同步地执行。在个别情况下（例如测量布局），有单独的 `useLayoutEffect` Hook 供你使用，其 API 与 `useEffect` 相同。

需要清除的 effect

之前，我们研究了如何使用不需要清除的副作用，还有一些副作用是需要清除的。例如订阅外部数据源。这种情况下，清除工作是非常重要的，可以防止引起内存泄露！现在让我们来比较一下如何用 Class 和 Hook 来实现。

使用 Class 的示例

在 React class 中，你通常会在 `componentDidMount` 中设置订阅，并在 `componentWillUnmount` 中清除它。例如，假设我们有一个 `ChatAPI` 模块，它允许我们订阅好友的在线状态。以下是我们如何使用 class 订阅和显示该状态：

```
class FriendStatus extends React.Component {
  constructor(props) {
    super(props);
    this.state = { isOnline: null };
    this.handleStatusChange = this.handleStatusChange.bind(this);
  }

  componentDidMount() {
    ChatAPI.subscribeToFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }

  componentWillUnmount() {
    ChatAPI.unsubscribeFromFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }

  handleStatusChange(status) {
    this.setState({
      isOnline: status.isOnline
    });
  }

  render() {
    if (this.state.isOnline === null) {
      return 'Loading...';
    }
    return this.state.isOnline ? 'Online' : 'Offline';
  }
}
```

你会注意到 `componentDidMount` 和 `componentWillUnmount` 之间相互对应。使用生命周期函数迫使我们拆分这些逻辑代码，即使这两部分代码都作用于相同的副作用。

注意

眼尖的读者可能已经注意到了，这个示例还需要编写 `componentDidUpdate` 方法才能保证完全正确。我们先暂时忽略这一点，本章节中[后续部分](#)会介绍它。

使用 Hook 的示例

如何使用 Hook 编写这个组件。

你可能认为需要单独的 effect 来执行清除操作。但由于添加和删除订阅的代码的紧密性，所以 `useEffect` 的设计是在同一个地方执行。如果你的 effect 返回一个函数，React 将会在执行清除操作时调用它：

```
import React, { useState, useEffect } from 'react';

function FriendStatus(props) {
  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    // Specify how to clean up after this effect:
    return function cleanup() {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}
```

为什么要在 effect 中返回一个函数？ 这是 effect 可选的清除机制。每个 effect 都可以返回一个清除函数。如此可以将添加和移除订阅的逻辑放在一起。它们都属于 effect 的一部分。

React 何时清除 effect？ React 会在组件卸载的时候执行清除操作。正如之前学到的，effect 在每次渲染的时候都会执行。这就是为什么 React 会在执行当前 effect 之前对上一个 effect 进行清除。我们稍后将讨论[为什么这将有助于避免 bug](#)以及[如何在遇到性能问题时跳过此行为](#)。

注意

并不是必须为 effect 中返回的函数命名。这里我们将其命名为 `cleanup` 是为了表明此函数的目的，但其实也可以返回一个箭头函数或者给起一个别的名字。

小结

了解了 `useEffect` 可以在组件渲染后实现各种不同的副作用。有些副作用可能需要清除，所以需要返回一个函数：

```
useEffect(() => {
  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
  return () => {
```

```
    ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
  };
});
```

其他的 effect 可能不必清除，所以不需要返回。

```
useEffect(() => {
  document.title = `You clicked ${count} times`;
});
```

effect Hook 使用同一个 API 来满足这两种情况。

如果你对 **Effect Hook** 的机制已经有很好的把握，或者暂时难以消化更多内容，你现在就可以跳转到[下一章节学习 Hook 的规则](#)。

使用 Effect 的提示

在本节中将继续深入了解 `useEffect` 的某些特性，有经验的 React 使用者可能会对此感兴趣。你不一定要在现在了解他们，你可以随时查看此页面以了解有关 Effect Hook 的更多详细信息。

提示：使用多个 Effect 实现关注点分离

使用 Hook 其中一个目的就是要解决 class 中生命周期函数经常包含不相关的逻辑，但又把相关逻辑分离到了几个不同方法中的问题。下述代码是将前述示例中的计数器和好友在线状态指示器逻辑组合在一起的组件：

```
class FriendStatusWithCounter extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0, isOnline: null };
    this.handleStatusChange = this.handleStatusChange.bind(this);
  }

  componentDidMount() {
    document.title = `You clicked ${this.state.count} times`;
    ChatAPI.subscribeToFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }

  componentDidUpdate() {
    document.title = `You clicked ${this.state.count} times`;
  }

  componentWillUnmount() {
    ChatAPI.unsubscribeFromFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }

  handleStatusChange(status) {
    this.setState({
      isOnline: status.isOnline
    });
  }
}
```

```
    });
}

// ...
```

可以发现设置 `document.title` 的逻辑是如何被分割到 `componentDidMount` 和 `componentDidUpdate` 中的，订阅逻辑又是如何被分割到 `componentDidMount` 和 `componentWillUnmount` 中的。而且 `componentDidMount` 中同时包含了两个不同功能的代码。

那么 Hook 如何解决这个问题呢？就像[你可以使用多个 state 的 Hook 一样](#)，你也可以使用多个 effect。这会将不相关逻辑分离到不同的 effect 中：

```
function FriendStatusWithCounter(props) {
  const [count, setCount] = useState(0);
  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });

  const [isOnline, setIsOnline] = useState(null);
  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });
}

// ...
```

Hook 允许我们按照代码的用途分离他们，而不是像生命周期函数那样。React 将按照 effect 声明的顺序依次调用组件中的每一个 effect。

解释：为什么每次更新的时候都要运行 Effect

如果你已经习惯了使用 class，那么你或许会疑惑为什么 effect 的清除阶段在每次重新渲染时都会执行，而不是只在卸载组件的时候执行一次。让我们看一个实际的例子，看看为什么这个设计可以帮助我们创建 bug 更少的组件。

在[本章节开始时](#)，我们介绍了一个用于显示好友是否在线的 `FriendStatus` 组件。从 class 中 props 读取 `friend.id`，然后在组件挂载后订阅好友的状态，并在卸载组件的时候取消订阅：

```
componentDidMount() {
  ChatAPI.subscribeToFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
  );
}

componentWillUnmount() {
  ChatAPI.unsubscribeFromFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
);
```

但是当组件已经显示在屏幕上时，`friend prop` 发生变化时会发生什么？我们的组件将继续展示原来的好友状态。这是一个 bug。而且我们还会因为取消订阅时使用错误的好友 ID 导致内存泄露或崩溃的问题。

在 class 组件中，我们需要添加 `componentDidUpdate` 来解决这个问题：

```
componentDidMount() {
  ChatAPI.subscribeToFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
  );
}

componentDidUpdate(prevProps) {
  // 取消订阅之前的 friend.id
  ChatAPI.unsubscribeFromFriendStatus(
    prevProps.friend.id,
    this.handleStatusChange
  );
  // 订阅新的 friend.id
  ChatAPI.subscribeToFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
  );
}

componentWillUnmount() {
  ChatAPI.unsubscribeFromFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
);
}
```

忘记正确地处理 `componentDidUpdate` 是 React 应用中常见的 bug 来源。

现在看一下使用 Hook 的版本：

```
function FriendStatus(props) {
  // ...
  useEffect(() => {
    // ...
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });
}
```

它并不会受到此 bug 影响。（虽然我们没有对它做任何改动。）

并不需要特定的代码来处理更新逻辑，因为 `useEffect` 默认就会处理。它会在调用一个新的 effect 之前对前一个 effect 进行清理。为了说明这一点，下面按时间列出一个可能会产生的订阅和取消订阅操作调用序列：

```
ChatAPI.subscribeToFriendStatus(100, handleStatusChange); // 运行第一个 effect
ChatAPI.unsubscribeFromFriendStatus(100, handleStatusChange); // 清除上一个 effect
ChatAPI.subscribeToFriendStatus(200, handleStatusChange); // 运行下一个 effect
ChatAPI.unsubscribeFromFriendStatus(200, handleStatusChange); // 清除上一个 effect
ChatAPI.subscribeToFriendStatus(300, handleStatusChange); // 运行下一个 effect
```

```
//Unmount
ChatAPI.unsubscribeFromFriendStatus(300, handleStatusChange); // 清除最后一个 effect
```

此默认行为保证了一致性，避免了在 class 组件中因为没有处理更新逻辑而导致常见的 bug。

提示：通过跳过 Effect 进行性能优化

在某些情况下，每次渲染后都执行清理或者执行 effect 可能会导致性能问题。在 class 组件中，我们可以通过在 `componentDidUpdate` 中添加对 `prevProps` 或 `prevState` 的比较逻辑解决：

```
componentDidUpdate(prevProps, prevState) {
  if (prevState.count !== this.state.count) {
    document.title = `You clicked ${this.state.count} times`;
  }
}
```

这是很常见的需求，所以它被内置到了 `useEffect` 的 Hook API 中。如果某些特定值在两次重渲染之间没有发生变化，你可以通知 React 跳过对 effect 的调用，只要传递数组作为 `useEffect` 的第二个可选参数即可：

```
useEffect(() => {
  document.title = `You clicked ${count} times`;
}, [count]); // 仅在 count 更改时更新
```

上面这个示例中，我们传入 `[count]` 作为第二个参数。这个参数是什么作用呢？如果 `count` 的值是 `5`，而且我们的组件重渲染的时候 `count` 还是等于 `5`，React 将对前一次渲染的 `[5]` 和后一次渲染的 `[5]` 进行比较。因为数组中的所有元素都是相等的(`5 === 5`)，React 会跳过这个 effect，这就实现了性能的优化。

当渲染时，如果 `count` 的值更新成了 `6`，React 将会把前一次渲染时的数组 `[5]` 和这次渲染的数组 `[6]` 中的元素进行对比。这次因为 `5 !== 6`，React 就会再次调用 effect。如果数组中有多个元素，即使只有一个元素发生变化，React 也会执行 effect。

对于有清除操作的 effect 同样适用：

```
useEffect(() => {
  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
  return () => {
    ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
  };
}, [props.friend.id]); // 仅在 props.friend.id 发生变化时，重新订阅
```

未来版本，可能会在构建时自动添加第二个参数。

注意：

如果你要使用此优化方式，请确保数组中包含了所有外部作用域中会随时间变化并且在 effect 中使用的变量，否则你的代码会引用到先前渲染中的旧变量。参阅文档，了解更多关于[如何处理函数](#)以及[数组频繁变化时的措施](#)内容。

如果想执行只运行一次的 effect (仅在组件挂载和卸载时执行), 可以传递一个空数组 ([]) 作为第二个参数。这就告诉 React 你的 effect 不依赖于 props 或 state 中的任何值, 所以它永远都不需要重复执行。这并不属于特殊情况——它依然遵循依赖数组的工作方式。

如果你传入了一个空数组 ([]), effect 内部的 props 和 state 就会一直拥有其初始值。尽管传入 [] 作为第二个参数更接近大家更熟悉的 `componentDidMount` 和 `componentWillUnmount` 思维模式, 但我们有[更好的方式](#)来避免过于频繁的重复调用 effect。除此之外, 请记得 React 会等待浏览器完成画面渲染之后才会延迟调用 `useEffect`, 因此会使得额外操作很方便。

我们推荐启用 `eslint-plugin-react-hooks` 中的 `exhaustive-deps` 规则。此规则会在添加错误依赖时发出警告并给出修复建议。

下一步

恭喜你！完成本章节学习, 希望关于 effect 的大多数问题都得到了解答。你已经学习了 State Hook 和 Effect Hook, 将它们结合起来你可以做很多事情了。它们涵盖了大多数使用 class 的用例 —— 如果没有, 你可以查看[其他的 Hook](#)。

我们看到了 Hook 如何解决[简介章节中动机部分](#)提出的问题。我们也发现 effect 的清除机制可以避免 `componentDidUpdate` 和 `componentWillUnmount` 中的重复, 同时让相关的代码关联更加紧密, 帮助我们避免一些 bug。我们还看到了我们如何根据 effect 的功能分隔他们, 这是在 class 中无法做到的。

此时你可能会好奇 Hook 是如何工作的。在两次渲染间, React 如何知道哪个 `useState` 调用对应于哪个 state 变量? React 又是如何匹配前后两次渲染中的每一个 effect 的?[在下一章节中我们会学习使用 Hook 的规则](#) —— 这对 Hook 的工作至关重要。

Hooks FAQ

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 React 特性。

此章节回答了关于 *Hook* 的常见问题。

- **采纳策略**

- 哪个版本的 React 包含了 Hook ?
- 我需要重写所有的 class 组件吗 ?
- 有什么是 Hook 能做而 class 做不到的 ?
- 我的 React 知识还有多少是仍然有用的 ?
- 我应该使用 Hook, class, 还是两者混用 ?
- Hook 能否覆盖 class 的所有使用场景 ?
- Hook 会替代 render props 和高阶组件吗 ?
- Hook 对于 Redux connect() 和 React Router 等流行的 API 来说, 意味着什么 ?
- Hook 能和静态类型一起用吗 ?
- 如何测试使用了 Hook 的组件 ?
- lint 规则具体强制了哪些内容 ?

- **从 Class 迁移到 Hook**

- 生命周期方法要如何对应到 Hook ?
- 我该如何使用 Hook 进行数据获取 ?
- 有类似实例变量的东西吗 ?
- 我应该使用单个还是多个 state 变量 ?
- 我可以只在更新时运行 effect 吗 ?
- 如何获取上一轮的 props 或 state ?
- 为什么我会在我的函数中看到陈旧的 props 和 state ?
- 我该如何实现 getDerivedStateFromProps ?
- 有类似 forceUpdate 的东西吗 ?
- 我可以引用一个函数组件吗 ?
- 我该如何测量 DOM 节点 ?
- const [thing, setThing] = useState() 是什么意思 ?

- **性能优化**

- 我可以在更新时跳过 effect 吗 ?
- 在依赖列表中省略函数是否安全 ?
- 如果我的 effect 的依赖频繁变化, 我该怎么办 ?
- 我该如何实现 shouldComponentUpdate ?
- 如何记忆计算结果 ?
- 如何惰性创建昂贵的对象 ?
- Hook 会因为在渲染时创建函数而变慢吗 ?
- 如何避免向下传递回调 ?
- 如何从 useCallback 读取一个经常变化的值 ?

- **底层原理**

- React 是如何把对 Hook 的调用和组件联系起来的 ?

- Hook 使用了哪些现有技术？

采纳策略

哪个版本的 React 包含了 Hook？

从 16.8.0 开始，React 在以下模块中包含了 React Hook 的稳定实现：

- React DOM
- React Native
- React DOM Server
- React Test Renderer
- React Shallow Renderer

请注意，要启用 Hook，所有 React 相关的 package 都必须升级到 **16.8.0** 或更高版本。如果你忘记更新诸如 React DOM 之类的 package，Hook 将无法运行。

React Native 0.59 及以上版本支持 Hook。

我需要重写所有的 class 组件吗？

不。我们并 [没有计划](#) 从 React 中移除 class —— 我们也需要不断地发布产品，重写成本较高。我们推荐在新代码中尝试 Hook。

有什么是 Hook 能做而 class 做不到的？

Hook 提供了强大而富有表现力的方式来在组件间复用功能。通过 「[自定义 Hook](#)」 这一节可以了解能用它做什么。这篇来自一位 React 核心团队的成员的 [文章](#) 则更加深入地剖析了 Hook 解锁了哪些新的能力。

我的 React 知识还有多少是仍然有用的？

Hook 是使用你已经知道的 React 特性的一种更直接的方式 —— 比如 state, 生命周期, context, 以及 refs。它们并没有从根本上改变 React 的工作方式，你对组件, props, 以及自顶向下的数据流的知识并没有改变。

Hook 确实有它们自己的学习曲线。如果这份文档中遗失了一些什么，[提一个 issue](#)，我们会尽可能地帮你。

我应该使用 Hook, class, 还是两者混用？

当你准备好了，我们鼓励你在写新组件的时候开始尝试 Hook。请确保你团队中的每个人都愿意使用它们并且熟知这份文档中的内容。我们不推荐用 Hook 重写你已有的 class，除非你本就打算重写它们。（例如：为了修复bug）。

你不能在 class 组件内部使用 Hook，但毫无疑问你可以在组件树里混合使用 class 组件和使用了 Hook 的函数组件。不论一个组件是 class 还是一个使用了 Hook 的函数，都只是这个组件的实现细节而已。长远来看，我们期望 Hook 能够成为人们编写 React 组件的主要方式。

Hook 能否覆盖 class 的所有使用场景？

我们给 Hook 设定的目标是尽早覆盖 class 的所有使用场景。目前暂时还没有对应不常用的 `getSnapshotBeforeUpdate`, `getDerivedStateFromError` 和 `componentDidCatch` 生命周期的 Hook 等价写法，但我们计划尽早把它们加进来。

Hook 会替代 render props 和高阶组件吗？

通常，render props 和高阶组件只渲染一个子节点。我们认为让 Hook 来服务这个使用场景更加简单。这两种模式仍有用武之地，（例如，一个虚拟滚动条组件或许会有一个 `renderItem` 属性，或是一个可见的容器组件或许会有它自己的 DOM 结构）。但在大部分场景下，Hook 足够了，并且能够帮助减少嵌套。

Hook 对于 Redux `connect()` 和 React Router 等流行的 API 来说，意味着什么？

你可以继续使用之前使用的 API；它们仍会继续有效。

React Redux 从 v7.1.0 开始[支持 Hook API](#) 并暴露了 `useDispatch` 和 `useSelector` 等 hook。

React Router 从 v5.1 开始[支持 hook](#)。

其它第三库也将即将支持 hook。

Hook 能和静态类型一起用吗？

Hook 在设计阶段就考虑了静态类型的问题。因为它们是函数，所以它们比像高阶组件这样的模式更易于设定正确的类型。最新版的 Flow 和 TypeScript React 定义已经包含了对 React Hook 的支持。

重要的是，在你需要严格限制类型的时候，自定义 Hook 能够帮你限制 React 的 API。React 只是给你提供了基础功能，具体怎么用就是你自己的事了。

如何测试使用了 Hook 的组件？

在 React 看来，一个使用了 Hook 的组件只不过是一个常规组件。如果你的测试方案不依赖于 React 的内部实现，测试带 Hook 的组件应该和你通常测试组件的方式没什么差别。

注意

[测试技巧](#) 中包含了许多可以拷贝粘贴的示例。

举个例子，比如我们有这么个计数器组件：

```
function Example() {
  const [count, setCount] = useState(0);
  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

我们会使用 React DOM 来测试它。为了确保它表现得和在浏览器中一样，我们会把代码渲染的部分包裹起来，并更新为 `ReactTestUtils.act()` 调用：

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import { act } from 'react-dom/test-utils';
import Counter from './Counter';

let container;

beforeEach(() => {
  container = document.createElement('div');
  document.body.appendChild(container);
});

afterEach(() => {
  document.body.removeChild(container);
  container = null;
});

it('can render and update a counter', () => {
  // 测试首次渲染和 effect
  act(() => {
    ReactDOM.createRoot(container).render(<Counter />);
  });
  const button = container.querySelector('button');
  const label = container.querySelector('p');
  expect(label.textContent).toBe('You clicked 0 times');
  expect(document.title).toBe('You clicked 0 times');

  // 测试第二次渲染和 effect
  act(() => {
    button.dispatchEvent(new MouseEvent('click', {bubbles: true}));
  });
  expect(label.textContent).toBe('You clicked 1 times');
  expect(document.title).toBe('You clicked 1 times');
});
```

对 `act()` 的调用也会清空它们内部的 effect。

如果你需要测试一个自定义 Hook，你可以在你的测试代码中创建一个组件并在其中使用你的 Hook。然后你就可以测试你刚写的组件了。

为了减少不必要的模板项目，我们推荐使用 [React Testing Library](#)，该项目旨在鼓励你按照终端用户使用组件的方式来编写测试。

欲了解更多，请参阅[测试技巧](#)一节。

lint 规则具体强制了哪些内容？

我们提供了一个 [ESLint 插件](#) 来强制 Hook 规范以避免 Bug。它假设任何以「`use`」开头并紧跟着一个大写字母的函数就是一个 Hook。我们知道这种启发方式并不完美，甚至存在一些假阳性，但如果没有一个全生态范围的约定就没法让 Hook 很好的工作——而名字太长会让人要么不愿意采用 Hook，要么不愿意遵守约定。

规范尤其强制了以下内容：

- 对 Hook 的调用要么在一个 `大驼峰法` 命名的函数（视作一个组件）内部，要么在另一个 `useSomething` 函数（视作一个自定义 Hook）中。
- Hook 在每次渲染时都按照相同的顺序被调用。

还有一些其他的启发方式，但随着我们不断地调优以在发现 Bug 和避免假阳性之间取得平衡，这些方式随时会改变。

从 Class 迁移到 Hook

生命周期方法要如何对应到 Hook？

- `constructor`：函数组件不需要构造函数。你可以通过调用 `useState` 来初始化 state。如果计算的代价比较昂贵，你可以传一个函数给 `useState`。
- `getDerivedStateFromProps`：改为 `在渲染时` 安排一次更新。
- `shouldComponentUpdate`：详见 [下方 `React.memo`](#)。
- `render`：这是函数组件体本身。
- `componentDidMount`, `componentDidUpdate`, `componentWillUnmount`：`useEffect Hook` 可以表达所有这些(包括 `不那么常见` 的场景)的组合。
- `getSnapshotBeforeUpdate`, `componentDidCatch` 以及 `getDerivedStateFromError`：目前还没有这些方法的 Hook 等价写法，但很快会被添加。

我该如何使用 Hook 进行数据获取？

该 [demo](#) 会帮助你开始理解。欲了解更多，请查阅 [此文章](#) 来了解如何使用 Hook 进行数据获取。

有类似实例变量的东西吗？

有！`useRef()` Hook 不仅可以用于 DOM refs。「ref」对象是一个 `current` 属性可变且可以容纳任意值的通用容器，类似于一个 class 的实例属性。

你可以在 `useEffect` 内部对其进行写入：

```
function Timer() {
  const intervalRef = useRef();

  useEffect(() => {
    const id = setInterval(() => {
      // ...
    });
    intervalRef.current = id;
    return () => {
      clearInterval(intervalRef.current);
    };
  });
  // ...
}
```

如果我们只是想设定一个循环定时器，我们不会需要这个 ref (`id` 可以是在 effect 本地的)，但如果我们想要在一个事件处理器中清除这个循环定时器的话这就很有用了：

```
// ...
function handleCancelClick() {
  clearInterval(intervalRef.current);
}
// ...
```

从概念上讲，你可以认为 `refs` 就像是一个 class 的实例变量。除非你正在做 [懒加载](#)，否则避免在渲染期间设置 `refs` —— 这可能会导致意外的行为。相反的，通常你应该在事件处理器和 effects 中修改 `refs`。

我应该使用单个还是多个 state 变量？

如果你之前用过 class，你或许会试图总是在一次 `useState()` 调用中传入一个包含了所有 state 的对象。如果你愿意的话你可以这么做。这里有一个跟踪鼠标移动的组件的例子。我们在本地 state 中记录它的位置和尺寸：

```
function Box() {
  const [state, setState] = useState({ left: 0, top: 0, width: 100, height: 100 });
}
// ...
```

现在假设我们想要编写一些逻辑以便在用户移动鼠标时改变 `left` 和 `top`。注意到我们是如何必须手动把这些字段合并到之前的 state 对象的：

```
// ...
useEffect(() => {
  function handleWindowMouseMove(e) {
    // 展开「...state」以确保我们没有「丢失」width 和 height
    setState(state => ({ ...state, left: e.pageX, top: e.pageY }));
  }
  // 注意：这是个简化版的实现
  window.addEventListener('mousemove', handleWindowMouseMove);
  return () => window.removeEventListener('mousemove', handleWindowMouseMove);
}, []);
// ...
```

这是因为当我们更新一个 state 变量，我们会 替换 它的值。这和 class 中的 `this.setState` 不一样，后者会把更新后的字段 合并 入对象中。

如果你还怀念自动合并，你可以写一个自定义的 `useLegacyState` Hook 来合并对象 state 的更新。然而，**我们推荐把 state 切分成多个 state 变量，每个变量包含的不同值会在同时发生变化。**

举个例子，我们可以把组件的 state 拆分为 `position` 和 `size` 两个对象，并永远以非合并的方式去替换 `position`：

```
function Box() {
  const [position, setPosition] = useState({ left: 0, top: 0 });
  const [size, setSize] = useState({ width: 100, height: 100 });

  useEffect(() => {
    function handleWindowMouseMove(e) {
      setPosition({ left: e.pageX, top: e.pageY });
    }
  }, ...);
```

把独立的 state 变量拆分开还有另外的好处。这使得后期把一些相关的逻辑抽取到一个自定义 Hook 变得容易，比如说：

```
function Box() {
  const position = useWindowPosition();
  const [size, setSize] = useState({ width: 100, height: 100 });
  // ...
}

function useWindowPosition() {
  const [position, setPosition] = useState({ left: 0, top: 0 });
  useEffect(() => {
    // ...
  }, []);
  return position;
}
```

注意看我们是如何做到不改动代码就把对 `position` 这个 state 变量的 `useState` 调用和相关的 effect 移动到一个自定义 Hook 的。如果所有的 state 都存在同一个对象中，想要抽取出来就比较难了。

把所有 state 都放在同一个 `useState` 调用中，或是每一个字段都对应一个 `useState` 调用，这两方式都能跑通。当你在这两个极端之间找到平衡，然后把相关 state 组合到几个独立的 state 变量时，组件就会更加的可读。如果 state 的逻辑开始变得复杂，我们推荐 [用 reducer 来管理它](#)，或使用自定义 Hook。

我可以只在更新时运行 effect 吗？

这是个比较罕见的使用场景。如果你需要的话，你可以 [使用一个可变的 ref 手动存储一个布尔值来表示是首次渲染还是后续渲染](#)，然后在你的 effect 中检查这个标识。（如果你发现自己经常在这么做，你可以为它创建一个自定义 Hook。）

如何获取上一轮的 props 或 state？

在以下两种场景下，你可能会需要获取上一轮的 props 或 state。

有时，你需要获取之前的 props 来 [进行副作用的清理](#)。例如，你可能有这样一个副作用，依赖 `userId` props 来订阅 socket。如果 `userId` 发生了变化，你需要取消 [之前](#) `userId` 的订阅，再进行后续订阅。这种情况下，你无需做额外操作即可实现：

```
useEffect(() => {
  ChatAPI.subscribeToSocket(props.userId);
  return () => ChatAPI.unsubscribeFromSocket(props.userId);
}, [props.userId]);
```

在上述示例中，如果 `userId` 从 `3` 变为 `4`，`ChatAPI.unsubscribeFromSocket(3)` 将会优先运行，然后才会执行 `ChatAPI.subscribeToSocket(4)`。这种情况下，你没必要获取之前的 `userId`，因为清理函数将在闭包中捕获它。

其他情况下，你可能需要 [根据 props 或其他 state 的变化来调整 state](#)。但这一般并不常用，出现这种情况说明你代码中存在重复或多余的 state。然而，如果你需要应对这种场景，你可以在 [状态中存储之前的 state 或 props，并在渲染时更新它们](#)。

我们之前曾建议使用 `usePrevious` 的自定义 Hook 来保持前值。然后，我们发现大多数用例，都属于上述两种场景。如果你的用例与上述两种情况不同，你可以在 [Ref 中对该值进行存储](#) 并在需要时手动更新它。注意，应避免在渲染过程中读取和更新 refs，因为这使得你组件的行为难以预测，且难以理解。

为什么我会在我的函数中看到陈旧的 props 和 state ?

组件内部的任何函数，包括事件处理函数和 effect，都是从它被创建的那次渲染中被「看到」的。例如，考虑这样的代码：

```
function Example() {
  const [count, setCount] = useState(0);

  function handleAlertClick() {
    setTimeout(() => {
      alert('You clicked on: ' + count);
    }, 3000);
  }

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
      <button onClick={handleAlertClick}>
        Show alert
      </button>
    </div>
  );
}
```

如果你先点击「Show alert」然后增加计数器的计数，那这个 alert 会显示在你点击『Show alert』按钮时的 `count` 变量。这避免了那些因为假设 props 和 state 没有改变的代码引起问题。

如果你刻意地想要从某些异步回调中读取 **最新的 state**，你可以用 [一个 ref](#) 来保存它，修改它，并从中读取。

最后，你看到陈旧的 props 和 state 的另一个可能的原因，是你使用了「依赖数组」优化但没有正确地指定所有的依赖。举个例子，如果一个 effect 指定了 `[]` 作为第二个参数，但在内部读取了 `someProp`，它会一直「看到」 `someProp` 的初始值。解决办法是要么移除依赖数组，要么修正它。这里介绍了[你该如何处理函数](#)，而这里介绍了关于如何减少 effect 的运行而不必错误的跳过依赖的[一些常见策略](#)。

注意

我们提供了一个 `exhaustive-deps` ESLint 规则作为 `eslint-plugin-react-hooks` 包的一部分。它会在依赖被错误指定时发出警告，并给出修复建议。

我该如何实现 `getDerivedStateFromProps` ?

尽管你可能[不需要它](#)，但在一些罕见的你需要用到的场景下（比如实现一个 `<Transition>` 组件），你可以在渲染过程中更新 state。React 会立即退出第一次渲染并用更新后的 state 重新运行组件以避免耗费太多性能。

这里我们把 `row` prop 上一轮的值存在一个 state 变量中以便比较：

```
function ScrollView({row}) {
  const [isScrollingDown, setIsScrollingDown] = useState(false);
  const [prevRow, setPrevRow] = useState(null);
```

```

if (row !== prevRow) {
  // Row 自上次渲染以来发生过改变。更新 isScrollingDown。
  setIsScrollingDown(prevRow !== null && row > prevRow);
  setPrevRow(row);
}

return `Scrolling down: ${isScrollingDown}`;
}

```

初看这或许有点奇怪，但渲染期间的一次更新恰恰就是 `getDerivedStateFromProps` 一直以来的概念。

有类似 `forceUpdate` 的东西吗？

如果前后两次的值相同，`useState` 和 `useReducer` Hook 都会放弃更新。原地修改 state 并调用 `setState` 不会引起重新渲染。

通常，你不应该在 React 中修改本地 state。然而，作为一条出路，你可以用一个增长的计数器来在 state 没变的时候依然强制一次重新渲染：

```

const [ignored, forceUpdate] = useReducer(x => x + 1, 0);

function handleClick() {
  forceUpdate();
}

```

可能的话尽量避免这种模式。

我可以引用一个函数组件吗？

尽管你不应该经常需要这么做，但你可以通过 `useImperativeHandle` Hook 暴露一些命令式的方法给父组件。

我该如何测量 DOM 节点？

获取 DOM 节点的位置或是大小的基本方式是使用 `callback ref`。每当 ref 被附加到一个另一个节点，React 就会调用 callback。这里有一个 [小 demo](#)：

```

function MeasureExample() {
  const [height, setHeight] = useState(0);

  const measuredRef = useCallback(node => {
    if (node !== null) {
      setHeight(node.getBoundingClientRect().height);
    }
  }, []);

  return (
    <>
      <h1 ref={measuredRef}>Hello, world</h1>
      <h2>The above header is {Math.round(height)}px tall</h2>
    </>
  );
}

```

在这个案例中，我们没有选择使用 `useRef`，因为当 `ref` 是一个对象时它并不会把当前 `ref` 的值的变化通知到我们。使用 `callback ref` 可以确保 即便子组件延迟显示被测量的节点 (比如为了响应一次点击)，我们依然能够在父组件接收到相关的信息，以便更新测量结果。

注意到我们传递了 `[]` 作为 `useCallback` 的依赖列表。这确保了 `ref callback` 不会在再次渲染时改变，因此 React 不会在非必要的时候调用它。

在此示例中，当且仅当组件挂载和卸载时，`callback ref` 才会被调用，因为渲染的 `<h1>` 组件在整个重新渲染期间始终存在。如果你希望在每次组件调整大小时都收到通知，则可能需要使用 `ResizeObserver` 或基于其构建的第三方 Hook。

如果你愿意，你可以 把这个逻辑抽取出来作为 一个可复用的 Hook:

```
function MeasureExample() {
  const [rect, ref] = useClientRect();
  return (
    <>
      <h1 ref={ref}>Hello, world</h1>
      {rect !== null &&
        <h2>The above header is {Math.round(rect.height)}px tall</h2>
      }
    </>
  );
}

function useClientRect() {
  const [rect, setRect] = useState(null);
  const ref = useCallback(node => {
    if (node !== null) {
      setRect(node.getBoundingClientRect());
    }
  }, []);
  return [rect, ref];
}
```

`const [thing, setThing] = useState()` 是什么意思？

如果你不熟悉这个语法，可以查看 State Hook 文档中的 [解释](#) 一节。

性能优化

我可以在更新时跳过 effect 吗？

可以的。参见 [条件式的发起 effect](#)。注意，忘记处理更新常会 导致 bug，这也正是我们没有默认使用条件式 effect 的原因。

在依赖列表中省略函数是否安全？

一般来说，不安全。

```
function Example({ someProp }) {
  function doSomething() {
    console.log(someProp);
  }

  useEffect(() => {
```

```
    doSomething();
}, []); // 🔴 这样不安全（它调用的 `doSomething` 函数使用了 `someProp`）
}
```

要记住 effect 外部的函数使用了哪些 props 和 state 很难。这也是为什么 **通常你会想要在 effect 内部去声明它所需要的函数**。这样就能容易的看出那个 effect 依赖了组件作用域中的哪些值：

```
function Example({ someProp }) {
useEffect(() => {
  function doSomething() {
    console.log(someProp);
  }

  doSomething();
}, [someProp]); // ✅ 安全（我们的 effect 仅用到了 `someProp`）
}
```

如果这样之后我们依然没用到组件作用域中的任何值，就可以安全地把它指定为 `[]`：

```
useEffect(() => {
  function doSomething() {
    console.log('hello');
  }

  doSomething();
}, []); // ✅ 在这个例子中是安全的，因为我们没有用到组件作用域中的 *任何* 值
```

根据你的用例，下面列举了一些其他办法。

注意

我们提供了一个 `exhaustive-deps` ESLint 规则作为 `eslint-plugin-react-hooks` 包的一部分。它会帮助你找出无法一致地处理更新的组件。

让我们来看看这有什么关系。

如果你指定了一个 [依赖列表](#) 作为 `useEffect`、`useLayoutEffect`、`useMemo`、`useCallback` 或 `useImperativeHandle` 的最后一个参数，它必须包含回调中的所有值，并参与 React 数据流。这就包括 props、state，以及任何由它们衍生而来的东西。

只有 当函数（以及它所调用的函数）不引用 props、state 以及由它们衍生而来的值时，你才能放心地把它们从依赖列表中省略。下面这个案例有一个 Bug：

```
function ProductPage({ productId }) {
  const [product, setProduct] = useState(null);

  async function fetchProduct() {
    const response = await fetch(`http://myapi/product/${productId}`); // 使用了
    productId prop
    const json = await response.json();
    setProduct(json);
  }
}
```

```
useEffect(() => {
  fetchProduct();
}, []); // 🚫 这样是无效的，因为 `fetchProduct` 使用了 `productId`  
// ...
}
```

推荐的修复方案是把那个函数移动到你的 **effect 内部**。这样就能很容易的看出来你的 effect 使用了哪些 props 和 state，并确保它们都被声明了：

```
function ProductPage({ productId }) {
  const [product, setProduct] = useState(null);

  useEffect(() => {
    // 把这个函数移动到 effect 内部后，我们可以清楚地看到它用到的值。
    async function fetchProduct() {
      const response = await fetch('http://myapi/product/' + productId);
      const json = await response.json();
      setProduct(json);
    }

    fetchProduct();
  }, [productId]); // ✅ 有效，因为我们的 effect 只用到了 productId
  // ...
}
```

这同时也允许你通过 effect 内部的局部变量来处理无序的响应：

```
useEffect(() => {
  let ignore = false;
  async function fetchProduct() {
    const response = await fetch('http://myapi/product/' + productId);
    const json = await response.json();
    if (!ignore) setProduct(json);
  }

  fetchProduct();
  return () => { ignore = true };
}, [productId]);
```

我们把这个函数移动到 effect 内部，这样它就不用出现在它的依赖列表中了。

提示

看看 [这个小 demo](#) 和 [这篇文章](#) 来了解更多关于如何用 Hook 进行数据获取。

如果出于某些原因你 无法 把一个函数移动到 **effect 内部**，还有一些其他办法：

- 你可以尝试把那个函数移动到你的 **组件之外**。那样一来，这个函数就肯定不会依赖任何 props 或 state，并且也不用出现在依赖列表中了。
- 如果你所调用的方法是一个纯计算，并且可以在渲染时调用，你可以 转而在 **effect 之外调用它**，并让 effect 依赖于它的返回值。
- 万不得已的情况下，你可以 **把函数加入 effect 的依赖但 把它的定义包裹进 useCallback Hook**。这就确保了它不随渲染而改变，除非 它自身 的依赖发生了改变：

```

function ProductPage({ productId }) {
  // ✅ 用 useCallback 包裹以避免随渲染发生改变
  const fetchProduct = useCallback(() => {
    // ... Does something with productId ...
  }, [productId]); // ✅ useCallback 的所有依赖都被指定了

  return <ProductDetails fetchProduct={fetchProduct} />;
}

function ProductDetails({ fetchProduct }) {
  useEffect(() => {
    fetchProduct();
  }, [fetchProduct]); // ✅ useEffect 的所有依赖都被指定了
  // ...
}

```

注意在上面的案例中，我们 **需要** 让函数出现在依赖列表中。这确保了 `ProductPage` 的 `productId` prop 的变化会自动触发 `ProductDetails` 的重新获取。

如果我的 effect 的依赖频繁变化, 我该怎么办?

有时候，你的 effect 可能会使用一些频繁变化的值。你可能会忽略依赖列表中 state，但这通常会引起 Bug：

```

function Counter() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    const id = setInterval(() => {
      setCount(count + 1); // 这个 effect 依赖于 `count` state
    }, 1000);
    return () => clearInterval(id);
  }, []); // 🚨 Bug: `count` 没有被指定为依赖

  return <h1>{count}</h1>;
}

```

传入空的依赖数组 `[]`，意味着该 hook 只在组件挂载时运行一次，并非重新渲染时。但如此会有问题，在 `setInterval` 的回调中，`count` 的值不会发生变化。因为当 effect 执行时，我们会创建一个闭包，并将 `count` 的值被保存在该闭包当中，且初值为 `0`。每隔一秒，回调就会执行 `setCount(0 + 1)`，因此，`count` 永远不会超过 `1`。

指定 `[count]` 作为依赖列表就能修复这个 Bug，但会导致每次改变发生时定时器都被重置。事实上，每个 `setInterval` 在被清除前（类似于 `setTimeout`）都会调用一次。但这并不是我们想要的。要解决这个问题，我们可以使用 `useState` 的 [函数式更新形式](#)。它允许我们指定 state 该 **如何** 改变而不用引用 **当前 state**：

```

function Counter() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    const id = setInterval(() => {
      setCount(c => c + 1); // ✅ 在这不依赖于外部的 `count` 变量
    }, 1000);
    return () => clearInterval(id);
  }, []); // ✅ 我们的 effect 不使用组件作用域中的任何变量

  return <h1>{count}</h1>;
}

```

(`setCount` 函数的身份是被确保稳定的，所以可以放心的省略掉)

此时，`setInterval` 的回调依旧每秒调用一次，但每次 `setCount` 内部的回调取到的 `count` 是最新值（在回调中变量命名为 `c`）。

在一些更加复杂的场景中（比如一个 `state` 依赖于另一个 `state`），尝试用 `useReducer` Hook 把 `state` 更新逻辑移到 `effect` 之外。这篇文章 提供了一个你该如何做到这一点的案例。`useReducer` 的 `dispatch` 的身份永远是稳定的——即使 `reducer` 函数是定义在组件内部并且依赖 `props`。

万不得已的情况下，如果你想要类似 class 中的 `this` 的功能，你可以 使用一个 `ref` 来保存一个可变的变量。然后你就对它进行读写了。举个例子：

```
function Example(props) {
  // 把最新的 props 保存在一个 ref 中
  const latestProps = useRef(props);
  useEffect(() => {
    latestProps.current = props;
  });

  useEffect(() => {
    function tick() {
      // 在任何时候读取最新的 props
      console.log(latestProps.current);
    }

    const id = setInterval(tick, 1000);
    return () => clearInterval(id);
  }, []);
}
```

仅当你实在找不到更好办法的时候才这么做，因为依赖于变更会使得组件更难以预测。如果有某些特定的模式无法很好地转化成这样，发起一个 issue 并配上可运行的实例代码以便，我们会尽可能帮助你。

我该如何实现 `shouldComponentUpdate`？

你可以用 `React.memo` 包裹一个组件来对它的 `props` 进行浅比较：

```
const Button = React.memo((props) => {
  // 你的组件
});
```

这不是一个 Hook 因为它的写法和 Hook 不同。`React.memo` 等效于 `PureComponent`，但它只比较 `props`。（你也可以通过第二个参数指定一个自定义的比较函数来比较新旧 `props`。如果函数返回 `true`，就会跳过更新。）

`React.memo` 不比较 `state`，因为没有单一的 `state` 对象可供比较。但你也可以让子节点变为纯组件，或者 用 `useMemo` 优化每一个具体的子节点。

如何记忆计算结果？

`useMemo` Hook 允许你通过「记住」上一次计算结果的方式在多次渲染之间缓存计算结果：

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

这行代码会调用 `computeExpensiveValue(a, b)`。但如果依赖数组 `[a, b]` 自上次赋值以来没有改变过，`useMemo` 会跳过二次调用，只是简单复用它上一次返回的值。

记住，传给 `useMemo` 的函数是在渲染期间运行的。不要在其中做任何你通常不会在渲染期间做的事。举个例子，副作用属于 `useEffect`，而不是 `useMemo`。

你可以把 `useMemo` 作为一种性能优化的手段，但不要把它当做一种语义上的保证。未来，React 可能会选择「忘掉」一些之前记住的值并在下一次渲染时重新计算它们，比如为离屏组件释放内存。建议自己编写相关代码以便没有 `useMemo` 也能正常工作——然后把它加入性能优化。(在某些取值必须从不被重新计算的罕见场景，你可以 [惰性初始化](#) 一个 ref。)

方便起见，`useMemo` 也允许你跳过一次子节点的昂贵的重新渲染：

```
function Parent({ a, b }) {
  // Only re-rendered if `a` changes:
  const child1 = useMemo(() => <Child1 a={a} />, [a]);
  // Only re-rendered if `b` changes:
  const child2 = useMemo(() => <Child2 b={b} />, [b]);
  return (
    <>
      {child1}
      {child2}
    </>
  )
}
```

注意这种方式在循环中是无效的，因为 Hook 调用 [不能](#) 被放在循环中。但你可以为列表项抽取一个单独的组件，并在其中调用 `useMemo`。

如何惰性创建昂贵的对象？

如果依赖数组的值相同，`useMemo` 允许你 [记住一次昂贵的计算](#)。但是，这仅作为一种提示，并不 [保证](#) 计算不会重新运行。但有时候需要确保一个对象仅被创建一次。

第一个常见的使用场景是当创建初始 `state` 很昂贵时：

```
function Table(props) {
  // ⚠️ createRows() 每次渲染都会被调用
  const [rows, setRows] = useState(createRows(props.count));
  // ...
}
```

为避免重新创建被忽略的初始 `state`，我们可以传一个 `函数` 给 `useState`：

```
function Table(props) {
  // ✅ createRows() 只会被调用一次
  const [rows, setRows] = useState(() => createRows(props.count));
  // ...
}
```

React 只会在首次渲染时调用这个函数。参见 [useState API 参考](#)。

你或许也会偶尔想要避免重新创建 `useRef()` 的初始值。举个例子，或许你想确保某些命令式的 class 实例只被创建一次：

```
function Image(props) {
  // ⚠️ IntersectionObserver 在每次渲染都会被创建
  const ref = useRef(new IntersectionObserver(onIntersect));
  // ...
}
```

`useRef` 不会像 `useState` 那样接受一个特殊的函数重载。相反，你可以编写你自己的函数来创建并将其设为惰性的：

```
function Image(props) {
  const ref = useRef(null);

  // ✅ IntersectionObserver 只会被惰性创建一次
  function getObserver() {
    if (ref.current === null) {
      ref.current = new IntersectionObserver(onIntersect);
    }
    return ref.current;
  }

  // 当你需要时，调用 getObserver()
  // ...
}
```

这避免了我们在一个对象被首次真正需要之前就创建它。如果你使用 Flow 或 TypeScript，你还可以为了方便给 `getObserver()` 一个不可为 `null` 的类型。

Hook 会因为在渲染时创建函数而变慢吗？

不会。在现代浏览器中，闭包和类的原始性能只有在极端场景下才会有明显的差别。

除此之外，可以认为 Hook 的设计在某些方面更加高效：

- Hook 避免了 class 需要的额外开支，像是创建类实例和在构造函数中绑定事件处理器的成本。
- 符合语言习惯的代码在使用 Hook 时不需要很深的组件树嵌套。这个现象在使用高阶组件、render props、和 context 的代码库中非常普遍。组件树小了，React 的工作量也随之减少。

传统上认为，在 React 中使用内联函数对性能的影响，与每次渲染都传递新的回调会如何破坏子组件的 `shouldComponentUpdate` 优化有关。Hook 从三个方面解决了这个问题。

- `useCallback` Hook 允许你在重新渲染之间保持对相同的回调引用以使得 `shouldComponentUpdate` 继续工作：

```
// 除非 `a` 或 `b` 改变，否则不会变
const memoizedCallback = useCallback(() => {
  doSomething(a, b);
}, [a, b]);
```

- `useMemo` Hook 使得控制具体子节点何时更新变得更容易，减少了对纯组件的需要。
- 最后，`useReducer` Hook 减少了对深层传递回调的依赖，正如下面解释的那样。

如何避免向下传递回调？

我们已经发现大部分人并不喜欢在组件树的每一层手动传递回调。尽管这种写法更明确，但这给人感觉像错综复杂的管道工程一样麻烦。

在大型的组件树中，我们推荐的替代方案是通过 context 用 `useReducer` 往下传一个 `dispatch` 函数：

```
const TodosDispatch = React.createContext(null);

function TodosApp() {
  // 提示：`dispatch` 不会在重新渲染之间变化
  const [todos, dispatch] = useReducer(todosReducer);

  return (
    <TodosDispatch.Provider value={dispatch}>
      <DeepTree todos={todos} />
    </TodosDispatch.Provider>
  );
}
```

`TodosApp` 内部组件树里的任何子节点都可以使用 `dispatch` 函数来向上传递 actions 到 `TodosApp`：

```
function DeepChild(props) {
  // 如果我们想要执行一个 action，我们可以从 context 中获取 dispatch。
  const dispatch = useContext(TodosDispatch);

  function handleClick() {
    dispatch({ type: 'add', text: 'hello' });
  }

  return (
    <button onClick={handleClick}>Add todo</button>
  );
}
```

总而言之，从维护的角度来看更加方便（不用不断转发回调），同时也避免了回调的问题。像这样向下传递 `dispatch` 是处理深度更新的推荐模式。

注意，你依然可以选择将应用的 `state` 作为 `props`（更显明确）向下传递或者使用 `context`（对很深的更新而言更加方便）向下传递。如果你选择使用 `context` 来向下传递 `state`，请使用两种不同的 `context` 类型传递 `state` 和 `dispatch` —— 由于 `dispatch` `context` 永远不会变，因此读取它的组件不需要重新渲染，除非这些组件也需要用到应用程序的 `state`。

如何从 `useCallback` 读取一个经常变化的值？

注意

我们推荐 在 `context` 中向下传递 `dispatch` 而非在 `props` 中使用独立的回调。下面的方法仅仅出于文档完整性考虑，以及作为一条出路在此提及。

在某些罕见场景中，你可能会需要用 `useCallback` 记住一个回调，但由于内部函数必须经常重新创建，记忆效果不是很好。如果你想要记住的函数是一个事件处理器并且在渲染期间没有被用到，你可以 把 ref 当做实例变量 来用，并手动把最后提交的值保存在它当中：

```
function Form() {
  const [text, updateText] = useState('');
  const textRef = useRef();
```

```

useEffect(() => {
  textRef.current = text; // 把它写入 ref
});

const handleSubmit = useCallback(() => {
  const currentText = textRef.current; // 从 ref 读取它
  alert(currentText);
}, [textRef]); // 不要像 [text] 那样重新创建 handleSubmit

return (
  <>
    <input value={text} onChange={e => updateText(e.target.value)} />
    <ExpensiveTree onSubmit={handleSubmit} />
  </>
);
}
}

```

这是一个比较麻烦的模式，但这表示如果你需要的话你可以用这条出路进行优化。如果你把它抽取成一个自定义 Hook 的话会更加好受些：

```

function Form() {
  const [text, updateText] = useState('');
  // 即便 `text` 变了也会被记住：
  const handleSubmit = useEventCallback(() => {
    alert(text);
  }, [text]);

  return (
    <>
      <input value={text} onChange={e => updateText(e.target.value)} />
      <ExpensiveTree onSubmit={handleSubmit} />
    </>
  );
}

function useEventCallback(fn, dependencies) {
  const ref = useRef(() => {
    throw new Error('Cannot call an event handler while rendering.');
  });

  useEffect(() => {
    ref.current = fn;
  }, [fn, ...dependencies]);

  return useCallback(() => {
    const fn = ref.current;
    return fn();
  }, [ref]);
}

```

无论如何，我们都 **不推荐使用这种模式**，只是为了文档的完整性而把它展示在这里。相反的，我们更倾向于 [避免向下深入传递回调](#)。

底层原理

React 是如何把对 Hook 的调用和组件联系起来的？

React 保持对当前渲染中的组件的追踪。多亏了 [Hook 规范](#)，我们得知 Hook 只会在 React 组件中被调用（或自定义 Hook —— 同样只会在 React 组件中被调用）。

每个组件内部都有一个「记忆单元格」列表。它们只不过是我们用来存储一些数据的 JavaScript 对象。当你用 `useState()` 调用一个 Hook 的时候，它会读取当前的单元格（或在首次渲染时将其初始化），然后把指针移动到下一个。这就是多个 `useState()` 调用会得到各自独立的本地 state 的原因。

Hook 使用了哪些现有技术？

Hook 由不同的来源的多个想法构成：

- [react-future](#) 这个仓库中包含我们对函数式 API 的老旧实验。
- React 社区对 render prop API 的实验，其中包括 [Ryan Florence](#) 的 [Reactions Component](#)。
- [Dominic Gannaway](#) 的用 `adopt` 关键字 作为 render props 的语法糖的提案。
- [DisplayScript](#) 中的 state 变量和 state 单元格。
- ReasonReact 中的 [Reducer components](#)。
- Rx 中的 [Subscriptions](#)。
- Multicore OCaml 提到的 [Algebraic effects](#)。

[Sebastian Markbåge](#) 想到了 Hook 最初的设计，后来经过 [Andrew Clark](#), [Sophie Alpert](#), [Dominic Gannaway](#), 和 React 团队的其他成员的提炼。

Hook 简介

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 React 特性。

```
import React, { useState } from 'react';

function Example() {
  // 声明一个新的叫做 "count" 的 state 变量
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

`useState` 是我们要学习的第一个“Hook”，这个例子是简单演示。如果不理解也不用担心。

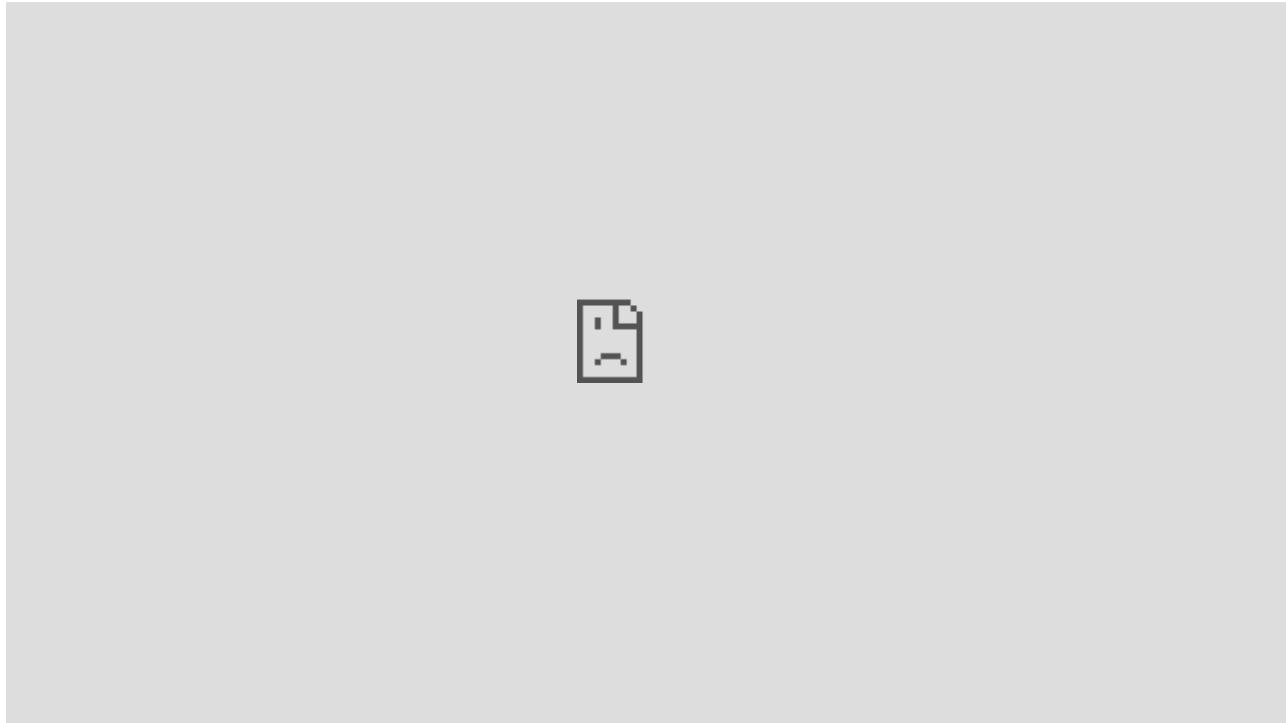
你将在[下一章节](#)正式开始学习 Hook。这一章节，我们将会解释为什么会在 React 中加入 Hook，以及如何使用 Hook 写出更好的应用。

注意

React 16.8.0 是第一个支持 Hook 的版本。升级时，请注意更新所有的 package，包括 React DOM。React Native 从[0.59 版本](#)开始支持 Hook。

视频介绍

在 React Conf 2018 上，Sophie Alpert 和 Dan Abramov 介绍了 Hook，紧接着 Ryan Florence 演示了如何使用 Hook 重构应用。你可以在这里看到这个视频：



没有破坏性改动

在我们继续之前，请记住 Hook 是：

- **完全可选的。** 你无需重写任何已有代码就可以在一些组件中尝试 Hook。但是如果你不想，你不必现在就去学习或使用 Hook。
- **100% 向后兼容的。** Hook 不包含任何破坏性改动。
- **现在可用。** Hook 已发布于 v16.8.0。

没有计划从 React 中移除 class。 你可以在本页[底部的章节](#)读到更多关于 Hook 的渐进策略。

Hook 不会影响你对 React 概念的理解。 恰恰相反，Hook 为已知的 React 概念提供了更直接的 API：props, state, context, refs 以及生命周期。稍后我们将看到，Hook 还提供了一种更强大的方式来组合他们。

如果不想了解添加 Hook 的具体原因，可以直接[跳到下一章节开始学习 Hook！](#) 当然你也可以继续阅读这一章节来了解原因，并且可以了解到如何在不重写应用的情况下使用 Hook。

动机

Hook 解决了我们五年来编写和维护成千上万的组件时遇到的各种各样看起来不相关的问题。无论你正在学习 React，或每天使用，或者更愿尝试另一个和 React 有相似组件模型的框架，你都可能对这些问题似曾相识。

在组件之间复用状态逻辑很难

React 没有提供将可复用性行为“附加”到组件的途径（例如，把组件连接到 store）。如果你使用过 React 一段时间，你也许会熟悉一些解决此类问题的方案，比如 `render props` 和 `高阶组件`。但是这类方案需要重新组织你的组件结构，这可能会很麻烦，使你的代码难以理解。如果你在 React DevTools 中观察过 React 应用，你会发现由 providers, consumers, 高阶组件, `render props` 等其他抽象层组成的组件会形成“嵌套地狱”。尽管我们可以在 [DevTools 过滤掉它们](#)，但这说明了一个更深层次的问题：React 需要为共享状态逻辑提供更好的原生途径。

你可以使用 Hook 从组件中提取状态逻辑，使得这些逻辑可以单独测试并复用。**Hook 使你在无需修改组件结构的情况下复用状态逻辑。** 这使得在组件间或社区内共享 Hook 变得更便捷。

具体将在[自定义 Hook](#) 中对此展开更多讨论。

复杂组件变得难以理解

我们经常维护一些组件，组件起初很简单，但是逐渐会被状态逻辑和副作用充斥。每个生命周期常常包含一些不相关的逻辑。例如，组件常常在 `componentDidMount` 和 `componentDidUpdate` 中获取数据。但是，同一个 `componentDidMount` 中可能也包含很多其它的逻辑，如设置事件监听，而之后需在 `componentWillUnmount` 中清除。相互关联且需要对照修改的代码被进行了拆分，而完全不相关的代码却在同一个方法中组合在一起。如此很容易产生 bug，并且导致逻辑不一致。

在多数情况下，不可能将组件拆分为更小的粒度，因为状态逻辑无处不在。这也给测试带来了一定挑战。同时，这也是很多人将 React 与状态管理库结合使用的原因之一。但是，这往往引入了很多抽象概念，需要你在不同的文件之间来回切换，使得复用变得更加困难。

为了解决这个问题，**Hook 将组件中相互关联的部分拆分成更小的函数（比如设置订阅或请求数据）**，而并非强制按照生命周期划分。你还可以使用 `reducer` 来管理组件的内部状态，使其更加可预测。

我们将在[使用 Effect Hook](#) 中对此展开更多讨论。

难以理解的 class

除了代码复用和代码管理会遇到困难外，我们还发现 `class` 是学习 React 的一大屏障。你必须去理解 JavaScript 中 `this` 的工作方式，这与其他语言存在巨大差异。还不能忘记绑定事件处理器。如果不使用 [ES2022 public class fields](#)，这些代码非常冗余。大家可以很好地理解 `props`, `state` 和自顶向下的数据流，但对 `class` 却一筹莫展。即便在有经验的 React 开发者之间，对于函数组件与 `class` 组件的差异也存在分歧，甚至还要区分两种组件的使用场景。

另外，React 已经发布五年了，我们希望它能在下一个五年也与时俱进。就像 [Svelte](#), [Angular](#), [Glimmer](#) 等其它的库展示的那样，组件[预编译](#)会带来巨大的潜力。尤其是在它不局限于模板的时候。最近，我们一直在使用 [Prepack](#) 来试验 [component folding](#)，也取得了初步成效。但是我们发现使用 `class` 组件会无意中鼓励开发者使用一些让优化措施无效的方案。`class` 也给目前的工具带来了一些问题。例如，`class` 不能很好的压缩，并且会使热重载出现不稳定的情况。因此，我们想提供一个使代码更易于优化的 API。

为了解决这些问题，**Hook 使你在非 class 的情况下可以使用更多的 React 特性。** 从概念上讲，React 组件一直更像是函数。而 Hook 则拥抱了函数，同时也并没有牺牲 React 的精神原则。Hook 提供了问题的解决方案，无需学习复杂的函数式或响应式编程技术。

示例

[Hook 概览](#)是开始学习 Hook 的不错选择。

渐进策略

总结：没有计划从 React 中移除 class。

大部分 React 开发者会专注于开发产品，而没时间关注每一个新 API 的发布。Hook 还很新，也许等到有更多示例和教程后，再考虑学习或使用它们也不迟。

我们也明白向 React 添加新的原生概念的门槛非常高。我们为好奇的读者准备了[详细的征求意见文档](#)，在文档中用更多细节深入讨论了我们推进这件事的动机，也在具体设计决策和相关先进技术上提供了额外的视角。

最重要的是，Hook 和现有代码可以同时工作，你可以渐进式地使用他们。 不用急着迁移到 Hook。我们建议避免任何“大规模重写”，尤其是对于现有的、复杂的 class 组件。开始“用 Hook 的方式思考”前，需要做一些思维上的转变。按照我们的经验，最好先在新的不复杂的组件中尝试使用 Hook，并确保团队中的每一位成员都能适应。在你尝试使用 Hook 后，欢迎给我们提供[反馈](#)，无论好坏。

我们准备让 Hook 覆盖所有 class 组件的使用场景，但是我们将继续为 **class 组件提供支持**。在 Facebook，我们有成千上万的组件用 class 书写，我们完全没有重写它们的计划。相反，我们开始在新的代码中同时使用 Hook 和 class。

FAQ

我们准备了 [Hooks FAQ](#) 来解答最常见的关于 Hook 的问题。

下一步

在本章节的最后，你应该对 Hook 能解决什么问题有了粗略的理解，但可能还有许多细节不清楚。不要担心！[让我们去下一章节通过例子学习 Hook。](#)

[Go to TOC](#)

Hook 概览

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 React 特性。

Hook 是[向下兼容的](#)。本页面为有经验的 React 用户提供一个对 Hook 的概览。这是一个相当快速的概览，如果你有疑惑，可以参阅下面这样的黄色提示框。

详细说明

有关我们为什么要在 React 中引入 Hook 的原因，请参考[动机](#)。

↑↑↑ 每个部分的结尾都会有一个如上所示的黄色方框。它们会链接到更详细的说明。

📌 State Hook

这个例子用来显示一个计数器。当你点击按钮，计数器的值就会增加：

```
import React, { useState } from 'react';

function Example() {
  // 声明一个叫 “count” 的 state 变量。
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

在这里，`useState` 就是一个 *Hook*（等下我们会讲到这是什么意思）。通过在函数组件里调用它来给组件添加一些内部 state。React 会在重复渲染时保留这个 state。`useState` 会返回一对值：当前状态和一个让你更新它的函数，你可以在事件处理函数中或其他一些地方调用这个函数。它类似 class 组件的 `this.setState`，但是它不会把新的 state 和旧的 state 进行合并。（我们会在[使用 State Hook](#) 里展示一个对比 `useState` 和 `this.setState` 的例子）。

`useState` 唯一的参数就是初始 state。在上面的例子中，我们的计数器是从零开始的，所以初始 state 就是 `0`。值得注意的是，不同于 `this.state`，这里的 state 不一定要是一个对象——如果你有需要，它也可以是。这个初始 state 参数只有在第一次渲染时会被用到。

声明多个 state 变量

你可以在一个组件中多次使用 State Hook：

```
function ExampleWithManyStates() {
  // 声明多个 state 变量!
  const [age, setAge] = useState(42);
  const [fruit, setFruit] = useState('banana');
  const [todos, setTodos] = useState([{ text: 'Learn Hooks' }]);
  // ...
}
```

数组解构的语法让我们在调用 `useState` 时可以给 state 变量取不同的名字。当然，这些名字并不是 `useState` API 的一部分。React 假设当你多次调用 `useState` 的时候，你能保证每次渲染时它们的调用顺序是不变的。后面我们会再次解释它是如何工作的以及在什么场景下使用。

那么，什么是 Hook？

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。（我们不推荐把你已有的组件全部重写，但是你可以在新组件里开始使用 Hook。）

React 内置了一些像 `useState` 这样的 Hook。你也可以创建你自己的 Hook 来复用不同组件之间的状态逻辑。我们会先介绍这些内置的 Hook。

详细说明

你可以在这一章节了解更多关于 State Hook 的内容：[使用 State Hook](#)。

⚡ Effect Hook

你之前可能已经在 React 组件中执行过数据获取、订阅或者手动修改过 DOM。我们统一把这些操作称为“副作用”，或者简称为“作用”。

`useEffect` 就是一个 Effect Hook，给函数组件增加了操作副作用的能力。它跟 class 组件中的 `componentDidMount`、`componentDidUpdate` 和 `componentWillUnmount` 具有相同的用途，只不过被合并成了一个 API。（我们会在[使用 Effect Hook](#) 里展示对比 `useEffect` 和这些方法的例子。）

例如，下面这个组件在 React 更新 DOM 后会设置一个页面标题：

```
import React, { useState, useEffect } from 'react';

function Example() {
  const [count, setCount] = useState(0);

  // 相当于 componentDidMount 和 componentDidUpdate:
  useEffect(() => {
    // 使用浏览器的 API 更新页面标题
    document.title = `You clicked ${count} times`;
  });

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

```

        </div>
    );
}

```

当你调用 `useEffect` 时，就是在告诉 React 在完成对 DOM 的更改后运行你的“副作用”函数。由于副作用函数是在组件内声明的，所以它们可以访问到组件的 `props` 和 `state`。默认情况下，React 会在每次渲染后调用副作用函数 —— 包括第一次渲染的时候。（我们会在[使用 Effect Hook](#) 中跟 class 组件的生命周期方法做更详细的对比。）

副作用函数还可以通过返回一个函数来指定如何“清除”副作用。例如，在下面的组件中使用副作用函数来订阅好友的在线状态，并通过取消订阅来进行清除操作：

```

import React, { useState, useEffect } from 'react';

function FriendStatus(props) {
  const [isOnline, setIsOnline] = useState(null);

  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  useEffect(() => {
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);

    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}

```

在这个示例中，React 会在组件销毁时取消对 `ChatAPI` 的订阅，然后在后续渲染时重新执行副作用函数。（如果传给 `ChatAPI` 的 `props.friend.id` 没有变化，你也可以[告诉 React 跳过重新订阅。](#)）

跟 `useState` 一样，你可以在组件中多次使用 `useEffect`：

```

function FriendStatusWithCounter(props) {
  const [count, setCount] = useState(0);
  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });

  const [isOnline, setIsOnline] = useState(null);
  useEffect(() => {
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }
  // ...
}

```

通过使用 Hook，你可以把组件内相关的副作用组织在一起（例如创建订阅及取消订阅），而不要把它们拆分到不同的生命周期函数里。

详细说明

你可以在这一章节了解更多关于 `useEffect` 的内容：[使用 Effect Hook](#)

👉 Hook 使用规则

Hook 就是 JavaScript 函数，但是使用它们会有两个额外的规则：

- 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，我们稍后会学习到。）

同时，我们提供了 [linter 插件](#)来自动执行这些规则。这些规则乍看起来会有一些限制和令人困惑，但是要让 Hook 正常工作，它们至关重要。

详细说明

你可以在这章节了解更多关于这些规则的内容：[Hook 使用规则](#)。

💡 自定义 Hook

有时候我们会想要在组件之间重用一些状态逻辑。目前为止，有两种主流方案来解决这个问题：[高阶组件](#)和 [render props](#)。自定义 Hook 可以让你在不增加组件的情况下达到同样的目的。

前面，我们介绍了一个叫 `FriendStatus` 的组件，它通过调用 `useState` 和 `useEffect` 的 Hook 来订阅一个好友的在线状态。假设我们想在另一个组件里重用这个订阅逻辑。

首先，我们把这个逻辑抽取到一个叫做 `useFriendStatus` 的自定义 Hook 里：

```
import React, { useState, useEffect } from 'react';

function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);

  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  useEffect(() => {
    ChatAPI.subscribeToFriendStatus(friendID, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(friendID, handleStatusChange);
    };
  });
}
```

```
    return isOnline;
}
```

它将 `friendID` 作为参数，并返回该好友是否在线：

现在我们可以在两个组件中使用它：

```
function FriendStatus(props) {
  const isOnline = useFriendStatus(props.friend.id);

  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}

function FriendListItem(props) {
  const isOnline = useFriendStatus(props.friend.id);

  return (
    <li style={{ color: isOnline ? 'green' : 'black' }}>
      {props.friend.name}
    </li>
  );
}
```

每个组件间的 state 是完全独立的。Hook 是一种复用状态逻辑的方式，它不复用 state 本身。事实上 Hook 的每次调用都有一个完全独立的 state —— 因此你可以在单个组件中多次调用同一个自定义 Hook。

自定义 Hook 更像是一种约定而不是功能。如果函数的名字以 “`use`” 开头并调用其他 Hook，我们就说这是一个自定义 Hook。`useSomething` 的命名约定可以让我们的 linter 插件在使用 Hook 的代码中找到 bug。

你可以创建涵盖各种场景的自定义 Hook，如表单处理、动画、订阅声明、计时器，甚至可能还有更多我们没想到的场景。我们很期待看到 React 社区会出现什么样的自定义 Hook。

详细说明

我们会在这一章节介绍更多关于自定义 Hook 的内容：[创建你自己的 Hook](#)。

💡 其他 Hook

除此之外，还有一些使用频率较低的但是很有用的 Hook。比如，`useContext` 让你不使用组件嵌套就可以订阅 React 的 Context。

```
function Example() {
  const locale = useContext(LocaleContext);
  const theme = useContext(ThemeContext);
  // ...
}
```

另外 `useReducer` 可以让你通过 reducer 来管理组件本地的复杂 state。

```
function Todos() {  
  const [todos, dispatch] = useReducer(todosReducer);  
  // ...
```

详细说明

你可以在这一章节了解更多关于所有内置 Hook 的内容：[Hook API 索引](#)。

下一步

嗯，真快！如果你还有什么东西不是很理解或者想要了解更详细的内容，可以继续阅读下一章节：[State Hook](#)。

你也可以查阅 [Hook API 索引](#) 和 [Hooks FAQ](#)。

最后，不要忘记查阅 [Hook 简介](#)，它介绍了我们为什么要增加 Hook 以及如何在不重写整个应用的情况下将 Hook 跟 class 组件同时使用。

[Go to TOC](#)

Hook API 索引

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 React 特性。

本页面主要描述 React 中内置的 Hook API。

如果你刚开始接触 Hook，那么可能需要先查阅 [Hook 概览](#)。你也可以在 [Hooks FAQ](#) 章节中获取有用的信息。

- [基础 Hook](#)
 - [useState](#)
 - [useEffect](#)
 - [useContext](#)
- [额外的 Hook](#)
 - [useReducer](#)
 - [useCallback](#)
 - [useMemo](#)
 - [useRef](#)
 - [useImperativeHandle](#)
 - [useLayoutEffect](#)
 - [useDebugValue](#)
 - [useDeferredValue](#)
 - [useTransition](#)
 - [useId](#)
- [Library Hooks](#)
 - [useSyncExternalStore](#)
 - [useInsertionEffect](#)

基础 Hook

`useState` {#usestate}

```
const [state, setState] = useState(initialState);
```

返回一个 state，以及更新 state 的函数。

在初始渲染期间，返回的状态 (`state`) 与传入的第一个参数 (`initialState`) 值相同。

`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。

```
setState(newState);
```

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。

注意

React 会确保 `setState` 函数的标识是稳定的，并且不会在组件重新渲染时发生变化。这就是为什么可以安全地从 `useEffect` 或 `useCallback` 的依赖列表中省略 `setState`。

函数式更新

如果新的 state 需要通过使用先前的 state 计算得出，那么可以将函数传递给 `setState`。该函数将接收先前的 state，并返回一个更新后的值。下面的计数器组件示例展示了 `setState` 的两种用法：

```
function Counter({initialCount}) {
  const [count, setCount] = useState(initialCount);
  return (
    <>
      Count: {count}
      <button onClick={() => setCount(initialCount)}>Reset</button>
      <button onClick={() => setCount(prevCount => prevCount - 1)}>-</button>
      <button onClick={() => setCount(prevCount => prevCount + 1)}>+</button>
    </>
  );
}
```

“+”和“-”按钮采用函数式形式，因为被更新的 state 需要基于之前的 state。但是“重置”按钮则采用普通形式，因为它总是把 count 设置回初始值。

如果你的更新函数返回值与当前 state 完全相同，则随后的重渲染会被完全跳过。

注意

与 class 组件中的 `setState` 方法不同，`useState` 不会自动合并更新对象。你可以用函数式的 `setState` 结合展开运算符来达到合并更新对象的效果。

```
const [state, setState] = useState({});
setState(prevState => {
  // 也可以使用 Object.assign
  return {...prevState, ...updatedValues};
});
```

`useReducer` 是另一种可选方案，它更适合用于管理包含多个子值的 state 对象。

惰性初始 state

`initialState` 参数只会在组件的初始渲染中起作用，后续渲染时会被忽略。如果初始 state 需要通过复杂计算获得，则可以传入一个函数，在函数中计算并返回初始的 state，此函数只在初始渲染时被调用：

```
const [state, setState] = useState(() => {
  const initialState = someExpensiveComputation(props);
  return initialState;
});
```

跳过 state 更新

如果你更新 State Hook 后的 state 与当前的 state 相同时，React 将跳过子组件的渲染并且不会触发 effect 的执行。（React 使用 `Object.is` 比较算法 来比较 state。）

需要注意的是，React 可能仍需要在跳过渲染前渲染该组件。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。

Batching of state updates

React may group several state updates into a single re-render to improve performance. Normally, this improves performance and shouldn't affect your application's behavior.

Before React 18, only updates inside React event handlers were batched. Starting with React 18, [batching is enabled for all updates by default](#). Note that React makes sure that updates from several *different* user-initiated events -- for example, clicking a button twice -- are always processed separately and do not get batched. This prevents logical mistakes.

In the rare case that you need to force the DOM update to be applied synchronously, you may wrap it in `flushSync`. However, this can hurt performance so do this only where needed.

`useEffect` {#useeffect}

```
useEffect(didUpdate);
```

该 Hook 接收一个包含命令式、且可能有副作用代码的函数。

在函数组件主体内（这里指在 React 渲染阶段）改变 DOM、添加订阅、设置定时器、记录日志以及执行其他包含副作用的操作都是不被允许的，因为这可能会产生莫名其妙的 bug 并破坏 UI 的一致性。

使用 `useEffect` 完成副作用操作。赋值给 `useEffect` 的函数会在组件渲染到屏幕之后执行。你可以把 effect 看作从 React 的纯函数式世界通往命令式世界的逃生通道。

默认情况下，effect 将在每轮渲染结束后执行，但你可以选择让它 [在只有某些值改变的时候 才执行](#)。

清除 effect

通常，组件卸载时需要清除 effect 创建的诸如订阅或计时器 ID 等资源。要实现这一点，`useEffect` 函数需返回一个清除函数。以下就是一个创建订阅的例子：

```
useEffect(() => {
  const subscription = props.source.subscribe();
  return () => {
    // 清除订阅
    subscription.unsubscribe();
  };
});
```

为防止内存泄漏，清除函数会在组件卸载前执行。另外，如果组件多次渲染（通常如此），则[在执行下一个 effect 之前，上一个 effect 就已被清除](#)。在上述示例中，意味着组件的每一次更新都会创建新的订阅。若想避免每次更新都触发 effect 的执行，请参阅下一小节。

effect 的执行时机

与 `componentDidMount`、`componentDidUpdate` 不同的是，传给 `useEffect` 的函数会在浏览器完成布局与绘制之后，在一个延迟事件中被调用。这使得它适用于许多常见的副作用场景，比如设置订阅和事件处理等情况，因为绝大多数操作不应阻塞浏览器对屏幕的更新。

然而，并非所有 effect 都可以被延迟执行。例如，一个对用户可见的 DOM 变更就必须在浏览器执行下一次绘制前被同步执行，这样用户才不会感觉到视觉上的不一致。（概念上类似于被动监听事件和主动监听事件的区别。）React 为此提供了一个额外的 `useLayoutEffect` Hook 来处理这类 effect。它和 `useEffect` 的结构相同，区别只是调用时机不同。

此外，从 React 18 开始，当它是离散的用户输入（如点击）的结果时，或者当它是由 `flushSync` 包装的更新结果时，传递给 `useEffect` 的函数将在屏幕布局和绘制之前同步执行。这种行为便于事件系统或 `flushSync` 的调用者观察该效果的结果。

注意

这只影响传递给 `useEffect` 的函数被调用时 — 在这些 effect 中执行的更新仍会被推迟。这与 `useLayoutEffect` 不同，后者会立即启动该函数并处理其中的更新。

即使在 `useEffect` 被推迟到浏览器绘制之后的情况下，它也能保证在任何新的渲染前启动。React 在开始新的更新前，总会先刷新之前的渲染的 effect。

effect 的条件执行

默认情况下，effect 会在每轮组件渲染完成后执行。这样的话，一旦 effect 的依赖发生变化，它就会被重新创建。

然而，在某些场景下这么做可能会矫枉过正。比如，在上一章节的订阅示例中，我们不需要在每次组件更新时都创建新的订阅，而是仅需要在 `source` prop 改变时重新创建。

要实现这一点，可以给 `useEffect` 传递第二个参数，它是 effect 所依赖的值数组。更新后的示例如下：

```
useEffect(
  () => {
    const subscription = props.source.subscribe();
    return () => {
      subscription.unsubscribe();
    };
  },
  [props.source],
);
```

此时，只有当 `props.source` 改变后才会重新创建订阅。

注意

如果你要使用此优化方式，请确保数组中包含了所有外部作用域中会发生变化且在 effect 中使用的变量，否则你的代码会引用到先前渲染中的旧变量。请参阅文档，了解更多关于[如何处理函数](#)以及[数组频繁变化时的措施](#)的内容。

如果想执行只运行一次的 effect (仅在组件挂载和卸载时执行), 可以传递一个空数组 ([]) 作为第二个参数。这就告诉 React 你的 effect 不依赖于 props 或 state 中的任何值, 所以它永远都不需要重复执行。这并不属于特殊情况——它依然遵循输入数组的工作方式。

如果你传入了一个空数组 ([]), effect 内部的 props 和 state 就会一直持有其初始值。尽管传入 [] 作为第二个参数有点类似于 `componentDidMount` 和 `componentWillUnmount` 的思维模式, 但我们有 [更好的方式](#) 来避免过于频繁的重复调用 effect。除此之外, 请记得 React 会等待浏览器完成画面渲染之后才会延迟调用 `useEffect`, 因此会使得处理额外操作很方便。

我们推荐启用 `eslint-plugin-react-hooks` 中的 `exhaustive-deps` 规则。此规则会在添加错误依赖时发出警告并给出修复建议。

依赖项数组不会作为参数传给 effect 函数。虽然从概念上来说它表现为：所有 effect 函数中引用的值都应该出现在依赖项数组中。未来编译器会更加智能, 届时自动创建数组将成为可能。

`useContext` {#usecontext}

```
const value = useContext(MyContext);
```

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` 或 `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：

- 正确：`useContext(MyContext)`
- 错误：`useContext(MyContext.Consumer)`
- 错误：`useContext(MyContext.Provider)`

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](#)。

提示

如果你在接触 Hook 前已经对 context API 比较熟悉, 那应该可以理解, `useContext(MyContext)` 相当于 class 组件中的 `static contextType = MyContext` 或者 `<MyContext.Consumer>`。

`useContext(MyContext)` 只是让你能够读取 context 的值以及订阅 context 的变化。你仍然需要在上层组件树中使用 `<MyContext.Provider>` 来为下层组件提供 context。

把如下代码与 **Context.Provider** 放在一起

```

const themes = {
  light: {
    foreground: "#000000",
    background: "#eeeeee"
  },
  dark: {
    foreground: "#ffffff",
    background: "#222222"
  }
};

const ThemeContext = React.createContext(themes.light);

function App() {
  return (
    <ThemeContext.Provider value={themes.dark}>
      <Toolbar />
    </ThemeContext.Provider>
  );
}

function Toolbar(props) {
  return (
    <div>
      <ThemedButton />
    </div>
  );
}

function ThemedButton() {
  const theme = useContext(ThemeContext);

  return (
    <button style={{ background: theme.background, color: theme.foreground }}>
      I am styled by theme context!
    </button>
  );
}

```

对先前 [Context 高级指南](#)中的示例使用 hook 进行了修改，你可以在链接中找到有关如何 Context 的更多信息。

额外的 Hook

以下介绍的 Hook，有些是上一节中基础 Hook 的变体，有些则仅在特殊情况下会用到。不用特意预先学习它们。

`useReducer` {#usereducer}

```
const [state, dispatch] = useReducer(reducer, initialArg, init);
```

`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。（如果你熟悉 Redux 的话，就已经知道它如何工作了。）

在某些场景下，`useReducer` 会比 `useState` 更适用，例如 state 逻辑较复杂且包含多个子值，或者下一个 state 依赖于之前的 state 等。并且，使用 `useReducer` 还能给那些会触发深更新的组件做性能优化，因为你可以向子组件传递 `dispatch` 而不是回调函数。

以下是在 reducer 重写 `useState` 一节的计数器示例：

```

const initialState = {count: 0};

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return {count: state.count + 1};
    case 'decrement':
      return {count: state.count - 1};
    default:
      throw new Error();
  }
}

function Counter() {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <>
      Count: {state.count}
      <button onClick={() => dispatch({type: 'decrement'})}>-</button>
      <button onClick={() => dispatch({type: 'increment'})}>+</button>
    </>
  );
}

```

注意

React 会确保 `dispatch` 函数的标识是稳定的，并且不会在组件重新渲染时改变。这就是为什么可以安全地从 `useEffect` 或 `useCallback` 的依赖列表中省略 `dispatch`。

指定初始 state

有两种不同初始化 `useReducer` state 的方式，你可以根据使用场景选择其中的一种。将初始 state 作为第二个参数传入 `useReducer` 是最简单的方法：

```

const [state, dispatch] = useReducer(
  reducer,
  {count: initialCount}
);

```

注意

React 不使用 `state = initialState` 这一由 Redux 推广开来的参数约定。有时候初始值依赖于 `props`，因此需要在调用 Hook 时指定。如果你特别喜欢上述的参数约定，可以通过调用 `useReducer(reducer, undefined, reducer)` 来模拟 Redux 的行为，但我们不鼓励你这么做。

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：

```

function init(initialCount) {
  return {count: initialCount};
}

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return {count: state.count + 1};
    case 'decrement':
      return {count: state.count - 1};
    case 'reset':
      return init(action.payload);
    default:
      throw new Error();
  }
}

function Counter({initialCount}) {
  const [state, dispatch] = useReducer(reducer, initialCount, init);
  return (
    <>
      Count: {state.count}
      <button
        onClick={() => dispatch({type: 'reset', payload: initialCount})}>
        Reset
      </button>
      <button onClick={() => dispatch({type: 'decrement'})}>-</button>
      <button onClick={() => dispatch({type: 'increment'})}>+</button>
    </>
  );
}

```

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 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` {#usecallback}

```

const memoizedCallback = useCallback(
  () => {
    doSomething(a, b);
  },
  [a, b],
);

```

返回一个 `memoized` 回调函数。

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)`。

注意

依赖项数组不会作为参数传给回调函数。虽然从概念上来说它表现为：所有回调函数中引用的值都应该出现在依赖项数组中。未来编译器会更加智能，届时自动创建数组将成为可能。

我们推荐启用 `eslint-plugin-react-hooks` 中的 `exhaustive-deps` 规则。此规则会在添加错误依赖时发出警告并给出修复建议。

`useMemo` {#usememo}

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

返回一个 `memoized` 值。

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。

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不应该在渲染期间内执行的操作，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。

注意

依赖项数组不会作为参数传给“创建”函数。虽然从概念上来说它表现为：所有“创建”函数中引用的值都应该出现在依赖项数组中。未来编译器会更加智能，届时自动创建数组将成为可能。

我们推荐启用 `eslint-plugin-react-hooks` 中的 `exhaustive-deps` 规则。此规则会在添加错误依赖时发出警告并给出修复建议。

`useRef` {#useref}

```
const refContainer = useRef(initialValue);
```

`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。

一个常见的用例便是命令式地访问子组件：

```
function TextInputWithFocusButton() {
  const inputEl = useRef(null);
  const onButtonClick = () => {
    // `current` 指向已挂载到 DOM 上的文本输入元素
```

```

        inputEl.current.focus();
    };
    return (
      <>
        <input ref={inputEl} type="text" />
        <button onClick={onButtonClick}>Focus the input</button>
      </>
    );
}

```

本质上，`useRef` 就像是可以在其 `.current` 属性中保存一个可变值的“盒子”。

你应该熟悉 `ref` 这一种访问 DOM 的主要方式。如果你将 `ref` 对象以 `<div ref={myRef} />` 形式传入组件，则无论该节点如何改变，React 都会将 `ref` 对象的 `.current` 属性设置为相应的 DOM 节点。

然而，`useRef()` 比 `ref` 属性更有用。它可以很方便地保存任何可变值，其类似于在 class 中使用实例字段的方式。

这是因为它创建的是一个普通 Javascript 对象。而 `useRef()` 和自建一个 `{current: ...}` 对象的唯一区别是，`useRef` 会在每次渲染时返回同一个 `ref` 对象。

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` 来实现。

useImperativeHandle {#useimperativehandle}

`useImperativeHandle(ref, createHandle, [deps])`

`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` 一起使用：

```

function FancyInput(props, ref) {
  const inputRef = useRef();
  useImperativeHandle(ref, () => ({
    focus: () => {
      inputRef.current.focus();
    }
  }));
  return <input ref={inputRef} ... />;
}
FancyInput = forwardRef(FancyInput);

```

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

useLayoutEffect {#uselayouтеffect}

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。

提示

如果你正在将代码从 class 组件迁移到使用 Hook 的函数组件，则需要注意 `useLayoutEffect` 与 `componentDidMount`、`componentDidUpdate` 的调用阶段是一样的。但是，我们推荐你一开始先用 `useEffect`，只有当它出问题的时候再尝试使用 `useLayoutEffect`。

如果你使用服务端渲染，请记住，无论 `useLayoutEffect` 还是 `useEffect` 都无法在 Javascript 代码加载完成之前执行。这就是为什么在服务端渲染组件中引入 `useLayoutEffect` 代码时会触发 React 告警。解决这个问题，需要将代码逻辑移至 `useEffect` 中（如果首次渲染不需要这段逻辑的情况下），或是将该组件延迟到客户端渲染完成后显示（如果直到 `useLayoutEffect` 执行之前 HTML 都显示错乱的情况下）。

若要从服务端渲染的 HTML 中排除依赖布局 effect 的组件，可以通过使用 `showChild && <Child />` 进行条件渲染，并使用 `useEffect(() => { setShowChild(true); }, [])` 延迟展示组件。这样，在客户端渲染完成之前，UI 就不会像之前那样显示错乱了。

useDebugValue {#usedebugvalue}

`useDebugValue(value)`

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

例如，“[自定义 Hook](#)”章节中描述的名为 `useFriendStatus` 的自定义 Hook：

```
function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);

  // ...

  // 在开发者工具中的这个 Hook 旁边显示标签
  // e.g. "FriendStatus: Online"
  useDebugValue(isOnline ? 'Online' : 'Offline');

  return isOnline;
}
```

提示

我们不推荐你向每个自定义 Hook 添加 debug 值。当它作为共享库的一部分时才最有价值。

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。

例如，一个返回 `Date` 值的自定义 Hook 可以通过格式化函数来避免不必要的 `toDateString` 函数调用：

`useDebugValue(date, date => date.toDateString());`

useDeferredValue {#usedeferredvalue}

```
const deferredValue = useDeferredValue(value);
```

`useDeferredValue` 接受一个值，并返回该值的新副本，该副本将推迟到更紧急地更新之后。如果当前渲染是一个紧急更新的结果，比如用户输入，React 将返回之前的值，然后在紧急渲染完成后渲染新的值。

该 hook 与使用防抖和节流去延迟更新的用户空间 hooks 类似。使用 `useDeferredValue` 的好处是，React 将在其他工作完成（而不是等待任意时间）后立即进行更新，并且像 `startTransition` 一样，延迟值可以暂停，而不会触发现有内容的意外降级。

Memoizing deferred children

`useDeferredValue` 仅延迟你传递给它的值。如果你想要在紧急更新期间防止子组件重新渲染，则还必须使用 `React.memo` 或 `React.useMemo` 记忆孩子组件：

```
function Typeahead() {
  const query = useSearchQuery('');
  const deferredQuery = useDeferredValue(query);

  // Memoizing 告诉 React 仅当 deferredQuery 改变,
  // 而不是 query 改变的时候才重新渲染
  const suggestions = useMemo(() =>
    <SearchSuggestions query={deferredQuery} />,
    [deferredQuery]
  );

  return (
    <>
      <SearchInput query={query} />
      <Suspense fallback="Loading results...">
        {suggestions}
      </Suspense>
    </>
  );
}
```

记忆该子组件告诉 React 它仅当 `deferredQuery` 改变而不是 `query` 改变的时候才需要去重新渲染。这个限制不是 `useDeferredValue` 独有的，它和使用防抖或节流的 hooks 使用的相同模式。

useTransition {#usetransition}

```
const [isPending, startTransition] = useTransition();
```

返回一个状态值表示过渡任务的等待状态，以及一个启动该过渡任务的函数。

`startTransition` 允许你通过标记更新将提供的回调函数作为一个过渡任务：

```
startTransition(() => {
  setCount(count + 1);
})
```

`isPending` 指示过渡任务何时活跃以显示一个等待状态：

```

function App() {
  const [isPending, startTransition] = useTransition();
  const [count, setCount] = useState(0);

  function handleClick() {
    startTransition(() => {
      setCount(c => c + 1);
    })
  }

  return (
    <div>
      {isPending && <Spinner />}
      <button onClick={handleClick}>{count}</button>
    </div>
  );
}

```

注意：

过渡任务中触发的更新会让更紧急地更新先进行，比如点击。

过渡任务中的更新将不会展示由于再次挂起而导致降级的内容。这个机制允许用户在 React 渲染更新的时候继续与当前内容进行交互。

useId {#useid}

```
const id = useId();
```

`useId` 是一个用于生成横跨服务端和客户端的稳定的唯一 ID 的同时避免 hydration 不匹配的 hook。

Note

`useId` is **not** for generating `keys` in a list. Keys should be generated from your data.

For a basic example, pass the `id` directly to the elements that need it: 一个最简单的例子，直接传递 `id` 给需要它的元素：

```

function Checkbox() {
  const id = useId();
  return (
    <>
      <label htmlFor={id}>Do you like React?</label>
      <input id={id} type="checkbox" name="react"/>
    </>
  );
}

```

对于同一组件中的多个 ID，使用相同的 `id` 并添加后缀：

```

function NameFields() {
  const id = useId();
  return (
    <div>
      <label htmlFor={id + '-firstName'}>First Name</label>
      <div>
        <input id={id + '-firstName'} type="text" />
      </div>
      <label htmlFor={id + '-lastName'}>Last Name</label>
      <div>
        <input id={id + '-lastName'} type="text" />
      </div>
    </div>
  );
}

```

注意：

`useId` 生成一个包含`:`的字符串 token。这有助于确保 token 是唯一的，但在 CSS 选择器或 `querySelectorAll` 等 API 中不受支持。

`useId` 支持 `identifierPrefix` 以防止在多个根应用的程序中发生冲突。要进行配置，请参阅 `hydrateRoot` 和 `ReactDOMServer` 的选项。

Library Hooks

以下 hook 是为库作者提供的，用于将库深入集成到 React 模型中，通常不会在应用程序代码中使用。

`useSyncExternalStore` {#usesyncexternalstore}

```
const state = useSyncExternalStore(subscribe, getSnapshot[], getServerSnapshot);
```

`useSyncExternalStore` 是一个推荐用于读取和订阅外部数据源的 hook，其方式与选择性的 hydration 和时间切片等并发渲染功能兼容。

此方法返回存储的值并接受三个参数：

- `subscribe`：用于注册一个回调函数，当存储值发生更改时被调用。
- `getSnapshot`：返回当前存储值的函数。
- `getServerSnapshot`：返回服务端渲染期间使用的存储值的函数

最简单的例子就是订阅整个存储值：

```
const state = useSyncExternalStore(store.subscribe, store.getSnapshot);
```

然而，你也可以订阅特定字段：

```

const selectedField = useSyncExternalStore(
  store.subscribe,
  () => store.getSnapshot().selectedField,
);

```

当服务端渲染的时候，你必须序列化在服务端使用的存储值，并将其提供给 `useSyncExternalStore`。React 将在 hydration 过程中使用此快照来防止服务端不匹配：

```
const selectedField = useSyncExternalStore(
  store.subscribe,
  () => store.getSnapshot().selectedField,
  () => INITIAL_SERVER_SNAPSHOT.selectedField,
);
```

注意：

`getSnapshot` 必须返回缓存的值。如果 `getSnapshot` 连续多次调用，则必须返回相同的确切值，除非中间有存储值更新。

提供了一个楔子，发布为 `use-sync-external-store/shim`，用于支持多种版本的 React。如果可用，楔子将首选 `useSyncExternalStore`，如果不可用，则降级选择用户空间的实现。

为了方便起见，我们还提供了一个版本的 API，该 API 发布为 `use-sync-external-store/with-selector`，其自动支持记忆 `getSnapshot` 的结果。

useInsertionEffect {#useinsertioneffect}

```
useInsertionEffect(didUpdate);
```

该签名与 `useEffect` 相同，但它在所有 DOM 突变 之前 同步触发。使用它在读取 `useLayoutEffect` 中的布局之前将样式注入 DOM。由于这个 hook 的作用域有限，所以这个 hook 不能访问 refs，也不能安排更新。

注意：

`useInsertionEffect` 应仅限于 `css-in-js` 库作者使用。优先考虑使用 `useEffect` 或 `useLayoutEffect` 来替代。

Hook 规则

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 React 特性。

Hook 本质就是 JavaScript 函数，但是在使用它时需要遵循两条规则。我们提供了一个 [linter 插件](#) 来强制执行这些规则：

只在最顶层使用 Hook

不要在循环、条件或嵌套函数中调用 Hook，确保总是在你的 React 函数的最顶层以及任何 return 之前调用他们。遵守这条规则，你就能确保 Hook 在每一次渲染中都按照同样的顺序被调用。这让 React 能够在多次的 `useState` 和 `useEffect` 调用之间保持 hook 状态的正确。(如果你对此感到好奇，我们在[下面](#)会有更深入的解释。)

只在 React 函数中调用 Hook

不要在普通的 JavaScript 函数中调用 Hook。你可以：

- 在 React 的函数组件中调用 Hook
- 在自定义 Hook 中调用其他 Hook (我们将会在[下一页](#) 中学习这个。)

遵循此规则，确保组件的状态逻辑在代码中清晰可见。

ESLint 插件

我们发布了一个名为 `eslint-plugin-react-hooks` 的 ESLint 插件来强制执行这两条规则。如果你想尝试一下，可以将此插件添加到你的项目中：

我们打算后续版本默认添加此插件到 [Create React App](#) 及其他类似的工具包中。

```
npm install eslint-plugin-react-hooks --save-dev
```

```
// 你的 ESLint 配置
{
  "plugins": [
    // ...
    "react-hooks"
  ],
  "rules": {
    // ...
    "react-hooks/rules-of-hooks": "error", // 检查 Hook 的规则
    "react-hooks/exhaustive-deps": "warn" // 检查 effect 的依赖
  }
}
```

现在你可以跳转到下一章节学习如何编写[你自己的 Hook](#)。在本章节中，我们将继续解释这些规则背后的原因。

说明

正如我们[之前学到的](#)，我们可以在单个组件中使用多个 State Hook 或 Effect Hook：

```

function Form() {
  // 1. Use the name state variable
  const [name, setName] = useState('Mary');

  // 2. Use an effect for persisting the form
  useEffect(function persistForm() {
    localStorage.setItem('formData', name);
  });

  // 3. Use the surname state variable
  const [surname, setSurname] = useState('Poppins');

  // 4. Use an effect for updating the title
  useEffect(function updateTitle() {
    document.title = name + ' ' + surname;
  });

  // ...
}

```

那么 React 怎么知道哪个 state 对应哪个 `useState`？答案是 React 靠的是 Hook 调用的顺序。因为我们的示例中，Hook 的调用顺序在每次渲染中都是相同的，所以它能够正常工作：

```

// -----
// 首次渲染
// -----
useState('Mary')          // 1. 使用 'Mary' 初始化变量名为 name 的 state
useEffect(persistForm)    // 2. 添加 effect 以保存 form 操作
useState('Poppins')       // 3. 使用 'Poppins' 初始化变量名为 surname 的 state
useEffect(updateTitle)    // 4. 添加 effect 以更新标题

// -----
// 二次渲染
// -----
useState('Mary')          // 1. 读取变量名为 name 的 state (参数被忽略)
useEffect(persistForm)    // 2. 替换保存 form 的 effect
useState('Poppins')       // 3. 读取变量名为 surname 的 state (参数被忽略)
useEffect(updateTitle)    // 4. 替换更新标题的 effect

// ...

```

只要 Hook 的调用顺序在多次渲染之间保持一致，React 就能正确地将内部 state 和对应的 Hook 进行关联。但如果我们将一个 Hook (例如 `persistForm` effect) 调用放到一个条件语句中会发生什么呢？

```

// ① 在条件语句中使用 Hook 违反第一条规则
if (name !== '') {
  useEffect(function persistForm() {
    localStorage.setItem('formData', name);
  });
}

```

在第一次渲染中 `name !== ''` 这个条件值为 `true`，所以我们会执行这个 Hook。但是下一次渲染时我们可能清空了表单，表达式值变为 `false`。此时的渲染会跳过该 Hook，Hook 的调用顺序发生了改变：

```

useState('Mary')          // 1. 读取变量名为 name 的 state (参数被忽略)
// useEffect(persistForm) // ② 此 Hook 被忽略!
useState('Poppins')       // ② 2 (之前为 3)。读取变量名为 surname 的 state 失败
useEffect(updateTitle)    // ② 3 (之前为 4)。替换更新标题的 effect 失败

```

React 不知道第二个 `useState` 的 Hook 应该返回什么。React 会以为在该组件中第二个 Hook 的调用像上次的渲染一样，对应的是 `persistForm` 的 effect，但并非如此。从这里开始，后面的 Hook 调用都被提前执行，导致 bug 的产生。

这就是为什么 Hook 需要在我们组件的最顶层调用。如果我们想要有条件地执行一个 effect，可以将判断放到 Hook 的内部：

```
useEffect(function persistForm() {
  // 👍 将条件判断放置在 effect 中
  if (name !== '') {
    localStorage.setItem('formData', name);
  }
});
```

注意：如果使用了提供的 lint 插件，就无需担心此问题。不过你现在知道了为什么 Hook 会这样工作，也知道了这个规则是为了避免什么问题。

下一步

最后，接下来会学习**如何编写自定义 Hook**！自定义 Hook 可以将 React 中提供的 Hook 组合到定制的 Hook 中，以复用不同组件之间常见的状态逻辑。

使用 State 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 React 特性。

[Hook 简介章节](#) 中使用下面的例子介绍了 Hook：

```
import React, { useState } from 'react';

function Example() {
  // 声明一个叫 "count" 的 state 变量
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

我们将通过将这段代码与一个等价的 class 示例进行比较来开始学习 Hook。

等价的 class 示例

如果你之前在 React 中使用过 class，这段代码看起来应该很熟悉：

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Click me
        </button>
      </div>
    );
  }
}
```

state 初始值为 `{ count: 0 }`，当用户点击按钮后，我们通过调用 `this.setState()` 来增加 `state.count`。整个章节中都将使用该 class 的代码片段做示例。

注意

你可能想知道为什么我们在这里使用一个计数器例子而不是一个更实际的示例。因为我们还只是初步接触 Hook，这可以帮助我们将注意力集中到 API 本身。

Hook 和函数组件

复习一下，React 的函数组件是这样的：

```
const Example = (props) => {
  // 你可以在这使用 Hook
  return <div />;
}
```

或是这样：

```
function Example(props) {
  // 你可以在这使用 Hook
  return <div />;
}
```

你之前可能把它们叫做“无状态组件”。但现在我们为它们引入了使用 React state 的能力，所以我们更喜欢叫它“函数组件”。

Hook 在 class 内部是不起作用的。但你可以使用它们来取代 class。

Hook 是什么？

在新示例中，首先引入 React 中 `useState` 的 Hook

```
import React, { useState } from 'react';

function Example() {
  // ...
}
```

Hook 是什么？ Hook 是一个特殊的函数，它可以让你“钩入”React 的特性。例如，`useState` 是允许你在 React 函数组件中添加 state 的 Hook。稍后我们将学习其他 Hook。

什么时候我会用 Hook？ 如果你在编写函数组件并意识到需要向其添加一些 state，以前的做法是必须将其转化为 class。现在你可以在现有的函数组件中使用 Hook。

注意：

在组件中有些特殊的规则，规定什么地方能使用 Hook，什么地方不能使用。我们将在 [Hook 规则](#) 中学习它们。

声明 State 变量

在 class 中，我们通过在构造函数中设置 `this.state` 为 `{ count: 0 }` 来初始化 `count` state 为 `0`：

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
}
```

在函数组件中，我们没有 `this`，所以我们不能分配或读取 `this.state`。我们直接在组件中调用 `useState` Hook：

```
import React, { useState } from 'react';

function Example() {
  // 声明一个叫 "count" 的 state 变量
  const [count, setCount] = useState(0);
```

调用 `useState` 方法的时候做了什么？ 它定义一个“state 变量”。我们的变量叫 `count`，但是我们可以叫他任何名字，比如 `banana`。这是一种在函数调用时保存变量的方式——`useState` 是一种新方法，它与 class 里面的 `this.state` 提供的功能完全相同。一般来说，在函数退出后变量就会“消失”，而 state 中的变量会被 React 保留。

`useState` 需要哪些参数？ `useState()` 方法里面唯一的参数就是初始 state。不同于 class 的是，我们可以按照需要使用数字或字符串对其进行赋值，而不一定是对象。在示例中，只需使用数字来记录用户点击次数，所以我们传了 `0` 作为变量的初始 state。（如果我们想要在 state 中存储两个不同的变量，只需调用 `useState()` 两次即可。）

`useState` 方法的返回值是什么？ 返回值为：当前 state 以及更新 state 的函数。这就是我们写 `const [count, setCount] = useState()` 的原因。这与 class 里面 `this.state.count` 和 `this.setState` 类似，唯一区别就是你需要成对的获取它们。如果你不熟悉我们使用的语法，我们会在[本章节的底部](#)介绍它。

既然我们知道了 `useState` 的作用，我们的示例应该更容易理解了：

```
import React, { useState } from 'react';

function Example() {
  // 声明一个叫 "count" 的 state 变量
  const [count, setCount] = useState(0);
```

我们声明了一个叫 `count` 的 state 变量，然后把它设为 `0`。React 会在重复渲染时记住它当前的值，并且提供最新的值给我们的函数。我们可以通过调用 `setCount` 来更新当前的 `count`。

注意

你可能想知道：为什么叫 `useState` 而不叫 `createState`？

“Create”可能不是很准确，因为 state 只在组件首次渲染的时候被创建。在下一次重新渲染时，`useState` 返回给我们当前的 state。否则它就不是“state”了！这也是 Hook 的名字总是以 `use` 开头的一个原因。我们将在后面的[Hook 规则](#)中了解原因。

读取 State

当我们想在 class 中显示当前的 count，我们读取 `this.state.count`：

```
<p>You clicked {this.state.count} times</p>
```

在函数中，我们可以直接用 `count`：

```
<p>You clicked {count} times</p>
```

更新 State

在 class 中，我们需要调用 `this.setState()` 来更新 `count` 值：

```
<button onClick={() => this.setState({ count: this.state.count + 1 })}>
  Click me
</button>
```

在函数中，我们已经有了 `setCount` 和 `count` 变量，所以我们不需要 `this`：

```
<button onClick={() => setCount(count + 1)}>
  Click me
</button>
```

总结

现在让我们来仔细回顾一下学到的知识，看下我们是否真正理解了。

```
1: import React, { useState } from 'react';
2:
3: function Example() {
4:   const [count, setCount] = useState(0);
5:
6:   return (
7:     <div>
8:       <p>You clicked {count} times</p>
9:       <button onClick={() => setCount(count + 1)}>
10:         Click me
11:       </button>
12:     </div>
13:   );
14: }
```

- **第一行：**引入 React 中的 `useState` Hook。它让我们在函数组件中存储内部 state。
- **第四行：**在 `Example` 组件内部，我们通过调用 `useState` Hook 声明了一个新的 state 变量。它返回一对值给到我们命名的变量上。我们把变量命名为 `count`，因为它存储的是点击次数。我们通过传 `0` 作为 `useState` 唯一的参数来将其初始化为 `0`。第二个返回的值本身就是一个函数。它让我们可以更新 `count` 的值，所以我们叫它 `setCount`。
- **第九行：**当用户点击按钮后，我们传递一个新的值给 `setCount`。React 会重新渲染 `Example` 组件，并把最新的 `count` 传给它。

乍一看这似乎有点太多了。不要急于求成！如果你有不理解的地方，请再次查看以上代码并从头到尾阅读。我们保证一旦你试着“忘记” class 里面 state 是如何工作的，并用新的眼光看这段代码，就容易理解了。

提示：方括号有什么用？

你可能注意到我们用方括号定义了一个 state 变量

```
const [count, setCount] = useState(0);
```

等号左边名字并不是 React API 的部分，你可以自己取名字：

```
const [fruit, setFruit] = useState('banana');
```

这种 JavaScript 语法叫 [数组解构](#)。它意味着我们同时创建了 `fruit` 和 `setFruit` 两个变量，`fruit` 的值为 `useState` 返回的第一个值，`setFruit` 是返回的第二个值。它等价于下面的代码：

```
var fruitStateVariable = useState('banana'); // 返回一个有两个元素的数组
var fruit = fruitStateVariable[0]; // 数组里的第一个值
var setFruit = fruitStateVariable[1]; // 数组里的第二个值
```

当我们使用 `useState` 定义 state 变量时候，它返回一个有两个值的数组。第一个值是当前的 state，第二个值是更新 state 的函数。使用 `[0]` 和 `[1]` 来访问有点令人困惑，因为它们有特定的含义。这就是我们使用数组解构的原因。

注意

你可能会好奇 React 怎么知道 `useState` 对应的是哪个组件，因为我们并没有传递 `this` 给 React。我们将在 FAQ 部分回答[这个问题](#)以及许多其他问题。

提示：使用多个 state 变量

将 state 变量声明为一对 `[something, setSomething]` 也很方便，因为如果我们想使用多个 state 变量，它允许我们给不同的 state 变量取不同的名称：

```
function ExampleWithManyStates() {
  // 声明多个 state 变量
  const [age, setAge] = useState(42);
  const [fruit, setFruit] = useState('banana');
  const [todos, setTodos] = useState([{ text: '学习 Hook' }]);
}
```

在以上组件中，我们有局部变量 `age`，`fruit` 和 `todos`，并且我们可以单独更新它们：

```
function handleOrangeClick() {
  // 和 this.setState({ fruit: 'orange' }) 类似
  setFruit('orange');
}
```

你不必使用多个 state 变量。State 变量可以很好地存储对象和数组，因此，你仍然可以将相关数据分为一组。然而，不像 class 中的 `this.setState`，更新 state 变量总是替换它而不是合并它。

我们在 [FAQ 中](#) 提供了更多关于分离独立 state 变量的建议。

下一步

从上述内容中，我们了解了 React 提供的 `useState` Hook，有时候我们也叫它“State Hook”。它让我们在 React 函数组件上添加内部 state —— 这是我们首次尝试。

我们还学到了一些知识比如什么是 Hook。Hook 是能让你在函数组件中“钩入”React 特性的函数。它们名字通常都以 `use` 开始，还有更多 Hook 等着我们去探索。

现在我们将[学习另一个 Hook: `useEffect`](#)。 它能在函数组件中执行副作用，并且它与 class 中的生命周期函数极为类似。

贡献流程

React 是 Facebook 首批开源项目中的一员，开发状态保持活跃，并在 facebook.com 上为我们提供了源代码。现在，我们正不断解决若干个问题以使参与贡献 React 尽可能容易和公开透明，虽然目前还差得很远。我们希望本篇文章能够解释清楚贡献的流程，回答你可能会有的一些问题。

行为规范 {#code-of-conduct}

Facebook 将[参与者公约](#)作为行为规范，我们希望参与项目的各位严格遵守。请阅读[全文](#)去了解什么行为允许，什么行为不允许。

人人皆可开发

React 的一切工作在 [GitHub](#) 上完成，核心团队及其以外的贡献者发送 pull requests，其代码评审流程皆为一致。

语义化版本

React 遵循[语义化版本](#)。我们对重要的漏洞修复发布修订号；对新特性或不重要的变更发布次版本号；对重大且不兼容的变更发布主版本号。我们在开发重大且不兼容的变更时，还会在次版本号用 deprecation warnings 让用户得知将来的变更并提前迁移代码。请查看[版本号规则](#)来了解更多我们在稳定性和渐进迁移方面要做哪些事情。

每一个重要变更参见 [changelog file](#)。

分支管理

请直接提交你的变更至 `main branch`。对于开发或即将推出的版本，我们不会另建分支。我们尽力保持 `main` 不出问题，并通过所有测试。

合并进入 `main` 的代码必须与最新稳定版本兼容，可以有额外特性，但不能有重大变更。我们应从 `main` 随时能发布新的次版本号。

特性切换(Feature Flags)

我们为了使 `main` 能够发布，要求重大且不兼容的变更和实验性的特性必须用特性切换。

`packages/shared/ReactFeatureFlags.js` 中定义了特性切换。React 的一些版本可能启用了不同的特性切换；比如，React Native 可能与 React DOM 有不同的配置。这些特性切换见于 `packages/shared/forks`。特性切换使用了静态类型检查器 Flow，因此你可以运行 `yarn flow` 来确认所有必要文件已更新。

React 的构建系统（Build System）会先删去禁用的特性分支，之后再发布。每次 commit 都会运行持续集成（Continuous Integration）来检查包（Bundle）大小的变化。包大小的变化可以用来表明某特性正确合并。

漏洞

何处查找已知 issue

我们用 [GitHub Issues](#) 来公开漏洞。我们密切关注该版块，内部解决 bug 时也会想办法说明清楚。在你提交 issue 前，请确定没有重复 issue 出现。

报告新的 issue

修复 bug 的最佳方法是给出缩略版的测试用例。这个 [JSFiddle 模板](#)是个不错的起点。

安全漏洞

为了发现安全漏洞, Facebook 实行了漏洞[举报奖励制度](#) (Bounty Program)。为此, 请不要将这类漏洞提交到 public issues, 而要遵循举报奖励制度页面所描述的流程。

如何联系我们

- 因特网中继聊天 (IRC) : [#reactjs on freenode](#)
- [论坛](#)

你如果需要有关 React 的帮助, 还可以前往 Discord 聊天平台, 这里建有 React 用户们的[活跃社区](#)。

提出变更

如果你想改变公开的 API, 或者对实现有不小的变更, 我们建议你[发起 issue](#), 这会让我们先对你的提议达成一致, 然后再着手工作。

如果只是修复漏洞, 你当然可以立即提交 pull request, 不过我们还是建议先去提出 issue 来说明你修复了什么, 这就对一种情况来说就很方便: 我们没有接受特定 bug 的修复但想跟进该 issue 的情况。

首个 Pull Request

在写第一个 Pull Request ? 你可以从这一系列视频中学习怎么做 :

[How to Contribute to an Open Source Project on GitHub](#)

为了使你能够快速上手和熟悉贡献流程, 我们这里有个列表 [good first issues](#), 里面有相对没那么笼统的漏洞, 从这开始是个不错的选择。

如果你想解决一个 issue, 请确定检查了该 issue 下的评论以防有人正在处理它。如果目前没人在处理该 issue, 那么请留下评论去表明你想处理该 issue 以便其他人不会意外重复你的工作。

如果有人留言表明要处理该 issue 但是超过两周没有跟进, 你可以接手工作, 不过也应该留言说明。

提交 Pull Request

核心团队时刻关注 pull requests, 我们会先评审你的 pull request, 之后可能会合并, 可能会要求再次更改, 也可能会关闭该 pull request 并对此作出解释。对于 API 上的变更, 我们可能得确认在 Facebook.com 上的内部 API 使用方法, 因此该变更可能会推迟。我们会尽力全程更新和反馈。

提交 pull request 前, 请确保完成以下步骤 :

1. Fork [此仓库](#), 从 `main` 创建分支。
2. 在仓库根目录下执行 `yarn`。
3. 如果你修复了 bug 或者添加了代码, 而这些内容需要测试, 请添加测试 !
4. 确保通过测试套件 (`yarn test`)。提示: 开发环境下, `yarn test --watch TestName` 很有用。
5. 生产环境下, 执行 `yarn test --prod` 来进行测试。

6. 如果需要调试, 请执行 `yarn debug-test --watch TestName`, 打开 `chrome://inspect`, 之后再打开“审查”。
7. 使用 `prettier` (`yarn prettier`) 来格式化代码。
8. 确保 `lint` 校验代码 (`yarn lint`)。提示: 执行 `yarn linc` 去只检查更改过的文件。
9. 运行 `Flow` 来类型检查 (`yarn flow`)。
10. 请签订贡献者许可证协议 (Contributor License Agreement)。

贡献者许可证协议

为了让你的 pull request 得到接受, 你得提交贡献者许可证协议。你只需提交该协议一次, 所以如果你曾经对另一个 Facebook 开源项目提交过, 那么你已经准备好了。如果你是第一次提交 pull request, 请让我们得知你已提交协议, 这样我们能多方核对你的 GitHub 用户名。

[请在这里签订贡献者许可证协议](#)

必要条件

- 你需要安装 `Node` 的 LTS 版本和 `Yarn v1.2.0+`。
- 已安装 `JDK`。
- 你已安装 `gcc` (或者你在有必要安装编译器的情况下也不觉得麻烦), 因为一些依赖可能得经过编译, 而在 OS X, `Xcode` 命令行工具会帮你处理; 在 Ubuntu, `apt-get install build-essential` 会安装所需的 package (其它 Linux 发行版的类似命令也有效); 在 Windows 上得做些额外步骤, 请参考 `node-gyp` [安装步骤](#)。
- 熟悉 `Git`。

开发工作流程

克隆 React 项目后, 执行 `yarn` 来获取依赖。之后, 你可以执行以下命令:

- `yarn lint` 检查代码风格。
- `yarn linc` 和 `yarn lint` 差不多, 但是运行地更快, 因为只检查了分支中的不同文件。
- `yarn test` 运行完整的测试套装。
- `yarn test --watch` 运行交互式的测试监听器。
- `yarn test --prod` 在生产环境下运行测试。
- `yarn test <pattern>` 匹配文件名, 运行响应测试。
- `yarn debug-test` 和 `yarn test` 差不多, 不过多了个调试器, 你可以打开 `chrome://inspect` 并审查。
- `yarn flow` 运行 `Flow` 进行类型检查。
- `yarn build` 新建涉及所有包的 `build` 文件夹。
- `yarn build react/index,react-dom/index --type=UMD` 生成只有 React 和 ReactDOM 的 UMD 版本。

我们建议运行 `yarn test` (或上述命令) 以确保你的代码没有引入回归, 不管怎样, 这有助于尝试你的 React 构建版本。

首先, 运行 `yarn build`, 这会于 `build` 文件夹中生成预先构建的 bundle, 还会于 `build/packages` 中生成 npm 包。

想测试你做出的更改的话, 最简单的方法就是运行 `yarn build react/index,react-dom/index --type=UMD`, 之后再打开 `fixtures/packaging/babel-standalone/dev.html`, 该文件已使用 `build` 文件夹内的 `react.development.js` 来搞定你的更改。

如果你想测试你对已有 React 项目做出的更改，你可以复制 `build/node_modules/react/umd/react.development.js` 和 `build/node_modules/react-dom/umd/react-dom.development.js` 或其它构建版本，放入你的应用中并使用这些构建版本而非稳定版。

如果你的项目用 npm，你可以从依赖中删去 `react` 和 `react-dom`，使用 `yarn link` 将其指向本地文件夹的 `build` 目录。请注意，**当请在构建时，传递 `--type=NODE`，而不是 `--type=UMD`。同时，你还需要构建 `scheduler` 的 package：

```
cd ~/path_to_your_react_clone/
yarn build react/index,react/jsx,react-dom/index,scheduler --type=NODE

cd build/node_modules/react
yarn link
cd build/node_modules/react-dom
yarn link

cd ~/path/to/your/project
yarn link react react-dom
```

每当你在项目文件夹下运行 `yarn build`，更新版本会出现在 `node_modules` 文件夹，之后可以重新构建项目来测试更改。

如果依然缺少某些 package（例如，可能在项目中使用到 `react-dom/server`），则应始终执行 `yarn build` 进行完整构建。请注意，不带选项运行 `yarn build` 会耗费很长时间。

我们仍要求：`pull request` 得包括新功能对应的单元测试。这样，我们能确保以后你的代码不出问题。

风格指南

我们使用自动化代码格式化软件 [Prettier](#)。对代码做出更改后，运行 `yarn prettier`。

之后，`linter` 会捕获代码中可能出现的多数问题。你可以运行 `yarn linc` 来检查代码风格状态。

不过，`linter` 也有不能搞定的一些风格。如果有些东西不确定，请查看 [Airbnb's Style Guide](#) 来指导自己。

求意见稿(RFC)

许多更改（包括修复 bug 和完善文档）会经过通常所用的 GitHub pull request 工作流程来评审。

不过有些更改较大，对此，我们要求这些更改得经过一番设计流程，并在核心团队中达成共识。

求意见稿让新特性的合并入库经过一致且受控的流程。你可以前往 [rfcs repository](#) 去贡献。

License

你贡献 React 的同时也就同意了你的贡献部分使用了 MIT 协议。

接下来做什么？

请阅读[下一部分](#)来学习代码库是如何组织的。

[Go to TOC](#)

实现说明

这一部分是关于 [stack reconciler](#) 的一些实现说明。

这部分比较具有技术性，需要对 React 公共 API，以及 React 是如何将其分为 core、renderer 和 reconciler 的具有较好的理解。如果你对源码库还不是很熟悉，请先阅读[源码总览](#)。

这部分还要求了解 [React 组件及其实例和元素之间的不同](#)。

stack reconciler 是在 React 15 以及更早的版本中被采用。它的源码位于 [src/renderers/shared/stack/reconciler](#)。

视频：从零开始构建 React

[Paul O'Shannessy](#) 讲解的[从零开始构建 React](#) 对本文档有较大的启发。

本文档和他的讲解都是对实际代码库的简化，所以你能通过熟悉它们来获得更好的理解。

概览

reconciler 本身没有公共的 API。像 React DOM 和 React Native 这样的 renderer 使用它来根据用户写的 React 组件来高效地更新用户界面。

挂载是递归过程

让我们考虑第一次挂载组件时：

```
const root = ReactDOM.createRoot(rootEl);
root.render(<App />);
```

`root.render` 把 `<App />` 传递给 reconciler。请记住，`<App />` 是一个 React 元素，也就是对要渲染的内容的描述。可以把它视为普通的对象：

```
console.log(<App />);
// { type: App, props: {} }
```

reconciler 检查 `App` 是一个类还是一个函数。

如果 `App` 是函数，那么 reconciler 会调用 `App(props)` 来获取渲染的元素。

如果 `App` 是类，那么 reconciler 会通过 `new App(props)` 来实例化 `App`，并调用生命周期方法 `componentWillMount()`，之后调用 `render()` 方法来获取渲染的元素。

无论哪种方式，reconciler 都会探悉 `App` 的内容并渲染。

这个过程是递归的。`App` 可能会渲染某个 `<Greeting />`，`Greeting` 可能会渲染某个 `<Button />`，以此类推。当它探悉各个组件渲染的元素时，reconciler 会通过用户定义的组件递归地“向下探索”。

通过以下伪代码想象一下这个过程：

```

function isClass(type) {
  // 类组件会有这个标识位
  return (
    Boolean(type.prototype) &&
    Boolean(type.prototype.isReactComponent)
  );
}

// 这个函数接受一个 React 元素 (例如: <App />)
// 并返回表示已挂载树的 DOM 或者 原生节点
function mount(element) {
  var type = element.type;
  var props = element.props;

  // 将通过 type 作为函数运行
  // 或创建实例并调用 render()
  // 返回渲染后的元素
  var renderedElement;
  if (isClass(type)) {
    // 类组件
    var publicInstance = new type(props);
    // 设置 props
    publicInstance.props = props;
    // 如果有生命周期方法就调用
    if (publicInstance.componentWillMount) {
      publicInstance.componentWillMount();
    }
    // 调用 render() 返回渲染后的元素
    renderedElement = publicInstance.render();
  } else {
    // 函数组件
    renderedElement = type(props);
  }

  // 这个过程是递归的
  // 因为组件可能会返回具体另一个组件类型的元素
  return mount(renderedElement);

  // 提示: 这个实现是不完整的并且无限递归!
  // 只处理像 <App /> 或者 <Button /> 的元素
  // 还不能处理像 <div /> 或者 <p /> 的元素
}

var rootEl = document.getElementById('root');
var node = mount(<App />);
rootEl.appendChild(node);

```

注意：

这其实是一份伪代码。它与真实的实现并不相似。因为我们还没有讨论该递归过程何时停止，所以它也会造成堆栈溢出。

让我们回顾上面例子中的一些关键的想法：

- React 元素是用来表示组件的类型（例如：`App`）和 `props` 的简单的对象。
- 用户定义的组件（例如：`App`）可以是类，也可以是函数，但是它们都“渲染产生”元素。
- “挂载”是一个递归的过程，根据特定的顶层 React 元素（e.g. `<App />`）产生 DOM 或 Native 树。

挂载宿主元素

如果我们没有渲染某些东西输出到电脑屏幕，这个过程将会是无用的。

除了用户定义的（“组合”）组件，React 元素也可能表示为平台专属（“宿主”）组件。例如，`Button` 可能会从 `render` 方法返回一个 `<div />`。

如果元素的 `type` 属性是字符串，我们处理的就是宿主元素：

```
console.log(<div />);
// { type: 'div', props: {} }
```

宿主元素中没有用户定义代码。

当 reconciler 遇到宿主元素时，它会让 renderer 负责挂载它。例如，React DOM 会创建一个 DOM 节点。

如果宿主元素拥有子元素，reconciler 会根据上文提到的算法对其进行递归地挂载。无论子元素是宿主（像 `<div><hr /><div>`），还是组合（像 `<div><Button /></div>`），两者都无所谓。

子组件生成的 DOM 节点会附加在父 DOM 节点上，递归地完成整个 DOM 结构的组装。

注意：

reconciler 本身不与 DOM 绑定。挂载的确切结果（在源代码中有时叫做“挂载映像”）取决于 renderer，可以是一个 DOM 节点（React DOM），一个字符串（React DOM Server），或是一个表示原生视图的数字（React Native）。

如果我们扩展代码去处理宿主元素，会是如下样子：

```
function isClass(type) {
  // 类组件会有这个标识位
  return (
    Boolean(type.prototype) &&
    Boolean(type.prototype.isReactComponent)
  );
}

// 此函数仅处理组合类型的元素
// 例如，处理 <App /> 和 <Button />，但不处理 <div />
function mountComposite(element) {
  var type = element.type;
  var props = element.props;

  var renderedElement;
  if (isClass(type)) {
    // 类组件
    var publicInstance = new type(props);
    // 设置 props
    publicInstance.props = props;
    // 如果有生命周期方法就调用
    if (publicInstance.componentWillMount) {
      publicInstance.componentWillMount();
    }
  }
}
```

```

        renderedElement = publicInstance.render();
    } else if (typeof type === 'function') {
        // 函数组件
        renderedElement = type(props);
    }

    // 这是递归的，但是当元素是宿主(例如: <div />)而不是组合(例如 <App />)时,
    // 我们最终会到达递归的底部:
    return mount(renderedElement);
}

// 此函数只处理宿主类型的元素
// 例如: 处理 <div /> 和 <p />, 但不处理 <App />.
function mountHost(element) {
    var type = element.type;
    var props = element.props;
    var children = props.children || [];
    if (!Array.isArray(children)) {
        children = [children];
    }
    children = children.filter(Boolean);

    // 这段代码不应该出现在 reconciler。
    // 不同的 renderer 可能会以不同方式初始化节点。
    // 例如, React Native 会创建 iOS 或 Android 的视图。
    var node = document.createElement(type);
    Object.keys(props).forEach(propName => {
        if (propName !== 'children') {
            node.setAttribute(propName, props[propName]);
        }
    });
}

// 挂载子元素
children.forEach(childElement => {
    // 子元素可能是宿主(例如: <div />)或者组合 (例如: <Button />).
    // 我们还是递归挂载他们
    var childNode = mount(childElement);

    // 这一行代码也是特殊的 renderer。
    // 根据 renderer 不同, 方式也不同:
    node.appendChild(childNode);
});

// DOM 节点作为挂载的结果返回。
// 这是递归结束的位置。
return node;
}

function mount(element) {
    var type = element.type;
    if (typeof type === 'function') {
        // 用户定义组件
        return mountComposite(element);
    } else if (typeof type === 'string') {
        // 平台特定组件
        return mountHost(element);
    }
}

var rootEl = document.getElementById('root');
var node = mount(<App />);
rootEl.appendChild(node);

```

以上代码是可以运作的，但是与 reconciler 的实际实现依然相差很远。关键的缺失部分是对更新的支持。

引入内部实例

React 的关键特点是你可以重新渲染所有内容，并且不会重新生成 DOM 或重置 state：

```
root.render(<App />);
// 应该重用已经存在的 DOM:
root.render(<App />);
```

然而，之前的实现只是知道如何挂载最初的树。由于它没有储存所有的必要信息，例如所有的 `publicInstance`，或 DOM 节点属于哪个组件，所以它不能完成更新操作。

stack reconciler 源码通过把 `mount()` 函数作为一个类的方法来解决这个问题。这种方法是存在缺点的，所以我们正朝着与之相对的方向[进行 reconciler 的重写工作](#)。不过这就是它现在的运作方式。

我们会创建两个类：`DOMComponent` 和 `CompositeComponent`，而不是分离的两个函数 `mountHost` 和 `mountComposite`。

两个类都有一个接受 `element` 的构造函数，同时也有一个返回挂载后节点的 `mount()` 方法。我们用一个可以实例化正确类的工厂函数替换了顶层的 `mount()` 函数：

```
function instantiateComponent(element) {
  var type = element.type;
  if (typeof type === 'function') {
    // 用户定义组件
    return new CompositeComponent(element);
  } else if (typeof type === 'string') {
    // 平台特定组件
    return new DOMComponent(element);
  }
}
```

首先，让我们思考一下 `CompositeComponent` 的实现：

```
class CompositeComponent {
  constructor(element) {
    this.currentElement = element;
    this.renderedComponent = null;
    this.publicInstance = null;
  }

  getPublicInstance() {
    // 对于组合组件，公共类实例
    return this.publicInstance;
  }

  mount() {
    var element = this.currentElement;
    var type = element.type;
    var props = element.props;

    var publicInstance;
    var renderedElement;
    if (isClass(type)) {
      // 组件类
    }
  }
}
```

```

publicInstance = new type(props);
// 设置 props
publicInstance.props = props;
// 如果有生命周期方法就调用
if (publicInstance.componentWillMount) {
  publicInstance.componentWillMount();
}
renderedElement = publicInstance.render();
} else if (typeof type === 'function') {
// 函数组件
publicInstance = null;
renderedElement = type(props);
}

// 保存公共实例
this.publicInstance = publicInstance;

// 根据元素实例化子内部实例。
// <div /> 或者 <p /> 是 DOMComponent,
// 而 <App /> 或者 <Button /> 是 CompositeComponent。
var renderedComponent = instantiateComponent(renderedElement);
this.renderedComponent = renderedComponent;

// 挂载渲染后的输出
return renderedComponent.mount();
}
}

```

这与之前的 `mountComposite()` 的实现没有太多的不同，但是现在我们可以保存一些信息，如 `this.currentElement`, `this.renderedComponent` 和 `this.publicInstance`，用于更新期间使用。

需要注意的是 `CompositeComponent` 的实例与用户提供的 `element.type` 的实例是不同的东西。`CompositeComponent` 是我们的 reconciler 的实现细节，并且永远不会暴露给用户。用户定义的类是从 `element.type` 读取的，并且 `CompositeComponent` 会创建一个它的实例。

为了避免混淆，我们把 `CompositeComponent` 和 `DOMComponent` 的实例叫做“内部实例”。由于它们的存在，我们可以把一些长时间存在的数据存入其中。只有 renderer 和 reconciler 能意识到它们的存在。

相反，我们把用户定义的类的实例叫做“公共实例”。公共实例就是你在 `render()` 中所见到的 `this` 和你的自定义组件中的一些其他方法。

`mountHost()` 函数，重构为 `DOMComponent` 类的 `mount()` 方法，看起来也很熟悉：

```

class DOMComponent {
  constructor(element) {
    this.currentElement = element;
    this.renderedChildren = [];
    this.node = null;
  }

  getPublicInstance() {
    // 对于 DOM 组件，只公共 DOM 节点
    return this.node;
  }

  mount() {
    var element = this.currentElement;
    var type = element.type;
  }
}

```

```

var props = element.props;
var children = props.children || [];
if (!Array.isArray(children)) {
  children = [children];
}

// 创建并保存节点
var node = document.createElement(type);
this.node = node;

// 设置属性
Object.keys(props).forEach(propName => {
  if (propName !== 'children') {
    node.setAttribute(propName, props[propName]);
  }
});

// 创建并保存包含的子项
// 他们每个都可以是 DOMComponent 或者是 CompositeComponent,
// 取决于类型是字符串还是函数
var renderedChildren = children.map(instantiateComponent);
this.renderedChildren = renderedChildren;

// 收集他们在 mount 上返回的节点
var childNodes = renderedChildren.map(child => child.mount());
childNodes.forEach(childNode => node.appendChild(childNode));

// DOM 节点作为挂载结果返回
return node;
}
}

```

`mountHost()` 重构后主要的区别是我们保存了与内部 DOM 组件实例关联的 `this.node` 和 `this.renderedChildren`。在将来我们还使用他们来进行非破坏性更新。

因此，每个内部实例，组合或者宿主，现在都指向了它的子内部实例。为帮你更直观的了解，假设有函数组件 `<App>` 会渲染类组件 `<Button>`，并且 `Button` 渲染一个 `<div>`，其内部实例树将如下所示：

```

[object CompositeComponent] {
  currentElement: <App />,
  publicInstance: null,
  renderedComponent: [object CompositeComponent] {
    currentElement: <Button />,
    publicInstance: [object Button],
    renderedComponent: [object DOMComponent] {
      currentElement: <div />,
      node: [object HTMLDivElement],
      renderedChildren: []
    }
  }
}

```

在 DOM 中，你只能看到 `<div>`。但是在内部实例树包含了组合和宿主的内部实例。

组合内部实例需要存储：

- 当前元素。
- 如果元素的类型是类的公共实例

- 单次渲染后的内部实例。它可以是 `DOMComponent` 或 `CompositeComponent`。

宿主内部实例需要存储：

- 当前元素。
- DOM 节点。
- 所有子内部实例。它们中的每一个都可以是 `DOMComponent` 或 `CompositeComponent`。

如果你难以想象内部实例树在较为复杂的应用程序中的结构，[React DevTools](#) 可以给你一个相似的结果，因为它突出呈现了灰色的宿主实例，以及紫色的组合实例。

```
<TodoApp>
  ▼<div>
    <h3>TODO</h3>
    ▼<TodoList items=[...]>
      ▼<ul>
        <li key="1475260226151">Buy milk</li>
      </ul>
    </TodoList>
    ▼<form onSubmit=bound handleSubmit()>
      <input onChange=bound onChange() value="" />
      <button>Add #2</button>
    </form>
  </div>
</TodoApp>
```

为了完成重构，我们将引入一个函数，它将完整的树挂载到容器节点中并返回公共实例：

```
function mountTree(element, containerNode) {
  // 创建顶层内部实例
  var rootComponent = instantiateComponent(element);

  // 挂载顶层组件到容器中
  var node = rootComponent.mount();
  containerNode.appendChild(node);

  // 返回它提供的公共实例
  var publicInstance = rootComponent.getPublicInstance();
  return publicInstance;
}

var rootEl = document.getElementById('root');
mountTree(<App />, rootEl);
```

卸载

现在，我们有内部实例，以保留其子节点和 DOM 节点，我们可以实现卸载。对于组合组件，卸载调用生命周期方法和递归。

```

class CompositeComponent {
  // ...

  unmount() {
    // 如果有生命周期方法就调用
    var publicInstance = this.publicInstance;
    if (publicInstance) {
      if (publicInstance.componentWillUnmount) {
        publicInstance.componentWillUnmount();
      }
    }

    // 卸载单个渲染的组件
    var renderedComponent = this.renderedComponent;
    renderedComponent.unmount();
  }
}

```

对于 `DOMComponent`，会告诉每一个子项去卸载

```

class DOMComponent {
  // ...

  unmount() {
    // 卸载所有的子项
    var renderedChildren = this.renderedChildren;
    renderedChildren.forEach(child => child.unmount());
  }
}

```

在实践中，卸载 DOM 组件也需要删除事件侦听器和清除一些缓存，但我们将跳过这些细节。

我们现在可以添加一个叫 `unmountTree(containerNode)` 顶层函数，该函数类似于 `ReactDOM.unmountComponentAtNode()`。

```

function unmountTree(containerNode) {
  // 从 DOM 节点读取内部实例：
  // (这还不起作用，我们需要更改 mountTreeTree() 来存储它。)
  var node = containerNode.firstChild;
  var rootComponent = node._internalInstance;

  // 卸载树并清空容器
  rootComponent.unmount();
  containerNode.innerHTML = '';
}

```

为了使其工作，我们需要从 DOM 节点读取内部根实例。我们将修改 `mountTree()` 为其增加 `_internalInstance` 属性来添加 DOM 根节点，我们还将在 `mountTree()` 中实现销毁任何现有的树的功能，以便它可以被多次调用：

```

function mountTree(element, containerNode) {
  // 销毁所有现有的树
  if (containerNode.firstChild) {
    unmountTree(containerNode);
  }

  // 创建顶层的内部实例
}

```

```

var rootComponent = instantiateComponent(element);

// 挂载顶层组件到容器中
var node = rootComponent.mount();
containerNode.appendChild(node);

// 保存对内部实例的引用
node._internalInstance = rootComponent;

// 返回它提供的公共实例
var publicInstance = rootComponent.getPublicInstance();
return publicInstance;
}

```

现在，运行 `unmountTree()` 或重复运行 `mountTree()`，都会删除旧树并在组件上运行 `componentWillUnmount()` 生命周期方法。

更新

在上一个章节，我们实现了卸载。但是，如果每个 prop 更改都卸载整棵树，并重新挂载，那么 react 就不再高效了。reconciler 的目标是尽可能复用现有实例来保留 DOM 和状态：

```

var rootEl = document.getElementById('root');

mountTree(<App />, rootEl);
// 应该复用已经存在的 DOM:
mountTree(<App />, rootEl);

```

我们将用一种方法扩展内部实例。除了 `mount()` 和 `unmount()` 之外，`DOMComponent` 和 `CompositeComponent` 都将实现一个名为 `receive(nextElement)` 的新方法：

```

class CompositeComponent {
  // ...

  receive(nextElement) {
    // ...
  }
}

class DOMComponent {
  // ...

  receive(nextElement) {
    // ...
  }
}

```

它的工作是尽一切可能使组件（及其任何子组件）与 `nextElement` 提供的描述一起更新。

这通常被称为 "virtual DOM diffing" 的部分，但实际发生的情况是，我们递归遍历内部树，让每个内部实例接收更新。

更新组合组件

当一个组合组件接收一个新的元素时，我们将运行生命周期方法 `componentWillUpdate()`。

然后我们使用新的 prop 重新渲染组件，并获取下一次渲染的元素：

```

class CompositeComponent {
  // ...

  receive(nextElement) {
    var prevProps = this.currentElement.props;
    var publicInstance = this.publicInstance;
    var prevRenderedComponent = this.renderedComponent;
    var prevRenderedElement = prevRenderedComponent.currentElement;

    // 更新自己的元素
    this.currentElement = nextElement;
    var type = nextElement.type;
    var nextProps = nextElement.props;

    // 找下一次 render() 输出的是什么
    var nextRenderedElement;
    if (isClass(type)) {
      // 类组件
      // 如果有生命周期方法就调用
      if (publicInstance.componentWillUpdate) {
        publicInstance.componentWillUpdate(nextProps);
      }
      // 更新 props
      publicInstance.props = nextProps;
      // 重新渲染
      nextRenderedElement = publicInstance.render();
    } else if (typeof type === 'function') {
      // 函数组件
      nextRenderedElement = type(nextProps);
    }
  }
  // ...
}

```

接下来，我们可以看一下渲染元素的 `type`。如果 `type` 自上次渲染后没有改变，之后的组件也可以就地更新。

例如，如果第一次返回 `<Button color="red" />`，第二次返回 `<Button color="blue" />`，我们可以只告诉相应的内部实例 `receive()` 下一个元素：

```

// ...

// 如果渲染元素的 type 没有更改,
// 重用已经存在组件实例并退出。
if (prevRenderedElement.type === nextRenderedElement.type) {
  prevRenderedComponent.receive(nextRenderedElement);
  return;
}

// ...

```

但是，如果下一个渲染元素的 `type` 与先前渲染的元素不同，则无法更新内部实例。`<button>` 不能“变成”`<input>`。

相反，我们必须卸载现有的内部实例，然后挂载并渲染元素 `type` 对应的新实例。例如，当先前渲染 `<button />` 的组件再渲染 `<input />` 时，会发生这种情况：

```

// ...

// 如果我们达到这里，我们需要卸载以前挂载的组件。
// 挂载新的组件，并交换其节点。

```

```
// 查找旧节点，因为需要替换它
var prevNode = prevRenderedComponent.getHostNode();

// 卸载旧的子组件并挂载新的子组件
prevRenderedComponent.unmount();
var nextRenderedComponent = instantiateComponent(nextRenderedElement);
var nextNode = nextRenderedComponent.mount();

// 替换子组件的引用
this.renderedComponent = nextRenderedComponent;

// 将旧节点替换为新节点
// 注意：这是 renderer 特定的代码，
// 理想情况下应位于 CompositeComponent 之外：
prevNode.parentNode.replaceChild(nextNode, prevNode);
}

}
```

综上所述，当组合组件收到新元素时，它可以将更新委派给其渲染的内部实例，或者卸载它并在其位置挂载新元素。

还有另一种情况，组件将重新挂载而非接收元素，即元素的 `key` 已更改。在当前文档中，我们不讨论 `key` 处理，因为它增加了复杂教程的复杂性。

请注意，我们需要向内部实例添加名为 `getHostNode()` 的方法，以便可以在更新期间找到平台特定的节点并替换它。对于两个类，其实现都非常简单：

```
class CompositeComponent {
  // ...

  getHostNode() {
    // 要求渲染组件提供它。
    // 递归深入任意组合组件。
    return this.renderedComponent.getHostNode();
  }
}

class DOMComponent {
  // ...

  getHostNode() {
    return this.node;
  }
}
```

更新宿主组件

宿主组件实现，如 `DOMComponent`，更新方式不同。当他们收到一个元素时，他们需要更新平台特定的视图。在 React DOM 的情况下，这意味着更新 DOM 属性：

```
class DOMComponent {
  // ...

  receive(nextElement) {
    var node = this.node;
    var prevElement = this.currentElement;
    var prevProps = prevElement.props;
    var nextProps = nextElement.props;
    this.currentElement = nextElement;
```

```
// 删除旧的属性
Object.keys(prevProps).forEach(propName => {
  if (propName !== 'children' && !nextProps.hasOwnProperty(propName)) {
    node.removeAttribute(propName);
  }
});
// 设置新的属性
Object.keys(nextProps).forEach(propName => {
  if (propName !== 'children') {
    node.setAttribute(propName, nextProps[propName]);
  }
});
// ...

```

然后宿主组件需要更新其子组件。与组合组件不同，它们可能包含多个子组件。

在此简化的示例中，我们使用内部实例数组并遍历它，根据接收的 `type` 是否与以前的 `type` 匹配更新或替换内部实例。真正的 reconciler 还会在描述中获取元素的 `key`，并存储和跟踪除了插入和删除之外的移动，但我们这里将省略此逻辑。

我们在列表中收集子组件的 DOM 操作，以便可以批量执行它们：

```
// ...
// 这些是 React 元素的数组：
var prevChildren = prevProps.children || [];
if (!Array.isArray(prevChildren)) {
  prevChildren = [prevChildren];
}
var nextChildren = nextProps.children || [];
if (!Array.isArray(nextChildren)) {
  nextChildren = [nextChildren];
}
// 这些是内部实例的数组：
var prevRenderedChildren = this.renderedChildren;
var nextRenderedChildren = [];

// 当我们迭代子组件时，我们将向数组添加相应操作。
var operationQueue = [];

// 注意：以下部分非常简化！
// 它不处理重新排序、带空洞或有 key 的子组件。
// 它的存在只是为了说明整个流程，而不是细节。

for (var i = 0; i < nextChildren.length; i++) {
  // 尝试去获取此子组件现有的内部实例
  var prevChild = prevRenderedChildren[i];

  // 如果此索引下没有内部实例，
  // 则子实例已追加到末尾。
  // 创建新的内部实例，挂载它，并使用其节点。
  if (!prevChild) {
    var nextChild = instantiateComponent(nextChildren[i]);
    var node = nextChild.mount();

    // 记录我们需要追加的节点
    operationQueue.push({type: 'ADD', node});
    nextRenderedChildren.push(nextChild);
    continue;
}
}
```

```

// 仅当实例的元素类型匹配时，我们才能更新该实例。
// 例如，<Button size="small" /> 可以更新成 <Button size="large" />,
// 但是不能更新成 <App />。
var canUpdate = prevChildren[i].type === nextChildren[i].type;

// 如果我们无法更新现有的实例，
// 我们必须卸载它并安装一个新实例去替代
if (!canUpdate) {
  var prevNode = prevChild.getHostNode();
  prevChild.unmount();

  var nextChild = instantiateComponent(nextChildren[i]);
  var nextNode = nextChild.mount();

  // 记录我们需要替换的节点
  operationQueue.push({type: 'REPLACE', prevNode, nextNode});
  nextRenderedChildren.push(nextChild);
  continue;
}

// 如果我们能更新现有的内部实例，
// 只是让它接收下一个元素并处理自己的更新。
prevChild.receive(nextChildren[i]);
nextRenderedChildren.push(prevChild);
}

// 最后，卸载不存在的任何子组件：
for (var j = nextChildren.length; j < prevChildren.length; j++) {
  var prevChild = prevRenderedChildren[j];
  var node = prevChild.getHostNode();
  prevChild.unmount();

  // 记录我们需要删除的节点
  operationQueue.push({type: 'REMOVE', node});
}

// 将渲染的子级列表指向更新的版本。
this.renderedChildren = nextRenderedChildren;

// ...

```

最后一步，我们执行 DOM 操作。同样，真正的 reconciler 代码更为复杂，因为它还处理移动操作：

```

// ...

// 处理操作队列。
while (operationQueue.length > 0) {
  var operation = operationQueue.shift();
  switch (operation.type) {
    case 'ADD':
      this.node.appendChild(operation.node);
      break;
    case 'REPLACE':
      this.node.replaceChild(operation.nextNode, operation.prevNode);
      break;
    case 'REMOVE':
      this.node.removeChild(operation.node);
      break;
  }
}

```

}{}

这就是更新宿主组件。

顶层更新

现在，`CompositeComponent` 和 `DOMComponent` 都实现了 `receive(nextElement)` 方法，我们可以更改顶级的 `mountTree()` 函数，以便当元素的 `type` 与上次相同时使用它：

```
function mountTree(element, containerNode) {
  // 检查现有的树
  if (containerNode.firstChild) {
    var prevNode = containerNode.firstChild;
    var prevRootComponent = prevNode._internalInstance;
    var prevElement = prevRootComponent.currentElement;

    // 如果可以，重用现有的根组件
    if (prevElement.type === element.type) {
      prevRootComponent.receive(element);
      return;
    }

    // 否则，卸载现有树
    unmountTree(containerNode);
  }
  // ...
}
```

现在调用 `mountTree()` 两次相同的 `type` 是没有破坏性的

```
var rootEl = document.getElementById('root');

mountTree(<App />, rootEl);
// 应该重用已经存在的 DOM:
mountTree(<App />, rootEl);
```

这些都是 React 内部工作原理的基础知识。

我们遗漏了什么

与真实代码库相比，本文档得到了简化。我们没有解决几个重要方面：

- 组件可以呈现 `null`, reconciler 可以处理在数组中“空插槽”和渲染输出。
 - reconciler 还从元素中读取 `key`, 并使用它来确定哪个内部实例对应于数组中的哪个元素。实际 React 实现中的大部分复杂性与此相关。
 - 除了组合和宿主内部实例类外, 还有用于“文本”和“空”组件的类。它们表示文本节点和通过渲染 `null` 获得“空插槽”。
 - renderer 使用 [注入](#) 的方式将宿主内部类传递给 reconciler. 例如, React DOM 告诉 reconciler 使用 `ReactDOMComponent` 作为宿主内部实例实现。

- 更新子列表的逻辑被提取到一个名为 `ReactMultiChild` 的 mixin 中，它由 React DOM 和 React Native 中的宿主内部实例类实现使用。
- `reconciler` 还在组合组件中实现对 `setState()` 的支持。事件处理程序内的多个更新将被批处理为单一更新。
- `reconciler` 还负责将 `refs` 附加和分离到组合组件和宿主节点。
- 在 DOM 准备好之后调用的生命周期方法，例如 `componentDidMount()` 和 `componentDidUpdate()`，被收集到“回调队列”中并在一个批处理中执行。
- React 将有关当前更新的信息放入名为“transaction”的内部对象中。事务可用于跟踪挂起的生命周期方法的队列、警告的当前 DOM 嵌套以及特定更新的“全局”任何其他内容。事务还确保在更新后“清理所有内容”。例如，React DOM 提供的事务类在任何更新后还原 input selection。

跳转到代码

- `ReactMount` 就像本教程中 `mountTree()` 和 `unmountTree()` 这样的代码。它负责挂载和卸载顶层组件。`ReactNativeMount` 是 React Native 的模拟。
- `ReactDOMComponent` 相当于本教程中的 `DOMComponent`。它实现了 React DOM renderer 的宿主组件类。`ReactNativeBaseComponent` 是 React Native 的模拟。
- `ReactCompositeComponent` 相当于本教程中的 `CompositeComponent`。它处理调用用户定义的组件并维护其状态。
- `instantiateReactComponent` 包含选择要为元素构造的正确内部实例类的开关。它相当于本教程中的 `instantiateComponent()`。
- `ReactReconciler` 是一个包含 `mountComponent()`、`receiveComponent()` 和 `unmountComponent()` 方法的包装器。它调用内部实例上的底层实现，但也包括一些由所有内部实例实现共享的代码。
- `ReactChildReconciler` 实现根据子元素的 `key` 挂载、更新和卸载子级的逻辑。
- `ReactMultiChild` 实现对子组件插入、删除和移动操作队列的处理，独立于渲染器。
- 由于遗留原因，`mount()`、`receive()` 和 `unmount()` 在 React 代码库中实际上名字为 `mountComponent()`、`receiveComponent()` 和 `unmountComponent()`，但是它们接收元素。
- 内部实例的属性以下划线开头，例如，`_currentElement`。它们被认为是整个代码库中的只读公共字段。

未来方向

stack reconciler 具有固有的局限性，例如同步并且无法中断工作或将其拆分为块。新的 Fiber reconciler 正在进行中，具有完全不同的架构。在未来，我们打算用它替换 stack reconciler，但目前它还远远没有达到功能对等。

下一步

阅读[下一节](#) 了解我们用于开发 React 的指导原则。

[Go to TOC](#)

与第三方库协同

React 可以被用于任何 web 应用中。它可以被嵌入到其他应用，且需要注意，其他的应用也可以被嵌入到 React。本指南将介绍一些更常见的用例，专注于与 [jQuery](#) 和 [Backbone](#) 进行整合，同样的思路还可以应用于将组件与任意现有代码集成。

集成带有 DOM 操作的插件

React 不会理会 React 自身之外的 DOM 操作。它根据内部虚拟 DOM 来决定是否需要更新，而且如果同一个 DOM 节点被另一个库操作了，React 会觉得困惑而且没有办法恢复。

这并不意味着 React 与其他操作 DOM 的方式不能结合，也不一定结合困难，只不过需要你去关注每个库做的事情。

避免冲突的最简单方式就是防止 React 组件更新。你可以渲染无需更新的 React 元素，比如一个空的 `<div />`。

如何解决这个问题

为了证明这一点，我来草拟一个用于通用 jQuery 插件的 wrapper

我们会添加一个 `ref` 到这个根 DOM 元素。在 `componentDidMount` 中，我们能够获取它的引用这样我们就可以把它传递给 jQuery 插件了。

为了防止 React 在挂载之后去触碰这个 DOM，我们会从 `render()` 函数返回一个空的 `<div />`。这个 `<div />` 元素既没有属性也没有子元素，所以 React 没有理由去更新它，使得 jQuery 插件可以自由的管理这部分的 DOM：

```
class SomePlugin extends React.Component {
  componentDidMount() {
    this.$el = $(this.el);
    this.$el.somePlugin();
  }

  componentWillUnmount() {
    this.$el.somePlugin('destroy');
  }

  render() {
    return <div ref={el => this.el = el} />;
  }
}
```

注意我们同时定义了 `componentDidMount` 和 `componentWillUnmount` 生命周期函数。许多 jQuery 插件绑定事件监听到 DOM 上，所以在 `componentWillUnmount` 中注销监听是很重要的。如果这个插件没有提供一个用于清理的方法，你很可能会需要自己来提供一个，为了避免内存泄漏要记得把所有插件注册的监听都移除掉。

集成 jQuery Chosen 插件

对于应用这些概念的更具体的一个例子，我们给这个用于增强 `<select>` 输入的 [Chosen](#) 插件写一个最小的 wrapper。

注意：

仅仅是因为可能，但这并不意味着这是构建 React 应用的最佳方式。我们鼓励大家尽可能的使用 React 组件。组件在 React 应用中更易于复用，并且在大多数情况下能更好地控制其行为和显示。

首先，我们来看下 Chosen 对 DOM 做了哪些操作

如果你在一个 `<select>` DOM 节点上调用了它，它会读取原 DOM 节点的属性，使用行内样式隐藏它，然后紧挨着这个 `<select>` 之后增加一个独立的具有它自身显示表现的 DOM 节点。然后它会在值变化的时候触发 jQuery 事件来通知我们这些变化。

以下代码是我们最终要实现的效果：

```
function Example() {
  return (
    <Chosen onChange={value => console.log(value)}>
      <option>vanilla</option>
      <option>chocolate</option>
      <option>strawberry</option>
    </Chosen>
  );
}
```

为了简化，我们将它实现为 [uncontrolled component](#)

首先，我会创建一个空的组件，它的 `render()` 函数我们返回一个包含 `<select>` 的 `<div>`：

```
class Chosen extends React.Component {
  render() {
    return (
      <div>
        <select className="Chosen-select" ref={el => this.el = el}>
          {this.props.children}
        </select>
      </div>
    );
  }
}
```

注意我们为什么要把 `<select>` 使用一个额外的 `<div>` 包裹起来。这是很必要的，因为 Chosen 会紧挨着我们传递给它的 `<select>` 节点追加另一个 DOM 元素。然而，对于 React 来说 `<div>` 总是只有一个子节点。这样我们就能确保 React 更新不会和 Chosen 追加的额外 DOM 节点发生冲突。在 React 工作流之外修改 DOM 是非常重大的事情，你必须确保 React 没有理由去触碰那些节点。

接下来，我们会实现生命周期函数。我们需要在 `componentDidMount` 中使用 `<select>` 的引用初始化 Chosen，并且在 `componentWillUnmount` 中将其销毁：

```
componentDidMount() {
  this.$el = $(this.el);
  this.$el.chosen();
}

componentWillUnmount() {
  this.$el.chosen('destroy');
}
```

在 CodePen 上运行

注意 React 不会给 `this.el` 字段赋予特殊的含义。它能够工作只是因为我们之前在 `render()` 函数中把一个 `ref` 赋值给了这个字段：

```
<select className="Chosen-select" ref={el => this.el = el}>
```

到此已经足够让我们的组件去渲染了，但我们同时希望在值变化的时候被通知到。要做到这点，我们需要在订阅由 Chosen 管理的 `<select>` 上的 jQuery `change` 事件。

我们不直接把 `this.props.onChange` 传递给 Chosen 是因为组件的 props 可能随时变化，并且这也包括事件处理函数。对应的，我们会定义一个 `handleChange()` 方法来调用 `this.props.onChange`，并且订阅 jQuery 的 `change` 事件：

```
componentDidMount() {
  this.$el = $(this.el);
  this.$el.chosen();

  this.handleChange = this.handleChange.bind(this);
  this.$el.on('change', this.handleChange);
}

componentWillUnmount() {
  this.$el.off('change', this.handleChange);
  this.$el.chosen('destroy');
}

handleChange(e) {
  this.props.onChange(e.target.value);
}
```

在 CodePen 上运行

最后，还剩下一件事情需要处理。在 React 中，props 可以在不同的时间有不同的值。例如，如果父组件的状态发生变化 `<Chosen>` 组件可能得到不同的 `children`。这意味着从集成的角度来看，我们因应 prop 的更新而手动更新 DOM 这一点是非常重要的，因为我们已经不再使用 React 来帮我们管理 DOM 了。

Chosen 的文档建议我们使用 jQuery `trigger()` API 来通知原始 DOM 元素这些变化。我们会让 React 来管理在 `<select>` 中 `this.props.children` 的更新，但是我们同样需要增加一个 `componentDidUpdate()` 生命周期函数来通知 Chosen 关于 `children` 列表的变化：

```
componentDidUpdate(prevProps) {
  if (prevProps.children !== this.props.children) {
    this.$el.trigger("chosen:updated");
  }
}
```

通过这种方法，当由 React 管理的 `<select>` `children` 改变时，Chosen 会知道如何更新它的 DOM 元素。

Chosen 组件的完整实现看起来是这样的：

```
class Chosen extends React.Component {
  componentDidMount() {
    this.$el = $(this.el);
```

```

    this.$el.chosen();

    this.handleChange = this.handleChange.bind(this);
    this.$el.on('change', this.handleChange);
}

componentDidUpdate(prevProps) {
  if (prevProps.children !== this.props.children) {
    this.$el.trigger("chosen:updated");
  }
}

componentWillUnmount() {
  this.$el.off('change', this.handleChange);
  this.$el.chosen('destroy');
}

handleChange(e) {
  this.props.onChange(e.target.value);
}

render() {
  return (
    <div>
      <select className="Chosen-select" ref={el => this.el = el}>
        {this.props.children}
      </select>
    </div>
  );
}
}

```

[在 CodePen 上运行](#)

和其他视图库集成

得益于 `createRoot()` 的灵活性 React 可以被嵌入到其他的应用中。

虽然 React 通常被用来在启动的时候加载一个单独的根 React 组件到 DOM 上，`createRoot()` 同样可以在 UI 的独立部分上多次调用，这些部分可以小到一个按钮，也可以大到一个应用。

事实上，这正是 Facebook 如何使用 React 的。这让我们小块小块地在应用中使用 React，并且把他们结合到我们现存的服务端产生的模板和其他客户端代码中。

利用 React 替换基于字符串的渲染

在旧的 web 应用中一个通用的模式就是使用一个字符串描述 DOM 块并且通过类似 `$el.html(htmlString)` 这样的方式插入到 DOM 中。代码库中的这种例子是非常适合引入 React 的。直接把基于字符串的渲染重写成 React 组件即可。

那么下面这段 jQuery 的实现...

```

$('#container').html('<button id="btn">Say Hello</button>');
$('#btn').click(function() {
  alert('Hello!');
});

```

...可以使用 React 组件重写为：

```
function Button() {
  return <button id="btn">Say Hello</button>;
}

$('#btn').click(function() {
  alert('Hello!');
});
```

从这起你可开始以把更多的逻辑移动到组件中，并且开始应用更多通用 React 实践。例如，在组件中最好不要依赖 ID 因为同一个组件可能会被渲染多次。相反的，我们会使用 [React 事件系统](#) 并且直接注册 click 处理函数到 React `<button>` 元素：

```
function Button(props) {
  return <button onClick={props.onClick}>Say Hello</button>;
}

function HelloButton() {
  function handleClick() {
    alert('Hello!');
  }
  return <Button onClick={handleClick} />;
}
```

[在 CodePen 上运行](#)

只要你喜欢你可以有不限数量的这种独立组件，并且使用 `ReactDOM.createRoot()` 把他们渲染到不同的容器中。逐渐的，随着你把越来越多的应用转换到 React，你就可以把它们结合成更大的组件，并且把 `ReactDOM.createRoot()` 的调用移动到更上层的结构。

把 React 嵌入到 Backbone 视图

Backbone 视图通常使用 HTML 字符串，或者产生字符串的模板函数，来创建 DOM 元素的内容。这个过程，同样的，可以通过渲染一个 React 组件来替换掉。

如下，我们会创建一个名为 `ParagraphView` 的 Backbone 视图。他会重载 Backbone 的 `render()` 函数来渲染一个 React `<Paragraph>` 组件到 Backbone (`this.el`) 提供的 DOM 元素中。这里，同样的，我们将会使用 `ReactDOM.createRoot()`：

```
function Paragraph(props) {
  return <p>{props.text}</p>;
}

const ParagraphView = Backbone.View.extend({
  initialize(options) {
    this.reactRoot = ReactDOM.createRoot(this.el);
  },
  render() {
    const text = this.model.get('text');
    this.reactRoot.render(<Paragraph text={text} />);
    return this;
  },
  remove() {
    this.reactRoot.unmount();
  }
});
```

```
    Backbone.View.prototype.remove.call(this);
  }
});
```

[在 CodePen 上运行](#)

在 `remove` 方法中我们也需要调用 `root.unmount()` 以便在它解除的时候 React 清理组件树相关的事件处理的注册和其他的资源，这点是是很重要的。

当一个组件在 React 树中从内部删除的时候，清理工作是自动完成的，但是因为我们现在手动移除整个树，我们必须调用这个方法。

和 Model 层集成

虽然通常是推荐使用单向数据流动的，例如 [React state](#), [Flux](#), 或者 [Redux](#), React 组件也可以使用一个其他框架和库的 Model 层。

在 React 组件中使用 Backbone 的 Model

在 React 组件中使用 [Backbone](#) 的 model 和 collection 最简单的方法就是监听多种变化事件并且手动强制触发一个更新。

负责渲染 model 的组件会监听 `'change'` 事件，而负责渲染 collection 的组件需要监听 `'add'` 和 `'remove'` 事件。在这两种情况中，调用 `this.forceUpdate()` 来使用新的数据重新渲染组件。

在下面的例子中，`List` 组件渲染一个 Backbone collection，使用 `Item` 组件来渲染独立的项。

```
class Item extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
  }

  handleChange() {
    this.forceUpdate();
  }

  componentDidMount() {
    this.props.model.on('change', this.handleChange);
  }

  componentWillUnmount() {
    this.props.model.off('change', this.handleChange);
  }

  render() {
    return <li>{this.props.model.get('text')}</li>;
  }
}

class List extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
  }

  handleChange() {
```

```

    this.forceUpdate();
}

componentDidMount() {
  this.props.collection.on('add', 'remove', this.handleChange);
}

componentWillUnmount() {
  this.props.collection.off('add', 'remove', this.handleChange);
}

render() {
  return (
    <ul>
      {this.props.collection.map(model => (
        <Item key={model.cid} model={model} />
      ))}
    </ul>
  );
}
}

```

[在 CodePen 上运行](#)

从 Backbone Model 提取数据

前面的方式需要你的 React 组件知道 Backbone 的 model 和 collection。如果你计划迁移到另一个数据管理方案，你可能希望将关于Backbone的知识集中在尽可能少的代码部分中。

其中一个解决方案就是每当 model 中的属性变化时都把它提取成简单数据，并且把这个逻辑放在一个独立的地方。下面是一个[高阶组件](#)，它提取了 Backbone model 的所有数据存放到 state 中，并将数据传递到被包裹的组件中。

通过这种方法，只有高阶组件需要知道 Backbone model 的内部构造，而且应用中大多数的组件可以保持和 Backbone 无关。

在下面的例子中，我们会拷贝一份 model 的属性来形成初始的 state。我们订阅 `change` 事件（并且在取消挂载时停止订阅），而当变化发生时，我们使用 model 的当前属性更新这个 state。最终，我们确保了只要 `model` 属性本身变化的时候，我们不要忘了停止旧 model 的订阅并开始订阅新的 model。

请注意，这个例子并不是为了彻底完整展示如何与 Backbone 集成，而是它应该让你了解如何以通用的方式处理此问题：

```

function connectToBackboneModel(WrappedComponent) {
  return class BackboneComponent extends React.Component {
    constructor(props) {
      super(props);
      this.state = Object.assign({}, props.model.attributes);
      this.handleChange = this.handleChange.bind(this);
    }

    componentDidMount() {
      this.props.model.on('change', this.handleChange);
    }

    componentWillReceiveProps(nextProps) {
      this.setState(Object.assign({}, nextProps.model.attributes));
      if (nextProps.model !== this.props.model) {
        this.props.model.off('change', this.handleChange);
      }
    }
  }
}

```

```

        nextProps.model.on('change', this.handleChange);
    }
}

componentWillUnmount() {
    this.props.model.off('change', this.handleChange);
}

handleChange(model) {
    this.setState(model.changedAttributes());
}

render() {
    const propsExceptModel = Object.assign({}, this.props);
    delete propsExceptModel.model;
    return <WrappedComponent {...propsExceptModel} {...this.state} />;
}
}
}

```

要演示如何使用它，我们会链接一个 `NameInput` React 组件到一个 Backbone model，并且每当输入框变化时更新它的 `firstName` 属性：

```

function NameInput(props) {
    return (
        <p>
            <input value={props.firstName} onChange={props.handleChange} />
            <br />
            My name is {props.firstName}.
        </p>
    );
}

const BackboneNameInput = connectToBackboneModel(NameInput);

function Example(props) {
    function handleChange(e) {
        props.model.set('firstName', e.target.value);
    }

    return (
        <BackboneNameInput
            model={props.model}
            handleChange={handleChange}
        />
    );
}

const model = new Backbone.Model({ firstName: 'Frodo' });
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Example model={model} />);

```

[在 CodePen 上运行](#)

这个技术并不仅限于 Backbone。你可以通过在生命周期方法中订阅其更改，并选择性地，拷贝数据到本地 React state，来将 React 用于任何 model 库。

[Go to TOC](#)

JSX 简介

设想如下变量声明：

```
const element = <h1>Hello, world!</h1>;
```

这个有趣的标签语法既不是字符串也不是 HTML。

它被称为 JSX，是一个 JavaScript 的语法扩展。我们建议在 React 中配合使用 JSX，JSX 可以很好地描述 UI 应该呈现出它应有交互的本质形式。JSX 可能会使人联想到模板语言，但它具有 JavaScript 的全部功能。

JSX 可以生成 React “元素”。我们将在[下一章节](#)中探讨如何将这些元素渲染为 DOM。下面我们看下学习 JSX 所需的基础知识。

为什么使用 JSX？

React 认为渲染逻辑本质上与其他 UI 逻辑内在耦合，比如，在 UI 中需要绑定处理事件、在某些时刻状态发生变化时需要通知到 UI，以及需要在 UI 中展示准备好的数据。

React 并没有采用将标记与逻辑分离到不同文件这种人为的分离方式，而是通过将二者共同存放在称之为“组件”的松散耦合单元之中，来实现[关注点分离](#)。我们将在[后面章节](#)中深入学习组件。如果你还没有适应在 JS 中使用标记语言，这个[会议讨论](#)应该可以说服你。

React 不强制要求使用 JSX，但是大多数人发现，在 JavaScript 代码中将 JSX 和 UI 放在一起时，会在视觉上有辅助作用。它还可以使 React 显示更多有用的错误和警告消息。

搞清楚这个问题后，我们就开始学习 JSX 吧！

在 JSX 中嵌入表达式

在下面的例子中，我们声明了一个名为 `name` 的变量，然后在 JSX 中使用它，并将它包裹在大括号中：

```
const name = 'Josh Perez';
const element = <h1>Hello, {name}</h1>;
```

在 JSX 语法中，你可以在大括号内放置任何有效的 [JavaScript 表达式](#)。例如，`2 + 2`，`user.firstName` 或 `formatName(user)` 都是有效的 JavaScript 表达式。

在下面的示例中，我们将调用 JavaScript 函数 `formatName(user)` 的结果，并将结果嵌入到 `<h1>` 元素中。

```
function formatName(user) {
  return user.firstName + ' ' + user.lastName;
}

const user = {
  firstName: 'Harper',
  lastName: 'Perez'
};

const element = (
  <h1>
```

```
Hello, {formatName(user)}!
  </h1>
);
```

[在 CodePen 上试试](#)

为了便于阅读，我们会将 JSX 拆分为多行。同时，我们建议将内容包裹在括号中，虽然这样做不是强制要求的，但是这可以避免遇到[自动插入分号](#)陷阱。

JSX 也是一个表达式

在编译之后，JSX 表达式会被转为普通 JavaScript 函数调用，并且对其取值后得到 JavaScript 对象。

也就是说，你可以在 `if` 语句和 `for` 循环的代码块中使用 JSX，将 JSX 赋值给变量，把 JSX 当作参数传入，以及从函数中返回 JSX：

```
function getGreeting(user) {
  if (user) {
    return <h1>Hello, {formatName(user)}!</h1>;
  }
  return <h1>Hello, Stranger.</h1>;
}
```

JSX 中指定属性

你可以通过使用引号，来将属性值指定为字符串字面量：

```
const element = <a href="https://www.reactjs.org"> link </a>;
```

也可以使用大括号，来在属性值中插入一个 JavaScript 表达式：

```
const element = <img src={user.avatarUrl}></img>;
```

在属性中嵌入 JavaScript 表达式时，不要在大括号外面加上引号。你应该仅使用引号（对于字符串值）或大括号（对于表达式）中的一个，对于同一属性不能同时使用这两种符号。

警告：

因为 JSX 语法上更接近 JavaScript 而不是 HTML，所以 React DOM 使用 `camelCase`（小驼峰命名）来定义属性的名称，而不使用 HTML 属性名称的命名约定。

例如，JSX 里的 `class` 变成了 `className`，而 `tabindex` 则变为 `tabIndex`。

使用 JSX 指定子元素

假如一个标签里面没有内容，你可以使用 `/>` 来闭合标签，就像 XML 语法一样：

```
const element = <img src={user.avatarUrl} />;
```

JSX 标签里能够包含很多子元素：

```
const element = (
  <div>
    <h1>Hello!</h1>
    <h2>Good to see you here.</h2>
  </div>
);
```

JSX 防止注入攻击

你可以安全地在 JSX 当中插入用户输入内容：

```
const title = response.potentiallyMaliciousInput;
// 直接使用是安全的：
const element = <h1>{title}</h1>;
```

React DOM 在渲染所有输入内容之前，默认会进行[转义](#)。它可以确保在你的应用中，永远不会注入那些并非自己明确编写的内容。所有的内容在渲染之前都被转换成了字符串。这样可以有效地防止 [XSS \(cross-site-scripting, 跨站脚本\)](#) 攻击。

JSX 表示对象

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

以下两种示例代码完全等效：

```
const element = (
  <h1 className="greeting">
    Hello, world!
  </h1>
);
```

```
const element = React.createElement(
  'h1',
  {className: 'greeting'},
  'Hello, world!'
);
```

`React.createElement()` 会预先执行一些检查，以帮助你编写无错代码，但实际上它创建了一个这样的对象：

```
// 注意：这是简化过的结构
const element = {
  type: 'h1',
  props: {
    className: 'greeting',
    children: 'Hello, world!'
  }
};
```

这些对象被称为“React 元素”。它们描述了你希望在屏幕上看到的内容。React 通过读取这些对象，然后使用它们来构建 DOM 以及保持随时更新。

我们将在[下一章节](#)中探讨如何将 React 元素渲染为 DOM。

提示：

我们推荐在你使用的编辑器中，使用“[Babel](#)”提供的语言定义，来正确地高亮显示 ES6 和 JSX 代码。

[Go to TOC](#)

深入 JSX

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

```
<MyButton color="blue" shadowSize={2}>
  Click Me
</MyButton>
```

会编译为：

```
React.createElement(
  MyButton,
  {color: 'blue', shadowSize: 2},
  'Click Me'
)
```

如果没有子节点，你还可以使用自闭合的标签形式，如：

```
<div className="sidebar" />
```

会编译为：

```
React.createElement(
  'div',
  {className: 'sidebar'}
)
```

如果你想测试一些特定的 JSX 会转换成什么样的 JavaScript，你可以尝试使用 [在线的 Babel 编译器](#)。

指定 React 元素类型

JSX 标签的第一部分指定了 React 元素的类型。

大写字母开头的 JSX 标签意味着它们是 React 组件。这些标签会被编译为对命名变量的直接引用，所以，当你使用 JSX `<Foo />` 表达式时，`Foo` 必须包含在作用域内。

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。

例如，在如下代码中，虽然 `React` 和 `CustomButton` 并没有被直接使用，但还是需要导入：

```
import React from 'react';
import CustomButton from './CustomButton';

function WarningButton() {
  // return React.createElement(CustomButton, {color: 'red'}, null);
  return <CustomButton color="red" />;
}
```

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。例如，如果 `MyComponents.DatePicker` 是一个组件，你可以在 JSX 中直接使用：

```
import React from 'react';

const MyComponents = {
  DatePicker: function DatePicker(props) {
    return <div>Imagine a {props.color} datepicker here.</div>;
  }
}

function BlueDatePicker() {
  return <MyComponents.DatePicker color="blue" />;
}
```

用户定义的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 `` 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)`。

我们建议使用大写字母开头命名自定义组件。如果你确实需要一个以小写字母开头的组件，则在 JSX 中使用它之前，必须将它赋值给一个大写字母开头的变量。

例如，以下的代码将无法按照预期运行：

```
import React from 'react';

// 错误！组件应该以大写字母开头：
function hello(props) {
  // 正确！这种 <div> 的使用是合法的，因为 div 是一个有效的 HTML 标签
  return <div>Hello {props.toWhat}</div>;
}

function HelloWorld() {
  // 错误！React 会认为 <hello /> 是一个 HTML 标签，因为它没有以大写字母开头：
  return <hello toWhat="World" />;
}
```

要解决这个问题，我们需要重命名 `hello` 为 `Hello`，同时在 JSX 中使用 `<Hello />`：

```
import React from 'react';

// 正确！组件需要以大写字母开头：
function Hello(props) {
  // 正确！这种 <div> 的使用是合法的，因为 div 是一个有效的 HTML 标签
  return <div>Hello {props.toWhat}</div>;
}

function HelloWorld() {
```

```
// 正确! React 知道 <Hello /> 是一个组件, 因为它是大写字母开头的:
return <Hello toWhat="World" />;
}
```

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：

```
import React from 'react';
import { PhotoStory, VideoStory } from './stories';

const components = {
  photo: PhotoStory,
  video: VideoStory
};

function Story(props) {
  // 错误! JSX 类型不能是一个表达式。
  return <components[props.storyType] story={props.story} />;
}
```

要解决这个问题，需要首先将类型赋值给一个大写字母开头的变量：

```
import React from 'react';
import { PhotoStory, VideoStory } from './stories';

const components = {
  photo: PhotoStory,
  video: VideoStory
};

function Story(props) {
  // 正确! JSX 类型可以是大写字母开头的变量。
  const SpecificStory = components[props.storyType];
  return <SpecificStory story={props.story} />;
}
```

JSX 中的 Props

有多种方式可以在 JSX 中指定 props。

JavaScript 表达式作为 Props

你可以把包裹在 `{}` 中的 JavaScript 表达式作为一个 prop 传递给 JSX 元素。例如，如下的 JSX：

```
<MyComponent foo={1 + 2 + 3 + 4} />
```

在 `MyComponent` 中，`props.foo` 的值等于 `1 + 2 + 3 + 4` 的执行结果 `10`。

`if` 语句以及 `for` 循环不是 JavaScript 表达式，所以不能在 JSX 中直接使用。但是，你可以用在 JSX 以外的代码中。比如：

```
function NumberDescriber(props) {
  let description;
  if (props.number % 2 == 0) {
    description = <strong>even</strong>;
  } else {
    description = <i>odd</i>;
  }
  return <div>{props.number} is an {description} number</div>;
}
```

你可以在对应的章节中学习更多关于[条件渲染](#)和[循环](#)的内容。

字符串字面量

你可以将字符串字面量赋值给 prop。如下两个 JSX 表达式是等价的：

```
<MyComponent message="hello world" />
<MyComponent message={['hello world']} />
```

当你将字符串字面量赋值给 prop 时，它的值是未转义的。所以，以下两个 JSX 表达式是等价的：

```
<MyComponent message=&lt;3&gt; />
<MyComponent message={['<3'']} />
```

这种行为通常是不重要的，这里只是提醒有这个用法。

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：

```
<MyTextBox autocomplete />
<MyTextBox autocomplete={true} />
```

通常，我们不建议不传递 `value` 给 prop，因为这可能与 [ES6 对象简写](#)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。

属性展开

如果你已经有了一个 props 对象，你可以使用展开运算符 `...` 来在 JSX 中传递整个 props 对象。以下两个组件是等价的：

```
function App1() {
  return <Greeting firstName="Ben" lastName="Hector" />;
}

function App2() {
  const props = {firstName: 'Ben', lastName: 'Hector'};
  return <Greeting {...props} />;
}
```

你还可以选择只保留当前组件需要接收的 props，并使用展开运算符将其他 props 传递下去。

```

const Button = props => {
  const { kind, ...other } = props;
  const className = kind === "primary" ? "PrimaryButton" : "SecondaryButton";
  return <button className={className} {...other} />;
};

const App = () => {
  return (
    <div>
      <Button kind="primary" onClick={() => console.log("clicked!")}>
        Hello World!
      </Button>
    </div>
  );
};

```

在上述例子中，`kind` 的 prop 会被安全的保留，它将不会被传递给 DOM 中的 `<button>` 元素。所有其他的 props 会通过 `...other` 对象传递，使得这个组件的应用可以非常灵活。你可以看到它传递了一个 `onClick` 和 `children` 属性。

属性展开在某些情况下很有用，但是也很容易将不必要的 props 传递给不相关的组件，或者将无效的 HTML 属性传递给 DOM。我们建议谨慎的使用该语法。

JSX 中的子元素

包含在开始和结束标签之间的 JSX 表达式内容将作为特定属性 `props.children` 传递给外层组件。有几种不同的方法来传递子元素：

字符串字面量

你可以将字符串放在开始和结束标签之间，此时 `props.children` 就只是该字符串。这对于很多内置的 HTML 元素很有用。例如：

```
<MyComponent>Hello world!</MyComponent>
```

这是一个合法的 JSX，`MyComponent` 中的 `props.children` 是一个简单的未转义字符串 `"Hello world!"`。因此你可以采用编写 HTML 的方式来编写 JSX。如下所示：

```
<div>This is valid HTML && JSX at the same time.</div>
```

JSX 会移除行首尾的空格以及空行。与标签相邻的空行均会被删除，文本字符串之间的新行会被压缩为一个空格。因此以下的几种方式都是等价的：

```

<div>Hello World</div>

<div>
  Hello World
</div>

<div>
  Hello
  World
</div>

```

```
<div>
  Hello World
</div>
```

JSX 子元素

子元素允许由多个 JSX 元素组成。这对于嵌套组件非常有用：

```
<MyContainer>
  <MyFirstComponent />
  <MySecondComponent />
</MyContainer>
```

你可以将不同类型的子元素混合在一起，因此你可以将字符串字面量与 JSX 子元素一起使用。这也是 JSX 类似 HTML 的一种表现，所以如下代码是合法的 JSX 并且也是合法的 HTML：

```
<div>
  Here is a list:
  <ul>
    <li>Item 1</li>
    <li>Item 2</li>
  </ul>
</div>
```

React 组件也能够返回存储在数组中的一组元素：

```
render() {
  // 不需要用额外的元素包裹列表元素!
  return [
    // 不要忘记设置 key :
    <li key="A">First item</li>,
    <li key="B">Second item</li>,
    <li key="C">Third item</li>,
  ];
}
```

JavaScript 表达式作为子元素

JavaScript 表达式可以被包裹在 {} 中作为子元素。例如，以下表达式是等价的：

```
<MyComponent>foo</MyComponent>
<MyComponent>{'foo'}</MyComponent>
```

这对于展示任意长度的列表非常有用。例如，渲染 HTML 列表：

```
function Item(props) {
  return <li>{props.message}</li>;
}

function TodoList() {
  const todos = ['finish doc', 'submit pr', 'nag dan to review'];
  return (
    <ul>
      {todos.map((message) => <Item key={message} message={message} />)}
```

```

        </ul>
    );
}

```

JavaScript 表达式也可以和其他类型的子元素组合。这种做法可以方便地替代模板字符串：

```

function Hello(props) {
  return <div>Hello {props.addressee}</div>;
}

```

函数作为子元素

通常，JSX 中的 JavaScript 表达式将会被计算为字符串、React 元素或者是列表。不过，`props.children` 和其他 prop 一样，它可以传递任意类型的数据，而不仅仅是 React 已知的可渲染类型。例如，如果你有一个自定义组件，你可以把回调函数作为 `props.children` 进行传递：

```

// 调用子元素回调 numTimes 次，来重复生成组件
function Repeat(props) {
  let items = [];
  for (let i = 0; i < props.numTimes; i++) {
    items.push(props.children(i));
  }
  return <div>{items}</div>;
}

function ListOfTenThings() {
  return (
    <Repeat numTimes={10}>
      {(index) => <div key={index}>This is item {index} in the list</div>}
    </Repeat>
  );
}

```

你可以将任何东西作为子元素传递给自定义组件，只要确保在该组件渲染之前能够被转换成 React 理解的对象。这种用法并不常见，但可以用于扩展 JSX。

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：

```

<div />
<div></div>
<div>{false}</div>
<div>{null}</div>
<div>{undefined}</div>
<div>{true}</div>

```

这有助于依据特定条件来渲染其他的 React 元素。例如，在以下 JSX 中，仅当 `showHeader` 为 `true` 时，才会渲染 `<Header />` 组件：

```
<div>
  {showHeader && <Header />}
  <Content />
</div>
```

值得注意的是有一些 "falsy" 值，如数字 0，仍然会被 React 渲染。例如，以下代码并不会像你预期那样工作，因为当 `props.messages` 是空数组时，将会渲染为数字 0：

```
<div>
  {props.messages.length &&
    <MessageList messages={props.messages} />
  }
</div>
```

要解决这个问题，确保 `&&` 之前的表达式总是布尔值：

```
<div>
  {props.messages.length > 0 &&
    <MessageList messages={props.messages} />
  }
</div>
```

反之，如果你想渲染 `false`、`true`、`null`、`undefined` 等值，你需要先将它们转换为字符串：

```
<div>
  My JavaScript variable is {String(myVariable)}.
</div>
```

过时的 Context

注意：

过时的 context API 会在未来的主要版本中被移除。 使用 16.3 版本中引入的 [新的 context API](#)。 过时的 API 将会继续在所有 16.x 版本中工作。

如何使用 Context

本节记录一个过时的 API。查看 [新的 API](#)。

假设你有这样一个结构：

```
class Button extends React.Component {
  render() {
    return (
      <button style={{background: this.props.color}}>
        {this.props.children}
      </button>
    );
  }
}

class Message extends React.Component {
  render() {
    return (
      <div>
        {this.props.text} <Button color={this.props.color}>Delete</Button>
      </div>
    );
  }
}

class MessageList extends React.Component {
  render() {
    const color = "purple";
    const children = this.props.messages.map((message) =>
      <Message text={message.text} color={color} />
    );
    return <div>{children}</div>;
  }
}
```

在这个示例中，我们手动传递一个 `color` 属性来设置 `Button` 和 `Message` 组件的样式。使用 context，我们可以通过组件树自动传递属性：

```
import PropTypes from 'prop-types';

class Button extends React.Component {
```

```

render() {
  return (
    <button style={{background: this.context.color}}>
      {this.props.children}
    </button>
  );
}

Button.contextTypes = {
  color: PropTypes.string
};

class Message extends React.Component {
  render() {
    return (
      <div>
        {this.props.text} <Button>Delete</Button>
      </div>
    );
  }
}

class MessageList extends React.Component {
  getChildContext() {
    return {color: "purple"};
  }

  render() {
    const children = this.props.messages.map((message) =>
      <Message text={message.text} />
    );
    return <div>{children}</div>;
  }
}

MessageList.childContextTypes = {
  color: PropTypes.string
};

```

通过给 `MessageList` (`context` 的生产者) 添加 `childContextTypes` 和 `getChildContext`, React 自动向下传递信息, 子树上的所有组件 (在这个例子中是 `Button`) 可以通过定义 `contextTypes` 来访问 `context`。

如果 `contextTypes` 没有被定义, `context` 就会是个空对象。

注意:

自从 React 15.5 版本之后, `React.PropTypes` 已经转移到了另一个包。请改用 `prop-types 库` 来定义 `contextTypes`。

我们提供了 [一个 codemod 脚本](#) 来自动转换。

父-子 结合

本节记录一个过时的 API。查看 [新的 API](#)。

Context 也能让你构建一个父子组件通信的 API。例如，[React Router V4](#) 这个库就是这样工作的：

```
import { BrowserRouter as Router, Route, Link } from 'react-router-dom';

const BasicExample = () => (
  <Router>
    <div>
      <ul>
        <li><Link to="/">Home</Link></li>
        <li><Link to="/about">About</Link></li>
        <li><Link to="/topics">Topics</Link></li>
      </ul>

      <hr />

      <Route exact path="/" component={Home} />
      <Route path="/about" component={About} />
      <Route path="/topics" component={Topics} />
    </div>
  </Router>
);
```

通过从 `Router` 组件向下传递一些信息，每一个 `Link` 和 `Route` 能够与包含它的 `Router` 通信。

在你构建类似的 API 的组件之前，先考虑是否有更简洁的方式。例如，如果你愿意的话，你可以将整个 React 组件作为 `props`。

在生命周期方法中引用 Context

本节记录一个过时的 API。查看 [新的 API](#)。

如果一个组件内定义了 `contextTypes`，下面的 [生命周期方法](#) 会接收一个额外参数，就是 `context` 对象：

- `constructor(props, context)`
- `componentWillReceiveProps(nextProps, nextContext)`
- `shouldComponentUpdate(nextProps, nextState, nextContext)`
- `componentWillUpdate(nextProps, nextState, nextContext)`

注意：

从 React 16 开始，`componentDidUpdate` 不再接收 `prevContext`。

在函数组件中引用 Context

本节记录一个过时的 API。查看 [新的 API](#)。

只要 `contextTypes` 被定义为函数的一个属性，函数组件也能够引用 `context`。下面的代码展示了一个函数组件写法的 `Button` 组件。

```
import PropTypes from 'prop-types';

const Button = ({children}, context) =>
  <button style={{background: context.color}}>
    {children}
  </button>

Button.contextTypes = {color: PropTypes.string};
```

更新 Context

本节记录一个过时的 API。查看 [新的 API](#)。

不要这样做。

React 有一个 API 可以更新 `context`，但它基本上是不靠谱的，你不应该使用它。

当 `state` 或者 `props` 改变的时候，`get ChildContext` 函数就会被调用。为了更新 `context` 里的数据，使用 `this.setState` 触发当前 `state` 的更新。这样会产生一个新的 `context` 并且子组件会接收到变化。

```
import PropTypes from 'prop-types';

class MediaQuery extends React.Component {
  constructor(props) {
    super(props);
    this.state = {type: 'desktop'};
  }

  getChildContext() {
    return {type: this.state.type};
  }

  componentDidMount() {
    const checkMediaQuery = () => {
      const type = window.matchMedia("(min-width: 1025px)").matches ? 'desktop' : 'mobile';
      if (type !== this.state.type) {
        this.setState({type});
      }
    };
    window.addEventListener('resize', checkMediaQuery);
    checkMediaQuery();
  }
}
```

```
render() {
  return this.props.children;
}

MediaQuery.childContextTypes = {
  type: PropTypes.string
};
```

问题是，如果组件提供的一个 context 发生了变化，而中间父组件的 `shouldComponentUpdate` 返回 `false`，那么使用到该值的后代组件不会进行更新。使用了 context 的组件则完全失控，所以基本上没有办法能够可靠的更新 context。[这篇博客文章](#)很好地解释了为何会出现此类问题，以及你该如何规避它。

事件池

注意

此文章仅适用于 React 16 及更早版本、React Native。

Web 端的 React 17 不使用事件池。

[查看更多](#) 关于 React 17 中的相关改动。

`SyntheticEvent` 对象会被放入池中统一管理。这意味着 `SyntheticEvent` 对象可以被复用，当所有事件处理函数被调用之后，其所有属性都会被置空。例如，以下代码是无效的：

```
function handleChange(e) {
  // This won't work because the event object gets reused.
  setTimeout(() => {
    console.log(e.target.value); // Too late!
  }, 100);
}
```

如果你需要在事件处理函数运行之后获取事件对象的属性，你需要调用 `e.persist()`：

```
function handleChange(e) {
  // Prevents React from resetting its properties:
  e.persist();

  setTimeout(() => {
    console.log(e.target.value); // Works
  }, 100);
}
```

[Go to TOC](#)

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。让我们看看它是如何运作的。

在本节中，我们将创建一个用于计算水在给定温度下是否会沸腾的温度计算器。

我们将从一个名为 `BoilingVerdict` 的组件开始，它接受 `celsius` 温度作为一个 prop，并据此打印出该温度是否足以将水煮沸的结果。

```
function BoilingVerdict(props) {
  if (props.celsius >= 100) {
    return <p>The water would boil.</p>;
  }
  return <p>The water would not boil.</p>;
}
```

接下来，我们创建一个名为 `Calculator` 的组件。它渲染一个用于输入温度的 `<input>`，并将其值保存在 `this.state.temperature` 中。

另外，它根据当前输入值渲染 `BoilingVerdict` 组件。

```
class Calculator extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {temperature: ''};
  }

  handleChange(e) {
    this.setState({temperature: e.target.value});
  }

  render() {
    const temperature = this.state.temperature;
    return (
      <fieldset>
        <legend>Enter temperature in Celsius:</legend>
        <input
          value={temperature}
          onChange={this.handleChange} />
        <BoilingVerdict
          celsius={parseFloat(temperature)} />
      </fieldset>
    );
  }
}
```

[在 CodePen 上尝试](#)

添加第二个输入框

我们的新需求是，在已有摄氏温度输入框的基础上，我们提供华氏度的输入框，并保持两个输入框的数据同步。

我们先从 `Calculator` 组件中抽离出 `TemperatureInput` 组件，然后为其添加一个新的 `scale` prop，它可以是 `"c"` 或是 `"f"`：

```
const scaleNames = {
  c: 'Celsius',
  f: 'Fahrenheit'
};

class TemperatureInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {temperature: ''};
  }

  handleChange(e) {
    this.setState({temperature: e.target.value});
  }

  render() {
    const temperature = this.state.temperature;
    const scale = this.props.scale;
    return (
      <fieldset>
        <legend>Enter temperature in {scaleNames[scale]}:</legend>
        <input value={temperature}>
          onChange={this.handleChange}
        </input>
      </fieldset>
    );
  }
}
```

我们现在可以修改 `Calculator` 组件让它渲染两个独立的温度输入框组件：

```
class Calculator extends React.Component {
  render() {
    return (
      <div>
        <TemperatureInput scale="c" />
        <TemperatureInput scale="f" />
      </div>
    );
  }
}
```

[在 CodePen 上尝试](#)

我们现在有了两个输入框，但当你在其中一个输入温度时，另一个并不会更新。这与我们的要求相矛盾：我们希望让它们保持同步。

另外，我们也不能通过 `Calculator` 组件展示 `BoilingVerdict` 组件的渲染结果。因为 `Calculator` 组件并不知道隐藏在 `TemperatureInput` 组件中的当前温度是多少。

编写转换函数

首先，我们将编写两个可以在摄氏度与华氏度之间相互转换的函数：

```

function toCelsius(fahrenheit) {
  return (fahrenheit - 32) * 5 / 9;
}

function toFahrenheit(celsius) {
  return (celsius * 9 / 5) + 32;
}

```

上述两个函数仅做数值转换。而我们将编写另一个函数，它接受字符串类型的 `temperature` 和转换函数作为参数并返回一个字符串。我们将使用它来依据一个输入框的值计算出另一个输入框的值。

当输入 `temperature` 的值无效时，函数返回空字符串，反之，则返回保留三位小数并四舍五入后的转换结果：

```

function tryConvert(temperature, convert) {
  const input = parseFloat(temperature);
  if (Number.isNaN(input)) {
    return '';
  }
  const output = convert(input);
  const rounded = Math.round(output * 1000) / 1000;
  return rounded.toString();
}

```

例如，`tryConvert('abc', toCelsius)` 返回一个空字符串，而 `tryConvert('10.22', toFahrenheit)` 返回 '`50.396`'。

状态提升

到目前为止，两个 `TemperatureInput` 组件均在各自内部的 `state` 中相互独立地保存着各自的数据。

```

class TemperatureInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {temperature: ''};
  }

  handleChange(e) {
    this.setState({temperature: e.target.value});
  }

  render() {
    const temperature = this.state.temperature;
    // ...
  }
}

```

然而，我们希望两个输入框内的数值彼此能够同步。当我们更新摄氏度输入框内的数值时，华氏度输入框内应当显示转换后的华氏温度，反之亦然。

在 React 中，将多个组件中需要共享的 `state` 向上移动到它们的最近共同父组件中，便可实现共享 `state`。这就是所谓的“状态提升”。接下来，我们将 `TemperatureInput` 组件中的 `state` 移动至 `Calculator` 组件中去。

如果 `Calculator` 组件拥有了共享的 `state`，它将成为两个温度输入框中当前温度的“数据源”。它能够使得两个温度输入框的数值彼此保持一致。由于两个 `TemperatureInput` 组件的 `props` 均来自共同的父组件 `Calculator`，因此两个输入框中的内容将始终保持一致。

让我们看看这是如何一步一步实现的。

首先，我们将 `TemperatureInput` 组件中的 `this.state.temperature` 替换为 `this.props.temperature`。现在，我们先假定 `this.props.temperature` 已经存在，尽管将来我们需要通过 `Calculator` 组件将其传入：

```
render() {
  // Before: const temperature = this.state.temperature;
  const temperature = this.props.temperature;
  // ...
```

我们知道 `props` 是只读的。当 `temperature` 存在于 `TemperatureInput` 组件的 `state` 中时，组件调用 `this.setState()` 便可修改它。然而，`temperature` 是由父组件传入的 prop，`TemperatureInput` 组件便失去了对它的控制权。

在 React 中，这个问题通常是通过使用“受控组件”来解决的。与 DOM 中的 `<input>` 接受 `value` 和 `onChange` 一样，自定义的 `TemperatureInput` 组件接受 `temperature` 和 `onTemperatureChange` 这两个来自父组件 `Calculator` 的 props。

现在，当 `TemperatureInput` 组件想更新温度时，需调用 `this.props.onTemperatureChange` 来更新它：

```
handleChange(e) {
  // Before: this.setState({temperature: e.target.value});
  this.props.onTemperatureChange(e.target.value);
  // ...
```

注意：

自定义组件中的 `temperature` 和 `onTemperatureChange` 这两个 prop 的命名没有任何特殊含义。我们可以给它们取其它任意的名字，例如，把它们命名为 `value` 和 `onChange` 就是一种习惯。

`onTemperatureChange` 的 prop 和 `temperature` 的 prop 一样，均由父组件 `Calculator` 提供。它通过修改父组件自身的内部 `state` 来处理数据的变化，进而使用新的数值重新渲染两个输入框。我们将很快看到修改后的 `Calculator` 组件效果。

在深入研究 `Calculator` 组件的变化之前，让我们回顾一下 `TemperatureInput` 组件的变化。我们移除组件自身的 `state`，通过使用 `this.props.temperature` 替代 `this.state.temperature` 来读取温度数据。当我们想要响应数据改变时，我们需要调用 `Calculator` 组件提供的 `this.props.onTemperatureChange()`，而不再使用 `this.setState()`。

```
class TemperatureInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
  }

  handleChange(e) {
    this.props.onTemperatureChange(e.target.value);
  }

  render() {
```

```

const temperature = this.props.temperature;
const scale = this.props.scale;
return (
  <fieldset>
    <legend>Enter temperature in {scaleNames[scale]}:</legend>
    <input value={temperature}
           onChange={this.handleChange} />
  </fieldset>
);
}
}

```

现在，让我们把目光转向 `Calculator` 组件。

我们会把当前输入的 `temperature` 和 `scale` 保存在组件内部的 `state` 中。这个 `state` 就是从两个输入框组件中“提升”而来的，并且它将用作两个输入框组件的共同“数据源”。这是我们为了渲染两个输入框所需要的所有数据的最小表示。

例如，当我们在摄氏度输入框中键入 37 时，`Calculator` 组件中的 `state` 将会是：

```
{
  temperature: '37',
  scale: 'c'
}
```

如果我们之后修改华氏度的输入框中的内容为 212 时，`Calculator` 组件中的 `state` 将会是：

```
{
  temperature: '212',
  scale: 'f'
}
```

我们可以存储两个输入框中的值，但这并不是必要的。我们只需要存储最近修改的温度及其计量单位即可，根据当前的 `temperature` 和 `scale` 就可以计算出另一个输入框的值。

由于两个输入框中的数值由同一个 `state` 计算而来，因此它们始终保持同步：

```

class Calculator extends React.Component {
  constructor(props) {
    super(props);
    this.handleCelsiusChange = this.handleCelsiusChange.bind(this);
    this.handleFahrenheitChange = this.handleFahrenheitChange.bind(this);
    this.state = {temperature: '', scale: 'c'};
  }

  handleCelsiusChange(temperature) {
    this.setState({scale: 'c', temperature});
  }

  handleFahrenheitChange(temperature) {
    this.setState({scale: 'f', temperature});
  }

  render() {
    const scale = this.state.scale;
    const temperature = this.state.temperature;
    const celsius = scale === 'f' ? tryConvert(temperature, toCelsius) :
      temperature;
    const fahrenheit = scale === 'c' ? tryConvert(temperature, toFahrenheit) :

```

```

temperature;

return (
  <div>
    <TemperatureInput
      scale="c"
      temperature={celsius}
      onTemperatureChange={this.handleCelsiusChange} />
    <TemperatureInput
      scale="f"
      temperature={fahrenheit}
      onTemperatureChange={this.handleFahrenheitChange} />
    <BoilingVerdict
      celsius={parseFloat(celsius)} />
  </div>
);
}
}

```

[在 CodePen 上尝试](#)

现在无论你编辑哪个输入框中的内容，`Calculator` 组件中的 `this.state.temperature` 和 `this.state.scale` 均会被更新。其中一个输入框保留用户的输入并取值，另一个输入框始终基于这个值显示转换后的结果。

让我们来重新梳理一下当你对输入框内容进行编辑时会发生些什么：

- React 会调用 DOM 中 `<input>` 的 `onChange` 方法。在本实例中，它是 `TemperatureInput` 组件的 `handleChange` 方法。
- `TemperatureInput` 组件中的 `handleChange` 方法会调用 `this.props.onTemperatureChange()`，并传入新输入的值作为参数。其 `props` 诸如 `onTemperatureChange` 之类，均由父组件 `Calculator` 提供。
- 起初渲染时，用于摄氏度输入的子组件 `TemperatureInput` 中的 `onTemperatureChange` 方法与 `Calculator` 组件中的 `handleCelsiusChange` 方法相同，而，用于华氏度输入的子组件 `TemperatureInput` 中的 `onTemperatureChange` 方法与 `Calculator` 组件中的 `handleFahrenheitChange` 方法相同。因此，无论哪个输入框被编辑都会调用 `Calculator` 组件中对应的方法。
- 在这些方法内部，`Calculator` 组件通过使用新的输入值与当前输入框对应的温度计量单位来调用 `this.setState()` 进而请求 React 重新渲染自己本身。
- React 调用 `Calculator` 组件的 `render` 方法得到组件的 UI 呈现。温度转换在这时进行，两个输入框中的数值通过当前输入温度和其计量单位来重新计算获得。
- React 使用 `Calculator` 组件提供的新 `props` 分别调用两个 `TemperatureInput` 子组件的 `render` 方法来获取子组件的 UI 呈现。
- React 调用 `BoilingVerdict` 组件的 `render` 方法，并将摄氏温度值以组件 `props` 方式传入。
- React DOM 根据输入值匹配水是否沸腾，并将结果更新至 DOM。我们刚刚编辑的输入框接收其当前值，另一个输入框内容更新为转换后的温度值。

得益于每次的更新都经历相同的步骤，两个输入框的内容才能始终保持同步。

学习小结

在 React 应用中，任何可变数据应当只有一个相对应的唯一“数据源”。通常，`state` 都是首先添加到需要渲染数据的组件中去。然后，如果其他组件也需要这个 `state`，那么你可以将它提升至这些组件的最近共同父组件中。你应当依靠[自上而下的数据流](#)，而不是尝试在不同组件间同步 `state`。

虽然提升 state 方式比双向绑定方式需要编写更多的“样板”代码，但带来的好处是，排查和隔离 bug 所需的工作量将会变少。由于“存在”于组件中的任何 state，仅有组件自己能够修改它，因此 bug 的排查范围被大大缩减了。此外，你也可以使用自定义逻辑来拒绝或转换用户的输入。

如果某些数据可以由 props 或 state 推导得出，那么它就不应该存在于 state 中。举个例子，本例中我们没有将 `celsiusValue` 和 `fahrenheitValue` 一起保存，而是仅保存了最后修改的 `temperature` 和它的 `scale`。这是因为另一个输入框的温度值始终可以通过这两个值以及组件的 `render()` 方法获得。这使得我们能够清除输入框内容，亦或是，在不损失用户操作的输入框内数值精度的前提下对另一个输入框内的转换数值做四舍五入的操作。

当你在 UI 中发现错误时，可以使用 [React 开发者工具](#) 来检查问题组件的 props，并且按照组件树结构逐级向上搜寻，直到定位到负责更新 state 的那个组件。这使得你能够追踪到产生 bug 的源头：

The screenshot shows a user interface with two input fields. The first input field is labeled "Enter temperature in Celsius:" and contains a blue placeholder box. The second input field is labeled "Enter temperature in Fahrenheit:" and also contains a blue placeholder box.

The water would not boil.

The screenshot shows the React DevTools developer tool. The top navigation bar includes "Elements", "React" (which is selected), "Console", "Sources", "Network", "Timeline", "Profiles", and "More". Below the navigation are three checkboxes: "Trace React Updates", "Highlight Search", and "Use Regular Expressions". The main area displays the component tree under the heading "<Calculator> (\$r in the console)". The tree shows the following structure:

```

<Calculator>
  <div>
    ><TemperatureInput scale="c" temperature="" onTemperatureChange="..."/>
    ><TemperatureInput scale="f" temperature="" onTemperatureChange="..."/>
    ><BoilingVerdict celsius=null>The water would not boil.</BoilingVerdict>
  </div>
</Calculator>
  
```

On the right side, there are sections for "Props" and "State". The "Props" section shows an "Empty object". The "State" section shows "scale: "c"" and "temperature: """.

列表 & Key

首先，让我们看下在 Javascript 中如何转化列表。

如下代码，我们使用 `map()` 函数让数组中的每一项变双倍，然后我们得到了一个新的列表 `doubled` 并打印出来：

```
const numbers = [1, 2, 3, 4, 5];
const doubled = numbers.map((number) => number * 2);
console.log(doubled);
```

代码打印出 `[2, 4, 6, 8, 10]`。

在 React 中，把数组转化为元素列表的过程是相似的。

渲染多个组件

你可以通过使用 `{}` 在 JSX 内构建一个元素集合。

下面，我们使用 Javascript 中的 `map()` 方法来遍历 `numbers` 数组。将数组中的每个元素变成 `` 标签，最后我们将得到的数组赋值给 `listItems`：

```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) =>
  <li>{number}</li>
);
```

然后，我们可以将整个 `listItems` 插入到 `` 元素中：

```
<ul>{listItems}</ul>
```

[在 CodePen 上尝试](#)

这段代码生成了一个 1 到 5 的项目符号列表。

基础列表组件

通常你需要在一个组件中渲染列表。

我们可以把前面的例子重构为一个组件，这个组件接收 `numbers` 数组作为参数并输出一个元素列表。

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <li>{number}</li>
  );
  return (
    <ul>{listItems}</ul>
  );
}
```

```
const numbers = [1, 2, 3, 4, 5];
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<NumberList numbers={numbers} />);
```

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。我们将在下一节讨论这是为什么。

让我们来给每个列表元素分配一个 `key` 属性来解决上面的那个警告：

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <li key={number.toString()}>
      {number}
    </li>
  );
  return (
    <ul>{listItems}</ul>
  );
}
```

[在 CodePen 上尝试](#)

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。

```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) =>
  <li key={number.toString()}>
    {number}
  </li>
);
```

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`：

```
const todoItems = todos.map((todo) =>
  <li key={todo.id}>
    {todo.text}
  </li>
);
```

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`：

```
const todoItems = todos.map((todo, index) =>
  // Only do this if items have no stable IDs
  <li key={index}>
    {todo.text}
  </li>
);
```

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。可以看看 Robin Pokorny 的[深度解析使用索引作为 key 的负面影响](#)这一篇文章。如果你选择不指定显式的 `key` 值，那么 React 将默认使用索引用作为列表项目的 `key` 值。

要是你有兴趣了解更多的话，这里有一篇文章[深入解析为什么 key 是必须的](#)可以参考。

用 key 提取组件

元素的 key 只有放在就近的数组上下文中才有意义。

比方说，如果你提取出一个 `ListItem` 组件，你应该把 key 保留在数组中的这个 `<ListItem />` 元素上，而不是放在 `ListItem` 组件中的 `` 元素上。

例子：不正确的使用 key 的方式

```
function ListItem(props) {
  const value = props.value;
  return (
    // 错误！你不需要在这里指定 key:
    <li key={value.toString()}>
      {value}
    </li>
  );
}

function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    // 错误！元素的 key 应该在这里指定：
    <ListItem value={number} />
  );
  return (
    <ul>
      {listItems}
    </ul>
  );
}
```

例子：正确的使用 key 的方式

```
function ListItem(props) {
  // 正确！这里不需要指定 key:
  return <li>{props.value}</li>;
}

function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    // 正确！key 应该在数组的上下文中被指定
    <ListItem key={number.toString()} value={number} />
  );
  return (
    <ul>
      {listItems}
    </ul>
  );
}
```

[在 CodePen 上尝试](#)

一个好的经验法则是：在 `map()` 方法中的元素需要设置 key 属性。

key 值在兄弟节点之间必须唯一

数组元素中使用的 key 在其兄弟节点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 key 值：

```
function Blog(props) {
  const sidebar = (
    <ul>
      {props.posts.map((post) =>
        <li key={post.id}>
          {post.title}
        </li>
      )}
    </ul>
  );
  const content = props.posts.map((post) =>
    <div key={post.id}>
      <h3>{post.title}</h3>
      <p>{post.content}</p>
    </div>
  );
  return (
    <div>
      {sidebar}
      <hr />
      {content}
    </div>
  );
}

const posts = [
];

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Blog posts={posts} />);
```

[在 CodePen 上尝试](#)

key 会传递信息给 React，但不会传递给你的组件。如果你的组件中需要使用 key 属性的值，请用其他属性名显式传递这个值：

```
const content = posts.map((post) =>
  <Post
    key={post.id}
    id={post.id}
    title={post.title} />
);
```

上面例子中，Post 组件可以读出 props.id，但是不能读出 props.key。

在 JSX 中嵌入 map()

在上面的例子中，我们声明了一个单独的 listItems 变量并将其包含在 JSX 中：

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
```

```
<ListItem key={number.toString()}  
          value={number} />  
);  
return (  
  <ul>  
    {listItems}  
  </ul>  
)  
};  
}
```

JSX 允许在大括号中[嵌入任何表达式](#)，所以我们可以内联 `map()` 返回的结果：

```
function NumberList(props) {  
  const numbers = props.numbers;  
  return (  
    <ul>  
      {numbers.map((number) =>  
        <ListItem key={number.toString()}  
                  value={number} />  
      )}  
    </ul>  
}
```

[在 CodePen 上尝试](#)

这么做有时可以使你的代码更清晰，但有时这种风格也会被滥用。就像在 JavaScript 中一样，何时需要为了可读性提取出一个变量，这完全取决于你。但请记住，如果一个 `map()` 嵌套了太多层级，那可能就是你[提取组件](#)的一个好时机。

性能优化

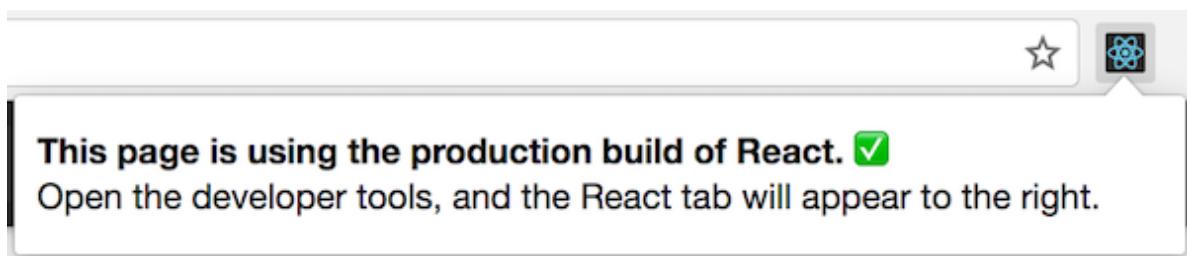
UI 更新需要昂贵的 DOM 操作，因此 React 内部使用了几种巧妙的技术来最小化 DOM 操作次数。对于大部分应用而言，使用 React 时无需做大量优化工作就能拥有高性能的用户界面。尽管如此，你仍然有办法来加速你的 React 应用。

使用生产版本

当你需要对你的 React 应用进行 benchmark，或者遇到了性能问题，请确保你正在使用压缩后的生产版本。

React 默认包含了许多有用的警告信息。这些警告信息在开发过程中非常有帮助。然而这使得 React 变得更大且更慢，所以你需要确保部署时使用了生产版本。

如果你不能确定你的编译过程是否设置正确，你可以通过安装 [Chrome 的 React 开发者工具](#) 来检查。如果你浏览一个基于 React 生产版本的网站，图标背景会变成深色：



如果你浏览一个基于 React 开发模式的网站，图标背景会变成红色：



推荐你在开发应用时使用开发模式，而在为用户部署应用时使用生产模式。

你可以在下面看到几种为应用构建生产版本的操作说明。

Create React App

如果你的项目是通过 [Create React App](#) 构建的，运行：

```
npm run build
```

这段命令将在你的项目下的 `build/` 目录中生成对应的生产版本。

注意只有在生产部署前才需要执行这个命令。正常开发使用 `npm start` 即可。

单文件构建

我们提供了可以在生产环境使用的单文件版 React 和 React DOM：

```
<script src="https://unpkg.com/react@18/umd/react.production.min.js"></script>
<script src="https://unpkg.com/react-dom@18/umd/react-dom.production.min.js">
</script>
```

注意只有以 `.production.min.js` 为结尾的 React 文件适用于生产。

Brunch

通过安装 `terser-brunch` 插件，来获得最高效的 Brunch 生产构建：

```
# 如果你使用 npm
npm install --save-dev terser-brunch

# 如果你使用 Yarn
yarn add --dev terser-brunch
```

接着，在 `build` 命令后添加 `-p` 参数，以创建生产构建：

```
brunch build -p
```

请注意，你只需要在生产构建时这么做。你不需要在开发环境中使用 `-p` 参数或者应用这个插件，因为这会隐藏有用的 React 警告信息并使得构建速度变慢。

Browserify

为了最高效的生产构建，需要安装一些插件：

```
# 如果你使用 npm
npm install --save-dev envify terser uglifyify

# 如果你使用 Yarn
yarn add --dev envify terser uglifyify
```

为了创建生产构建，确保你添加了以下转换器（顺序很重要）：

- `envify` 转换器用于设置正确的环境变量。设置为全局 (`-g`)。
- `uglifyify` 转换器移除开发相关的引用代码。同样设置为全局 (`-g`)。
- 最后，将产物传给 `terser` 并进行压缩（[为什么要这么做？](#)）。

举个例子：

```
browserify ./index.js \
-g [ envify --NODE_ENV production ] \
-g uglifyify \
| terser --compress --mangle > ./bundle.js
```

请注意，你只需要在生产构建时用到它。你不需要在开发环境应用这些插件，因为这会隐藏有用的 React 警告信息并使得构建速度变慢。

Rollup

为了高效的 Rollup 生产构建，需要安装一些插件：

```
# 如果你使用 npm
npm install --save-dev rollup-plugin-commonjs rollup-plugin-replace rollup-plugin-terser

# 如果你使用 Yarn
yarn add --dev rollup-plugin-commonjs rollup-plugin-replace rollup-plugin-terser
```

为了创建生产构建，确保你添加了以下插件 **(顺序很重要)**：

- `replace` 插件确保环境被正确设置。
- `commonjs` 插件用于支持 CommonJS。
- `terser` 插件用于压缩并生成最终的产物。

```
plugins: [
  // ...
  require('rollup-plugin-replace')({
    'process.env.NODE_ENV': JSON.stringify('production')
  }),
  require('rollup-plugin-commonjs')(),
  require('rollup-plugin-terser')(),
  // ...
]
```

[点击](#) 查看完整的安装示例。

请注意，你只需要在生产构建时用到它。你不需要在开发中使用 `terser` 插件或者 `replace` 插件替换 `'production'` 变量，因为这会隐藏有用的 React 警告信息并使得构建速度变慢。

webpack

注意：

如果你使用了 Create React App，请跟随[上面的说明](#)进行操作。

只有当你直接配置了 webpack 才需要参考以下内容。

在生产模式下，Webpack v4+ 将默认对代码进行压缩：

```
const TerserPlugin = require('terser-webpack-plugin');

module.exports = {
  mode: 'production',
  optimization: {
    minimizer: [new TerserPlugin({ /* additional options here */ })],
  },
};
```

你可以在 [webpack 文档](#) 中了解更多内容。

请注意，你只需要在生产构建时用到它。你不需要在开发中使用 `TerserPlugin` 插件，因为这会隐藏有用的 React 警告信息并使得构建速度变慢。

使用开发者工具中的分析器对组件进行分析

`react-dom` 16.5+ 和 `react-native` 0.57+ 加强了分析能力。在开发模式下，React 开发者工具会出现分析器标签。你可以在 [《介绍 React 分析器》](#) 这篇博客中了解概述。你也可以在 [YouTube 上](#) 观看分析器的视频指导。

如果你还未安装 React 开发者工具，你可以在这里找到它们：

- [Chrome 浏览器扩展](#)
- [Firefox 浏览器扩展](#)
- [独立 Node 包](#)

注意

`react-dom` 的生产分析包也可以在 `react-dom/profiling` 中找到。通过查阅 fb.me/react-profiling 来了解更多关于使用这个包的内容。

注意

在 React 17 之前，我们使用了标准的 [User Timing API](#)，用 chrome 的 performance 性能选项卡来配置组件。更详细的攻略，请参阅 [Ben Schwarz 的文章](#)。

虚拟化长列表

如果你的应用渲染了长列表（上百甚至上千的数据），我们推荐使用“虚拟滚动”技术。这项技术会在有限的时间内仅渲染有限的内容，并奇迹般地降低重新渲染组件消耗的时间，以及创建 DOM 节点的数量。

`react-window` 和 `react-virtualized` 是热门的虚拟滚动库。它们提供了多种可复用的组件，用于展示列表、网格和表格数据。如果你想要一些针对你的应用做定制优化，你也可以创建你自己的虚拟滚动组件，就像 [Twitter 所做的](#)。

避免调停

React 构建并维护了一套内部的 UI 渲染描述。它包含了来自你的组件返回的 React 元素。该描述使得 React 避免创建 DOM 节点以及不必要的节点访问，因为 DOM 操作相对于 JavaScript 对象操作更慢。虽然有时候它被称为“虚拟 DOM”，但是它在 React Native 中拥有相同的工作原理。

当一个组件的 `props` 或 `state` 变更，React 会将最新返回的元素与之前渲染的元素进行对比，以此决定是否有必要更新真实的 DOM。当它们不相同时，React 会更新该 DOM。

即使 React 只更新改变了的 DOM 节点，重新渲染仍然花费了一些时间。在大部分情况下它并不是问题，不过如果它已经慢到让人注意了，你可以通过覆盖生命周期方法 `shouldComponentUpdate` 来进行提速。该方法会在重新渲染前被触发。其默认实现总是返回 `true`，让 React 执行更新：

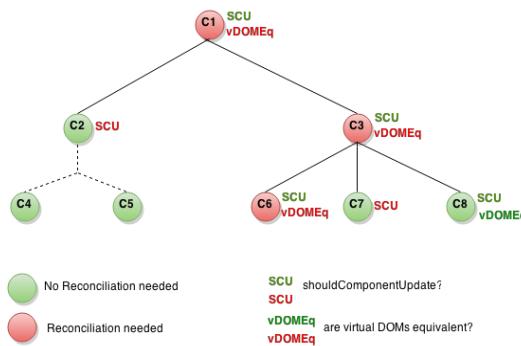
```
shouldComponentUpdate(nextProps, nextState) {
  return true;
}
```

如果你知道在什么情况下你的组件不需要更新，你可以在 `shouldComponentUpdate` 中返回 `false` 来跳过整个渲染过程。其包括该组件的 `render` 调用以及之后的操作。

在大部分情况下，你可以继承 `React.PureComponent` 以代替手写 `shouldComponentUpdate()`。它用当前与之前 `props` 和 `state` 的浅比较覆写了 `shouldComponentUpdate()` 的实现。

shouldComponentUpdate 的作用

这是一个组件的子树。每个节点中，`SCU` 代表 `shouldComponentUpdate` 返回的值，而 `vDOMEq` 代表返回的 React 元素是否相同。最后，圆圈的颜色代表了该组件是否需要被调停。



节点 C2 的 `shouldComponentUpdate` 返回了 `false`，React 因而不会去渲染 C2，也因此 C4 和 C5 的 `shouldComponentUpdate` 不会被调用到。

对于 C1 和 C3，`shouldComponentUpdate` 返回了 `true`，所以 React 需要继续向下查询子节点。这里 C6 的 `shouldComponentUpdate` 返回了 `true`，同时由于渲染的元素与之前的不同使得 React 更新了该 DOM。

最后一个有趣的例子是 C8。React 需要渲染这个组件，但是由于其返回的 React 元素和之前渲染的相同，所以不需要更新 DOM。

显而易见，你看到 React 只改变了 C6 的 DOM。对于 C8，通过对比了渲染的 React 元素跳过了渲染。而对于 C2 的子节点和 C7，由于 `shouldComponentUpdate` 使得 `render` 并没有被调用。因此它们也不需要对比元素了。

示例

如果你的组件只有当 `props.color` 或者 `state.count` 的值改变才需要更新时，你可以使用 `shouldComponentUpdate` 来进行检查：

```

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

  shouldComponentUpdate(nextProps, nextState) {
    if (this.props.color !== nextProps.color) {
      return true;
    }
    if (this.state.count !== nextState.count) {
      return true;
    }
    return false;
  }

  render() {
    return (
      <button
        color={this.props.color}
        onClick={() => this.setState(state => ({count: state.count + 1}))}>
        Count: {this.state.count}
      </button>
    );
  }
}

```

在这段代码中，`shouldComponentUpdate` 仅检查了 `props.color` 或 `state.count` 是否改变。如果这些值没有改变，那么这个组件不会更新。如果你的组件更复杂一些，你可以使用类似“浅比较”的模式来检查 `props` 和 `state` 中所有的字段，以此来决定是否组件需要更新。React 已经提供了一位好帮手来帮你实现这种常见的模式 - 你只要继承 `React.PureComponent` 就行了。所以这段代码可以改成以下这种更简洁的形式：

```

class CounterButton extends React.PureComponent {
  constructor(props) {
    super(props);
    this.state = {count: 1};
  }

  render() {
    return (
      <button
        color={this.props.color}
        onClick={() => this.setState(state => ({count: state.count + 1}))}>
        Count: {this.state.count}
      </button>
    );
  }
}

```

大部分情况下，你可以使用 `React.PureComponent` 来代替手写 `shouldComponentUpdate`。但它只进行浅比较，所以当 `props` 或者 `state` 某种程度是可变的话，浅比较会有遗漏，那你就不能使用它了。当数据结构很复杂时，情况会变得麻烦。例如，你想要一个 `ListOfWords` 组件来渲染一组用逗号分开的单词。它有一个叫做 `WordAdder` 的父组件，该组件允许你点击一个按钮来添加一个单词到列表中。以下代码并不正确：

```

class ListOfWords extends React.PureComponent {
  render() {
    return <div>{this.props.words.join(',')}</div>;
  }
}

```

```

}

class WordAdder extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      words: ['marklar']
    };
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    // 这部分代码很糟，而且还有 bug
    const words = this.state.words;
    words.push('marklar');
    this.setState({words: words});
  }

  render() {
    return (
      <div>
        <button onClick={this.handleClick} />
        <ListOfWords words={this.state.words} />
      </div>
    );
  }
}

```

问题在于 `PureComponent` 仅仅会对新老 `this.props.words` 的值进行简单的对比。由于代码中 `WordAdder` 的 `handleClick` 方法改变了同一个 `words` 数组，使得新老 `this.props.words` 比较的其实还是同一个数组。即便实际上数组中的单词已经变了，但是比较结果是相同的。可以看到，即便多了新的单词需要被渲染，`ListofWords` 却并没有被更新。

不可变数据的力量

避免该问题最简单的方式是避免更改你正用于 `props` 或 `state` 的值。例如，上面 `handleClick` 方法可以用 `concat` 重写：

```

handleClick() {
  this.setState(state => ({
    words: state.words.concat(['marklar'])
  }));
}

```

ES6 数组支持 [扩展运算符](#)，这让代码写起来更方便了。如果你在使用 Create React App，该语法已经默认支持了。

```

handleClick() {
  this.setState(state => ({
    words: [...state.words, 'marklar'],
  }));
}

```

你可以用类似的方式改写代码来避免可变对象的产生。例如，我们有一个叫做 `colormap` 的对象。我们希望写一个方法来将 `colormap.right` 设置为 `'blue'`。我们可以这么写：

```
function updateColorMap(colormap) {  
  colormap.right = 'blue';  
}
```

为了不改变原本的对象，我们可以使用 `Object.assign` 方法：

```
function updateColorMap(colormap) {  
  return Object.assign({}, colormap, {right: 'blue'});  
}
```

现在 `updateColorMap` 返回了一个新的对象，而不是修改老对象。`Object.assign` 是 ES6 的方法，需要 polyfill。

这里有一个 JavaScript 的提案，旨在添加[对象扩展属性](#)以使得更新不可变对象变得更方便：

```
function updateColorMap(colormap) {  
  return {...colormap, right: 'blue'};  
}
```

此特性已被收录在 JavaScript 的 ES2018 中。

如果你在使用 Create React App，`Object.assign` 以及对象扩展运算符已经默认支持了。

当处理深层嵌套对象时，以 `immutable`（不可变）的方式更新它们令人费解。如遇到此类问题，请参阅 [Immer](#) 或 [immutability-helper](#)。这些库会帮助你编写高可读性的代码，且不会失去 `immutability`（不可变性）带来的好处。

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。

`ReactDOM.createPortal(child, container)`

第一个参数 (`child`) 是任何可渲染的 React 子元素，例如一个元素，字符串或 fragment。第二个参数 (`container`) 是一个 DOM 元素。

用法

通常来讲，当你从组件的 `render` 方法返回一个元素时，该元素将被挂载到 DOM 节点中离其最近的父节点：

```
render() {
  // React 挂载了一个新的 div，并且把子元素渲染其中
  return (
    <div>
      {this.props.children}
    </div>
  );
}
```

然而，有时候将子元素插入到 DOM 节点中的不同位置也是有好处的：

```
render() {
  // React 并*没有*创建一个新的 div。它只是把子元素渲染到 `domNode` 中。
  // `domNode` 是一个可以在任何位置的有效 DOM 节点。
  return ReactDOM.createPortal(
    this.props.children,
    domNode
  );
}
```

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：

注意：

当在使用 portal 时，记住[管理键盘焦点](#)就变得尤为重要。

对于模态对话框，通过遵循 [WAI-ARIA 模态开发实践](#)，来确保每个人都能够运用它。

[在 CodePen 上尝试](#)

通过 Portal 进行事件冒泡

尽管 portal 可以被放置在 DOM 树中的任何地方，但在任何其他方面，其行为和普通的 React 子节点行为一致。由于 portal 仍存在于 React 树，且与 DOM 树中的位置无关，那么无论其子节点是否是 portal，像 context 这样的功能特性都是不变的。

这包含事件冒泡。一个从 portal 内部触发的事件会一直冒泡至包含 React 树的祖先，即便这些元素并不是 DOM 树中的祖先。假设存在如下 HTML 结构：

```
<html>
  <body>
    <div id="app-root"></div>
    <div id="modal-root"></div>
  </body>
</html>
```

在 `#app-root` 里的 `Parent` 组件能够捕获到未被捕获的从兄弟节点 `#modal-root` 冒泡上来的事件。

```
// 在 DOM 中有两个容器是兄弟级 (siblings)
const appRoot = document.getElementById('app-root');
const modalRoot = document.getElementById('modal-root');

class Modal extends React.Component {
  constructor(props) {
    super(props);
    this.el = document.createElement('div');
  }

  componentDidMount() {
    // 在 Modal 的所有子元素被挂载后,
    // 这个 portal 元素会被嵌入到 DOM 树中,
    // 这意味着子元素将被挂载到一个分离的 DOM 节点中。
    // 如果要求子组件在挂载时可以立刻接入 DOM 树,
    // 例如衡量一个 DOM 节点,
    // 或者在后代节点中使用 'autoFocus',
    // 则需添加 state 到 Modal 中,
    // 仅当 Modal 被插入 DOM 树中才能渲染子元素。
    modalRoot.appendChild(this.el);
  }

  componentWillUnmount() {
    modalRoot.removeChild(this.el);
  }

  render() {
    return ReactDOM.createPortal(
      this.props.children,
      this.el
    );
  }
}

class Parent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {clicks: 0};
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    console.log(`Clicked ${this.state.clicks} times`);
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>This is a React app.</h2>
        <button onClick={this.handleClick}>Click me</button>
      </div>
    );
  }
}
```

```

handleClick() {
  // 当子元素里的按钮被点击时,
  // 这个将会被触发更新父元素的 state,
  // 即使这个按钮在 DOM 中不是直接关联的后代
  this.setState(state => ({
    clicks: state.clicks + 1
  }));
}

render() {
  return (
    <div onClick={this.handleClick}>
      <p>Number of clicks: {this.state.clicks}</p>
      <p>
        Open up the browser DevTools
        to observe that the button
        is not a child of the div
        with the onClick handler.
      </p>
      <Modal>
        <Child />
      </Modal>
    </div>
  );
}

function Child() {
  // 这个按钮的点击事件会冒泡到父元素
  // 因为这里没有定义 'onClick' 属性
  return (
    <div className="modal">
      <button>Click</button>
    </div>
  );
}

const root = ReactDOM.createRoot(appRoot);
root.render(<Parent />);

```

在 CodePen 上尝试

在父组件里捕获一个来自 portal 冒泡上来的事件，使之能够在开发时具有不完全依赖于 portal 的更为灵活的抽象。例如，如果你在渲染一个 `<Modal />` 组件，无论其是否采用 portal 实现，父组件都能够捕获其事件。

不使用 ES6

通常我们会用 JavaScript 的 `class` 关键字来定义 React 组件：

```
class Greeting extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

如果你还未使用过 ES6，你可以使用 `create-react-class` 模块：

```
var createReactClass = require('create-react-class');
var Greeting = createReactClass({
  render: function() {
    return <h1>Hello, {this.props.name}</h1>;
  }
});
```

ES6 中的 `class` 与 `createReactClass()` 方法十分相似，但有以下几个区别值得注意。

声明默认属性

无论是函数组件还是 `class` 组件，都拥有 `defaultProps` 属性：

```
class Greeting extends React.Component {
  // ...
}

Greeting.defaultProps = {
  name: 'Mary'
};
```

如果使用 `createReactClass()` 方法创建组件，那就需要在组件中定义 `getDefaults()` 函数：

```
var Greeting = createReactClass({
  getDefaults: function() {
    return {
      name: 'Mary'
    };
  },
  // ...
});
```

初始化 State

如果使用 ES6 的 `class` 关键字创建组件，你可以通过给 `this.state` 赋值的方式来定义组件的初始 state：

```
class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {count: props.initialCount};
  }
  // ...
}
```

如果使用 `createReactClass()` 方法创建组件，你需要提供一个单独的 `getInitialState` 方法，让其返回初始 state：

```
var Counter = createReactClass({
  getInitialState: function() {
    return {count: this.props.initialCount};
  },
  // ...
});
```

自动绑定

对于使用 ES6 的 `class` 关键字创建的 React 组件，组件中的方法遵循与常规 ES6 class 相同的语法规则。这意味着这些方法不会自动绑定 `this` 到这个组件实例。你需要在 `constructor` 中显式地调用 `.bind(this)`：

```
class SayHello extends React.Component {
  constructor(props) {
    super(props);
    this.state = {message: 'Hello!'};
    // 这一行很重要!
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    alert(this.state.message);
  }

  render() {
    // 由于 `this.handleClick` 已经绑定至实例，因此我们才可以使用它来处理点击事件
    return (
      <button onClick={this.handleClick}>
        Say hello
      </button>
    );
  }
}
```

如果使用 `createReactClass()` 方法创建组件，组件中的方法会自动绑定至实例，所以不需要像上面那样做：

```
var SayHello = createReactClass({
  getInitialState: function() {
    return {message: 'Hello!'};
  },

  handleClick: function() {
    alert(this.state.message);
  },

  render: function() {
    return (

```

```

        <button onClick={this.handleClick}>
          Say hello
        </button>
      );
    }
  );
}

```

这就意味着，如果使用 ES6 class 关键字创建组件，在处理事件回调时就要多写一部分代码。但对于大型项目来说，这样做可以提升运行效率。

如果模板代码对你来说不太友好，你可以尝试使用 [ES2022 Class Properties](#) 语法：

```

class SayHello extends React.Component {
  constructor(props) {
    super(props);
    this.state = {message: 'Hello!'};
  }

  // 在这里使用箭头函数就可以把方法绑定给实例：
  handleClick = () => {
    alert(this.state.message);
  };

  render() {
    return (
      <button onClick={this.handleClick}>
        Say hello
      </button>
    );
  }
}

```

为了安全起见，你可以采用以下几种方式：

- 在 `constructor` 中绑定方法。
- 使用箭头函数，比如：`onClick={(e) => this.handleClick(e)}`。
- 继续使用 `createReactClass`。

Mixins

注意：

ES6 本身是不包含任何 mixin 支持。因此，当你在 React 中使用 ES6 class 时，将不支持 mixins。

我们也发现了很多使用 **mixins** 然后出现了问题的代码库。[并且不建议在新代码中使用它们。](#)

以下内容仅作为参考。

如果完全不同的组件有相似的功能，这就会产生“[横切关注点 \(cross-cutting concerns\)](#)”问题。针对这个问题，在使用 `createReactClass` 创建 React 组件的时候，引入 `mixins` 功能会是一个很好的解决方案。

比较常见的用法是，组件每隔一段时间更新一次。使用 `setInterval()` 可以很容易实现这个功能，但需要注意的是，当你不再需要它时，你应该清除它以节省内存。React 提供了[生命周期方法](#)，这样你就可以知道一个组件何时被创建或被销毁了。让我们创建一个简单的 mixin，它使用这些方法提供一个简单的 `setInterval()` 函数，它会在组件被销毁时被自动清理。

```
var SetIntervalMixin = {
  componentWillMount: function() {
    this.intervals = [];
  },
  setInterval: function() {
    this.intervals.push(setInterval.apply(null, arguments));
  },
  componentWillUnmount: function() {
    this.intervals.forEach(clearInterval);
  }
};

var createReactClass = require('create-react-class');

var TickTock = createReactClass({
  mixins: [SetIntervalMixin], // 使用 mixin
  getInitialState: function() {
    return {seconds: 0};
  },
  componentDidMount: function() {
    this.setInterval(this.tick, 1000); // 调用 mixin 上的方法
  },
  tick: function() {
    this.setState({seconds: this.state.seconds + 1});
  },
  render: function() {
    return (
      <p>
        React has been running for {this.state.seconds} seconds.
      </p>
    );
  }
});

const root = ReactDOM.createRoot(document.getElementById('example'));
root.render(<TickTock />);
```

如果组件拥有多个 mixins，且这些 mixins 中定义了相同的生命周期方法（例如，当组件被销毁时，几个 mixins 都想要进行一些清理工作），那么这些生命周期方法都会被调用的。使用 mixins 时，mixins 会先按照定义时的顺序执行，最后调用组件上对应的方法。

不使用 JSX 的 React

React 并不强制要求使用 JSX。当你不想在构建环境中配置有关 JSX 编译时，不在 React 中使用 JSX 会更加方便。

每个 JSX 元素只是调用 `React.createElement(component, props, ...children)` 的语法糖。因此，使用 JSX 可以完成的任何事情都可以通过纯 JavaScript 完成。

例如，用 JSX 编写的代码：

```
class Hello extends React.Component {
  render() {
    return <div>Hello {this.props.toWhat}</div>;
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Hello toWhat="World" />);
```

可以编写为不使用 JSX 的代码：

```
class Hello extends React.Component {
  render() {
    return React.createElement('div', null, `Hello ${this.props.toWhat}`);
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(React.createElement(Hello, {toWhat: 'World'}, null));
```

如果你想了解更多 JSX 转换为 JavaScript 的示例，可以尝试使用 [在线 Babel 编译器](#)。

组件可以是字符串，也可以是 `React.Component` 的子类，它还能是一个普通的函数。

如果你不想每次都键入 `React.createElement`，通常的做法是创建快捷方式：

```
const e = React.createElement;

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(e('div', null, 'Hello World'));
```

如果你使用了 `React.createElement` 的快捷方式，那么在没有 JSX 的情况下使用 React 几乎一样方便。

或者，你也可以参考社区项目，如：`react-hyperscript` 和 `hyperscript-helpers`，它们提供了更简洁的语法。

协调

React 提供的声明式 API 让开发者可以在对 React 的底层实现并不了解的情况下编写应用。在开发者编写应用时，可以保持相对简单的心智，但开发者无法了解其内部的实现原理。本文描述了在实现 React 的 "diffing" 算法过程中所作出的设计决策，以保证组件更新可预测，且在繁杂业务场景下依然保持应用的高性能。

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](#)，该算法的复杂程度仍为 $O(n^3)$ ，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. 开发者可以使用 `key` 属性标识哪些子元素在不同的渲染中可能是不变的。

在实践中，我们发现以上假设在几乎所有实用的场景下都成立。

Diffing 算法

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。

对比不同类型的元素

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 ``，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：

```
<div>
  <Counter />
</div>

<span>
  <Counter />
</span>
```

React 会销毁 `Counter` 组件并且重新装载一个新的组件。

注意：

这些方法被认为是过时的，在新的代码中应该[避免使用它们](#)：

- `UNSAFE_componentWillMount()`

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：

```
<div className="before" title="stuff" />
<div className="after" title="stuff" />
```

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：

```
<div style={{color: 'red', fontWeight: 'bold'}} />
<div style={{color: 'green', fontWeight: 'bold'}} />
```

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。

注意：

这些方法已过时，在新代码中应[避免使用它们](#)：

- `UNSAFE_componentWillUpdate()`
- `UNSAFE_componentWillReceiveProps()`

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：

```
<ul>
  <li>first</li>
  <li>second</li>
```

```
</ul>
<ul>
  <li>first</li>
  <li>second</li>
  <li>third</li>
</ul>
```

React 会先匹配两个 `first` 对应的树，然后匹配第二个元素 `second` 对应的树，最后插入第三个元素的 `third` 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：

```
<ul>
  <li>Duke</li>
  <li>Villanova</li>
</ul>

<ul>
  <li>Connecticut</li>
  <li>Duke</li>
  <li>Villanova</li>
</ul>
```

React 并不会意识到应该保留 `Duke` 和 `Villanova`，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：

```
<ul>
  <li key="2015">Duke</li>
  <li key="2016">Villanova</li>
</ul>

<ul>
  <li key="2014">Connecticut</li>
  <li key="2015">Duke</li>
  <li key="2016">Villanova</li>
</ul>
```

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。

实际开发中，编写一个 `key` 并不困难。你要展现的元素可能已经有了一个唯一 ID，于是 `key` 可以直接从你的数据中提取：

```
<li key={item.id}>{item.name}</li>
```

当以上情况不成立时，你可以新增一个 `ID` 字段到你的模型中，或者利用一部分内容作为哈希值来生成一个 `key`。这个 `key` 不需要全局唯一，但在列表中需要保持唯一。

最后，你也可以使用元素在数组中的下标作为 `key`。这个策略在元素不进行重新排序时比较合适，如果有顺序修改，`diff` 就会变慢。

当基于下标的组件进行重新排序时，组件 state 可能会遇到一些问题。由于组件实例是基于它们的 key 来决定是否更新以及复用，如果 key 是一个下标，那么修改顺序时会修改当前的 key，导致非受控组件的 state（比如输入框）可能相互篡改，会出现无法预期的变动。

在 Codepen 有两个例子，分别为 [展示使用下标作为 key 时导致的问题](#)，以及[不使用下标作为 key 的例子的版本，修复了重新排列，排序，以及在列表头插入的问题](#)。

权衡

请谨记协调算法是一个实现细节。React 可以在每个 action 之后对整个应用进行重新渲染，得到的最终结果也会是一样的。在此情境下，重新渲染表示在所有组件内调用 `render` 方法，这不代表 React 会卸载或装载它们。React 只会基于以上提到的规则来决定如何进行差异的合并。

我们定期优化启发式算法，让常见用例更高效地执行。在当前的实现中，可以理解为一棵子树能在其兄弟之间移动，但不能移动到其他位置。在这种情况下，算法会重新渲染整棵子树。

由于 React 依赖启发式算法，因此当以下假设没有得到满足，性能会有所损耗。

1. 该算法不会尝试匹配不同组件类型的子树。如果你发现在两种不同类型的组件中切换，但输出非常相似的内容，建议把它们改成同一类型。在实践中，我们没有遇到这类问题。
2. Key 应该具有稳定，可预测，以及列表内唯一的特质。不稳定的 key（比如通过 `Math.random()` 生成的）会导致许多组件实例和 DOM 节点被不必要地重新创建，这可能导致性能下降和子组件中的状态丢失。

DOM 元素

React 实现了一套独立于浏览器的 DOM 系统，兼顾了性能和跨浏览器的兼容性。我们借此机会完善了浏览器 DOM 实现的一些特殊情况。

在 React 中，所有的 DOM 特性和属性（包括事件处理）都应该是小驼峰命名的方式。例如，与 HTML 中的 `tabindex` 属性对应的 React 的属性是 `tabIndex`。例外的情况是 `aria-*` 以及 `data-*` 属性，一律使用小写字母命名。比如，你依然可以用 `aria-label` 作为 `aria-label`。

属性差异

React 与 HTML 之间有很多属性存在差异：

checked

当 `<input>` 组件的 `type` 类型为 `checkbox` 或 `radio` 时，组件支持 `checked` 属性。你可以使用它来设置组件是否被选中。这对于构建受控组件（controlled components）很有帮助。而 `defaultChecked` 则是非受控组件的属性，用于设置组件首次挂载时是否被选中。

className

`className` 属性用于指定 CSS 的 `class`，此特性适用于所有常规 DOM 节点和 SVG 元素，如 `<div>`，`<a>` 及其它标签。

如果你在 React 中使用 Web Components（这是一种不常见的使用方式），请使用 `class` 属性代替。

dangerouslySetInnerHTML

`dangerouslySetInnerHTML` 是 React 为浏览器 DOM 提供 `innerHTML` 的替换方案。通常来讲，使用代码直接设置 HTML 存在风险，因为很容易无意中使用户暴露于跨站脚本（XSS）的攻击。因此，你可以直接在 React 中设置 HTML，但当你想设置 `dangerouslySetInnerHTML` 时，需要向其传递包含 `key` 为 `__html` 的对象，以此来警示你。例如：

```
function createMarkup() {
  return {__html: 'First &nbsp; Second'};
}

function MyComponent() {
  return <div dangerouslySetInnerHTML={createMarkup()} />;
}
```

htmlFor

由于 `for` 在 JavaScript 中是保留字，所以 React 元素中使用了 `htmlFor` 来代替。

onChange

`onChange` 事件与预期行为一致：每当表单字段变化时，该事件都会被触发。我们故意没有使用浏览器已有的默认行为，是因为 `onChange` 在浏览器中的行为和名称不对称，并且 React 依靠该事件实时处理用户输入。

selected

如果要将 `<option>` 标记为已选中状态, 请在 `select` 的 `value` 中引用该选项的值。请查阅 "[select 标签](#)" 了解更多详情。

style

注意

在文档中, 部分例子为了方便, 直接使用了 `style`, 但是通常不推荐将 `style` 属性作为设置元素样式的主要方式。在多数情况下, 应使用 `className` 属性来引用外部 CSS 样式表中定义的 class。`style` 在 React 应用中多用于在渲染过程中添加动态计算的样式。另请参阅：[FAQ : Styling 和 CSS](#)。

`style` 接受一个采用小驼峰命名属性的 JavaScript 对象, 而不是 CSS 字符串。这与 DOM 中 `style` 的 JavaScript 属性是一致的, 同时会更高效的, 且能预防跨站脚本 (XSS) 的安全漏洞。例如：

```
const divStyle = {
  color: 'blue',
  backgroundImage: 'url(' + imgUrl + ')',
};

function HelloWorldComponent() {
  return <div style={divStyle}>Hello World!</div>;
}
```

注意：样式不会自动补齐前缀。如需支持旧版浏览器, 请手动补充对应的样式属性：

```
const divStyle = {
  WebkitTransition: 'all', // note the capital 'W' here
  msTransition: 'all' // 'ms' is the only lowercase vendor prefix
};

function ComponentWithTransition() {
  return <div style={divStyle}>This should work cross-browser</div>;
}
```

Style 中的 key 采用小驼峰命名是为了与 JS 访问 DOM 节点的属性保持一致 (例如：`node.style.backgroundImage`)。浏览器引擎前缀都应以大写字母开头, 除了 `ms`。因此, `WebkitTransition` 首字母为 "W"。

React 会自动添加 "px" 后缀到内联样式为数字的属性后。如需使用 "px" 以外的单位, 请将此值设为数字与所需单位组成的字符串。例如：

```
// Result style: '10px'
<div style={{ height: 10 }}>
  Hello World!
</div>

// Result style: '10%'
<div style={{ height: '10%' }}>
  Hello World!
</div>
```

但并非所有样式属性都转换为像素字符串。有些样式属性是没有单位的(例如 `zoom`, `order`, `flex`)。无单位属性的完整列表在此处。

suppressContentEditableWarning

通常, 当拥有子节点的元素被标记为 `contentEditable` 时, React 会发出一个警告, 因为这不会生效。该属性将禁止此警告。尽量不要使用该属性, 除非你要构建一个类似 `Draft.js` 的手动管理 `contentEditable` 属性的库。

suppressHydrationWarning

如果你使用 React 服务端渲染, 通常会在当服务端与客户端渲染不同的内容时发出警告。但是, 在一些极少数的情况下, 很难甚至于不可能保证内容的一致性。例如, 在服务端和客户端上, 时间戳通常是不同的。

如果设置 `suppressHydrationWarning` 为 `true`, React 将不会警告你属性与元素内容不一致。它只会对元素一级深度有效, 并且打算作为应急方案。因此不要过度使用它。你可以在 `ReactDOM.hydrateRoot()` 文档 中了解更多关于 `hydration` 的信息。

value

`<input>`、`<select>` 和 `<textarea>` 组件支持 `value` 属性。你可以使用它为组件设置 `value`。这对于构建受控组件是非常有帮助。`defaultValue` 属性对应的是非受控组件的属性, 用于设置组件第一次挂载时的 `value`。

All Supported HTML Attributes

在 React 16 中, 任何标准的或自定义的 DOM 属性都是完全支持的。

React 为 DOM 提供了一套以 JavaScript 为中心的 API。由于 React 组件经常采用自定义或和 DOM 相关的 props 的关系, React 采用了小驼峰命名的方式, 正如 DOM APIs 那样 :

```
<div tabIndex={-1} /> // Just like node.tabIndex DOM API
<div className="Button" /> // Just like node.className DOM API
<input readOnly={true} /> // Just like node.readOnly DOM API
```

除了上述文档提到的特殊拼写方式以外, 这些 props 的用法与 HTML 的属性也极为类似。

React 支持的 DOM 属性有 :

```
accept acceptCharset accessKey action allowFullScreen alt autoComplete
autoFocus autoPlay capture cellPadding cellSpacing challenge charSet checked
cite classID className colSpan cols content contentEditable contextMenu controls
controlsList coords crossOrigin data dateTime default defer dir disabled
download draggable encType form formAction formEncType formMethod formNoValidate
formTarget frameBorder headers height hidden high href hrefLang htmlFor
httpEquiv icon id inputMode integrity is keyParams keyType kind label lang list
loop low manifest marginHeight marginWidth max maxLength media mediaGroup method
min minLength multiple muted name noValidate nonce open optimum pattern
placeholder poster preload profile radioGroup readOnly rel required reversed
role rowSpan rows sandbox scope scoped scrolling seamless selected shape size
sizes span spellCheck src srcDoc srcLang srcSet start step style summary
tabIndex target title type useMap value width wmode wrap
```

同样, 所有的 SVG 属性也完全得到了支持 :

```
accentHeight accumulate additive alignmentBaseline allowReorder alphabetic
amplitude arabicForm ascent attributeName attributeType autoReverse azimuth
baseFrequency baseProfile baselineShift bbox begin bias by calcMode capHeight
clip clipPath clipPathUnits clipRule colorInterpolation
colorInterpolationFilters colorProfile colorRendering contentScriptType
contentStyleType cursor cx cy d decelerate descent diffuseConstant direction
display divisor dominantBaseline dur dx dy edgeMode elevation enableBackground
end exponent externalResourcesRequired fill fillOpacity fillRule filter
filterRes filterUnits floodColor floodOpacity focusable fontFamily fontSize
fontSizeAdjust fontStretch fontStyle fontVariant fontWeight format from fx fy
g1 g2 glyphName glyphOrientationHorizontal glyphOrientationVertical glyphRef
gradientTransform gradientUnits hanging horizAdvX horizOriginX ideographic
imageRendering in in2 intercept k k1 k2 k3 k4 kernelMatrix kernelUnitLength
kerning keyPoints keySplines keyTimes lengthAdjust letterSpacing lightingColor
limitingConeAngle local markerEnd markerHeight markerMid markerStart
markerUnits markerWidth mask maskContentUnits maskUnits mathematical mode
numOctaves offset opacity operator order orient orientation origin overflow
overlinePosition overlineThickness paintOrder panose1 pathLength
patternContentUnits patternTransform patternUnits pointerEvents points
pointsAtX pointsAtY pointsAtZ preserveAlpha preserveAspectRatio primitiveUnits
r radius refX refY renderingIntent repeatCount repeatDur requiredExtensions
requiredFeatures restart result rotate rx ry scale seed shapeRendering slope
spacing specularConstant specularExponent speed spreadMethod startOffset
stdDeviation stemh stemv stitchTiles stopColor stopOpacity
strikethroughPosition strikethroughThickness string stroke strokeDasharray
strokeDashoffset strokeLinecap strokeLinejoin strokeMiterlimit strokeOpacity
strokeWidth surfaceScale systemLanguage tableValues targetX targetY textAnchor
textDecoration textLength textRendering to transform u1 u2 underlinePosition
underlineThickness unicode unicodeBidi unicodeRange unitsPerEm vAlphabetic
vHanging vIdeographic vMathematical values vectorEffect version vertAdvY
vertOriginX vertOriginY viewBox viewTarget visibility widths wordSpacing
writingMode x x1 x2 xChannelSelector xHeight xlinkActuate xlinkArcrole
xlinkHref xlinkRole xlinkShow xlinkTitle xlinkType xmlns xmlnsXlink xmlBase
xmlLang xmlSpace y y1 y2 yChannelSelector z zoomAndPan
```

你也可以使用自定义属性，但要注意属性名全都为小写。

合成事件

此参考指南记录了构成 React 事件系统一部分的 `SyntheticEvent` 包装器。请参考有关[事件处理](#)的指南来了解更多。

概览

`SyntheticEvent` 实例将被传递给你的事件处理函数，它是浏览器的原生事件的跨浏览器包装器。除兼容所有浏览器外，它还拥有和浏览器原生事件相同的接口，包括 `stopPropagation()` 和 `preventDefault()`。

如果因为某些原因，当你需要使用浏览器的底层事件时，只需要使用 `nativeEvent` 属性来获取即可。合成事件与浏览器的原生事件不同，也不会直接映射到原生事件。例如，在 `onMouseLeave` 事件中 `event.nativeEvent` 将指向 `mouseout` 事件。每个 `SyntheticEvent` 对象都包含以下属性：

```
boolean bubbles
boolean cancelable
DOMEventTarget currentTarget
boolean defaultPrevented
number eventPhase
boolean isTrusted
DOMEvent nativeEvent
void preventDefault()
boolean isDefaultPrevented()
void stopPropagation()
boolean isPropagationStopped()
void persist()
DOMEventTarget target
number timeStamp
string type
```

注意：

从 v17 开始，`e.persist()` 将不再生效，因为 `SyntheticEvent` 不再放入[事件池](#)中。

注意：

从 v0.14 开始，事件处理器返回 `false` 时，不再阻止事件传递。你可以酌情手动调用 `e.stopPropagation()` 或 `e.preventDefault()` 作为替代方案。

支持的事件

React 通过将事件 normalize 以让他们在不同浏览器中拥有一致的属性。

以下的事件处理函数在冒泡阶段被触发。如需注册捕获阶段的事件处理函数，则应为事件名添加 `Capture`。例如，处理捕获阶段的点击事件请使用 `onClickCapture`，而不是 `onClick`。

- [Clipboard Events](#)
 - [Composition Events](#)
 - [Keyboard Events](#)
 - [Focus Events](#)
 - [Form Events](#)
 - [Generic Events](#)
 - [Mouse Events](#)
 - [Pointer Events](#)
 - [Selection Events](#)
 - [Touch Events](#)
 - [UI Events](#)
 - [Wheel Events](#)
 - [Media Events](#)
 - [Image Events](#)
 - [Animation Events](#)
 - [Transition Events](#)
 - [Other Events](#)
-

参考

剪贴板事件

事件名：

`onCopy onCut onPaste`

属性：

`DOMDataTransfer clipboardData`

复合事件

事件名:

`onCompositionEnd onCompositionStart onCompositionUpdate`

属性:

`string data`

键盘事件

事件名:

`onKeyDown onKeyPress onKeyUp`

属性：

```
boolean altKey
number charCode
boolean ctrlKey
boolean getModifierState(key)
string key
number keyCode
string locale
number location
boolean metaKey
boolean repeat
boolean shiftKey
number which
```

`key` 属性可以是 [DOM Level 3 Events spec](#) 里记录的任意值。

焦点事件

事件名：

`onFocus onBlur`

这些焦点事件在 React DOM 上的所有元素都有效，不只是表单元素。

属性：

`DOMEventTarget relatedTarget`

onFocus

`onFocus` 事件在元素（或其内部某些元素）聚焦时被调用。例如，当用户点击文本输入框时，就会调用该事件。

```
function Example() {
  return (
    <input
      onFocus={(e) => {
        console.log('Focused on input');
      }}
      placeholder="onFocus is triggered when you click this input."
    />
  )
}
```

onBlur

`onBlur` 事件处理程序在元素（或元素内某些元素）失去焦点时被调用。例如，当用户在已聚焦的文本输入框外点击时，就会被调用。

```
function Example() {
  return (
    <input
      onBlur={(e) => {
        console.log('Triggered because this input lost focus');
      }}
    />
  )
}
```

```

    placeholder="onBlur is triggered when you click this input and then you
    click outside of it."
    />
)
}

```

监听焦点的进入与离开

你可以使用 `currentTarget` 和 `relatedTarget` 来区分聚焦和失去焦点是否来自父元素外部。这里有个 DEMO，你可以复制并在本地运行，它展示了如何监听一个子元素的聚焦，元素本身的聚焦，以及整个子树进入焦点或离开焦点。

```

function Example() {
  return (
    <div
      tabIndex={1}
      onFocus={(e) => {
        if (e.currentTarget === e.target) {
          console.log('focused self');
        } else {
          console.log('focused child', e.target);
        }
        if (!e.currentTarget.contains(e.relatedTarget)) {
          // Not triggered when swapping focus between children
          console.log('focus entered self');
        }
      }}
      onBlur={(e) => {
        if (e.currentTarget === e.target) {
          console.log('unfocused self');
        } else {
          console.log('unfocused child', e.target);
        }
        if (!e.currentTarget.contains(e.relatedTarget)) {
          // Not triggered when swapping focus between children
          console.log('focus left self');
        }
      }}
    >
      <input id="1" />
      <input id="2" />
    </div>
  );
}

```

表单事件

事件名：

`onChange` `onInput` `onInvalid` `onReset` `onSubmit`

想了解 `onChange` 事件的更多信息，查看 [Forms](#)。

通用事件

事件名：

onError onLoad

鼠标事件

事件名：

onClick onContextMenu onDoubleClick onDrag onDragEnd onDragEnter onDragExit
onDragLeave onDragOver onDragStart onDrop onMouseDown onMouseEnter onMouseLeave
onMouseMove onMouseOut onMouseOver onMouseUp

`onMouseEnter` 和 `onMouseLeave` 事件从离开的元素向进入的元素传播，不是正常的冒泡，也没有捕获阶段。

属性：

```
boolean altKey
number button
number buttons
number clientX
number clientY
boolean ctrlKey
boolean getModifierState(key)
boolean metaKey
number pageX
number pageY
DOMEventTarget relatedTarget
number screenX
number screenY
boolean shiftKey
```

指针事件

事件名：

onPointerDown onPointerMove onPointerUp onPointerCancel onGotPointerCapture
onLostPointerCapture onPointerEnter onPointerLeave onPointerOver onPointerOut

`onPointerEnter` 和 `onPointerLeave` 事件从离开的元素向进入的元素传播，不是正常的冒泡，也没有捕获阶段。

属性：

如 W3 spec 中定义的，指针事件通过以下属性扩展了鼠标事件：

```
number pointerId
number width
number height
number pressure
number tangentialPressure
number tiltX
number tiltY
number twist
string pointerType
boolean isPrimary
```

关于跨浏览器支持的说明：

并非每个浏览器都支持指针事件（在写这篇文章时，已支持的浏览器有：Chrome, Firefox, Edge 和 Internet Explorer）。React 故意不通过 polyfill 的方式适配其他浏览器，主要是符合标准的 polyfill 会显著增加 react-dom 的 bundle 大小。

如果你的应用要求指针事件，我们推荐添加第三方的指针事件 polyfill。

选择事件

事件名：

onSelect

触摸事件

事件名：

onTouchCancel onTouchEnd onTouchMove onTouchStart

属性：

```
boolean altKey
DOMTouchList changedTouches
boolean ctrlKey
boolean getModifierState(key)
boolean metaKey
boolean shiftKey
DOMTouchList targetTouches
DOMTouchList touches
```

UI 事件

事件名：

onScroll

注意

从 React 17 开始，onScroll 事件在 React 中不再冒泡。这与浏览器的行为一致，并且避免了当一个嵌套且可滚动的元素在其父元素触发事件时造成混乱。

属性：

```
number detail
DOMAbstractView view
```

滚轮事件

事件名：

onWheel

属性：

```
number deltaMode  
number deltaX  
number deltaY  
number deltaZ
```

媒体事件

事件名：

```
onAbort onCanPlay onCanPlayThrough onDurationChange onEmptied onEncrypted  
onEnded onError onLoadedData onLoadedMetadata onLoadStart onPause onPlay  
onPlaying onProgress onRateChange onSeeked onSeeking onStalled onSuspend  
onTimeUpdate onVolumeChange onWaiting
```

图像事件

事件名：

onLoad onError

动画事件

事件名：

onAnimationStart onAnimationEnd onAnimationIteration

属性：

```
string animationName  
string pseudoElement  
float elapsedTime
```

过渡事件

事件名：

onTransitionEnd

属性：

```
string propertyName  
string pseudoElement  
float elapsedTime
```

其他事件

事件名：

```
onToggle
```

[Go to TOC](#)

React 术语词汇表

单页面应用

单页面应用(single-page application)，是一个应用程序，它可以加载单个 HTML 页面，以及运行应用程序所需的所有必要资源（例如 JavaScript 和 CSS）。与页面或后续页面的任何交互，都不再需要往返 server 加载资源，即页面不会重新加载。

你可以使用 React 来构建单页应用程序，但不是必须如此。React 还可用于增强现有网站的小部分，使其增加额外交互。用 React 编写的代码，可以与服务器端渲染的标记（例如 PHP）或其他客户端库和平共处。实际上，这也正是 Facebook 内部使用 React 的方式。

ES6, ES2015, ES2016 等

这些首字母缩写都是指 ECMAScript 语言规范标准的最新版本，JavaScript 语言是此标准的一个实现。其中 ES6 版本（也称为 ES2015）包括对前面版本的许多补充，例如：箭头函数、class、模板字面量、`let` 和 `const` 语句。可以在[这里](#)了解此规范特定版本的详细信息。

Compiler(编译器)

JavaScript compiler 接收 JavaScript 代码，然后对其进行转换，最终返回不同格式的 JavaScript 代码。最为常见的使用示例是，接收 ES6 语法，然后将其转换为旧版本浏览器能够解释执行的语法。[Babel](#) 是 React 最常用的 compiler。

Bundler(打包工具)

bundler 会接收写成单独模块（通常有数百个）的 JavaScript 和 CSS 代码，然后将它们组合在一起，最终生成出一些为浏览器优化的文件。常用的打包 React 应用的工具有 [webpack](#) 和 [Browserify](#)。

Package 管理工具

package 管理工具，是帮助你管理项目依赖的工具。[npm](#) 和 [Yarn](#) 是两个常用的管理 React 应用依赖的 package 管理工具。它们都是使用了相同 npm package registry 的客户端。

CDN

CDN 代表内容分发网络（Content Delivery Network）。CDN 会通过一个遍布全球的服务器网络来分发缓存的静态内容。

JSX

JSX 是一个 JavaScript 语法扩展。它类似于模板语言，但它具有 JavaScript 的全部能力。JSX 最终会被编译为 `React.createElement()` 函数调用，返回称为“React 元素”的普通 JavaScript 对象。通过[查看这篇文档](#)获取 JSX 语法的基本介绍，在[这篇文档](#)中可以找到 JSX 语法的更多深入教程。

React DOM 使用 camelCase (驼峰式命名) 来定义属性的名称，而不使用 HTML 属性名称的命名约定。例如，HTML 的 `tabindex` 属性变成了 JSX 的 `tabIndex`。而 `class` 属性则变为 `className`，这是因为 `class` 是 JavaScript 中的保留字：

```
<h1 className="hello">My name is Clementine!</h1>
```

元素 {#elements}

React 元素是构成 React 应用的基础砖块。人们可能会把元素与广为人知的“组件”概念相互混淆。元素描述了你在屏幕上想看到的内容。React 元素是不可变对象。

```
const element = <h1>Hello, world</h1>;
```

通常我们不会直接使用元素，而是从组件中返回元素。

组件 {#components}

React 组件是可复用的小的代码片段，它们返回要在页面中渲染的 React 元素。React 组件的最简版本是，一个返回 React 元素的普通 JavaScript 函数：

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

组件也可以使用 ES6 的 class 编写：

```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

组件可被拆分为不同的功能片段，这些片段可以在其他组件中使用。组件可以返回其他组件、数组、字符串和数字。根据经验来看，如果 UI 中有一部分被多次使用 (Button, Panel, Avatar)，或者组件本身就足够复杂 (App, FeedStory, Comment)，那么它就是一个可复用组件的候选项。组件名称应该始终以大写字母开头 (`<Wrapper/>` 而不是 `<wrapper/>`)。有关渲染组件的更多信息，请参阅[这篇文章档](#)。

props {#props}

`props` 是 React 组件的输入。它们是从父组件向下传递给子组件的数据。

记住，`props` 是只读的。不应以任何方式修改它们：

```
// 错误做法!
props.number = 42;
```

如果你想要修改某些值，以响应用户输入或网络响应，请使用 `state` 来作为替代。

props.children {#propschildren}

每个组件都可以获取到 `props.children`。它包含组件的开始标签和结束标签之间的内容。例如：

```
<Welcome>Hello world!</Welcome>
```

在 `Welcome` 组件中获取 `props.children`，就可以得到字符串 `Hello world!`：

```
function Welcome(props) {
  return <p>{props.children}</p>;
}
```

对于 class 组件，请使用 `this.props.children` 来获取：

```
class Welcome extends React.Component {
  render() {
    return <p>{this.props.children}</p>;
  }
}
```

state {#state}

当组件中的一些数据在某些时刻发生变化时，这时就需要使用 `state` 来跟踪状态。例如，`Checkbox` 组件可能需要 `isChecked` 状态，而 `NewsFeed` 组件可能需要跟踪 `fetchedPosts` 状态。

`state` 和 `props` 之间最重要的区别是：`props` 由父组件传入，而 `state` 由组件本身管理。组件不能修改 `props`，但它可以修改 `state`。

对于所有变化数据中的每个特定部分，只应该由一个组件在其 `state` 中“持有”它。不要试图同步来自于两个不同组件的 `state`。相反，应当将其提升到最近的共同祖先组件中，并将这个 `state` 作为 `props` 传递到两个子组件。

生命周期方法 {#lifecycle-methods}

生命周期方法，用于在组件不同阶段执行自定义功能。在组件被创建并插入到 DOM 时（即挂载中阶段（mounting）），组件更新时，组件取消挂载或从 DOM 中删除时，都有可以使用的生命周期方法。

受控组件 vs 非受控组件

React 有两种不同的方式来处理表单输入。

如果一个 `input` 表单元素的值是由 React 控制，就其称为受控组件。当用户将数据输入到受控组件时，会触发修改状态的事件处理器，这时由你的代码来决定此输入是否有效（如果有效就使用更新后的值重新渲染）。如果不重新渲染，则表单元素将保持不变。

一个非受控组件，就像是运行在 React 体系之外的表单元素。当用户将数据输入到表单字段（例如 `input`, `dropdown` 等）时，React 不需要做任何事情就可以映射更新后的信息。然而，这也意味着，你无法强制给这个表单字段设置一个特定值。

在大多数情况下，你应该使用受控组件。

key {#keys}

“key”是在创建元素数组时，需要用到的一个特殊字符串属性。key 帮助 React 识别出被修改、添加或删除的 item。应当给数组内的每个元素都设定 key，以使元素具有固定身份标识。

只需要保证，在同一个数组中的兄弟元素之间的 key 是唯一的。而不需要在整个应用程序甚至单个组件中保持唯一。

不要将 `Math.random()` 之类的值传递给 key。重要的是，在前后两次渲染之间的 key 要具有“固定身份标识”的特点，以便 React 可以在添加、删除或重新排序 item 时，前后对应起来。理想情况下，key 应该从数据中获取，对应着唯一且固定的标识符，例如 `post.id`。

Ref {#refs}

React 支持一个特殊的、可以附加到任何组件上的 `ref` 属性。此属性可以是一个由 `React.createRef()` 函数创建的对象、或者一个回调函数、或者一个字符串（遗留 API）。当 `ref` 属性是一个回调函数时，此函数会（根据元素的类型）接收底层 DOM 元素或 class 实例作为其参数。这能够让你直接访问 DOM 元素或组件实例。

谨慎使用 ref。如果你发现自己经常使用 ref 来在应用中“实现想要的功能”，你可以考虑去了解一下[自上而下的数据流](#)。

事件 {#events}

使用 React 元素处理事件时，有一些语法上差异：

- React 事件处理器使用 camelCase（驼峰式命名）而不使用小写命名。
- 通过 JSX，你可以直接传入一个函数，而不是传入一个字符串，来作为事件处理器。

协调 {#reconciliation}

当组件的 props 或 state 发生变化时，React 通过将最新返回的元素与原先渲染的元素进行比较，来决定是否有必要进行一次实际的 DOM 更新。当它们不相等时，React 才会更新 DOM。这个过程被称为“协调”。

JavaScript 环境要求

React 18 支持所有现代浏览器 (Edge, Firefox, Chrome, Safari 等)。

如果你支持旧的浏览器和设备, 如果 Internet Explorer, 它们没有提供原生的现代浏览器特性, 或有些不符合标准的实现, 你可以考虑在应用程序中引入一个全局的 polyfill。

以下是 React 18 使用的现代浏览器特性的列表 :

- `Promise`
- `Symbol`
- `Object.assign`

这些特性的 polyfill 取决于你的环境, 对于大部分用户来说, 你可以配置你的 [Browserlist](#)。对于其他用户, 你可能需要直接引入类似于 `core-js` 这样的 polyfills。

Profiler API

`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` 优化的部分，并从相关优化中获益。

注意：

Profiling 增加了额外的开支，所以它在[生产构建中会被禁用](#)。

为了将 profiling 功能加入生产环境中，React 提供了使 profiling 可用的特殊的生产构建环境。从 fb.me/react-profiling 了解更多关于如何使用这个构建环境的信息。

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)。

例如，为了分析 `Navigation` 组件和它的子代：

```
render(
  <App>
    <Profiler id="Navigation" onRender={callback}>
      <Navigation {...props} />
    </Profiler>
    <Main {...props} />
  </App>
);
```

多个 `Profiler` 组件能测量应用中的不同部分：

```
render(
  <App>
    <Profiler id="Navigation" onRender={callback}>
      <Navigation {...props} />
    </Profiler>
    <Profiler id="Main" onRender={callback}>
      <Main {...props} />
    </Profiler>
  </App>
);
```

嵌套使用 `Profiler` 组件来测量相同一个子树下的不同组件：

```
render(
  <App>
    <Profiler id="Panel" onRender={callback}>
      <Panel {...props}>
        <Profiler id="Content" onRender={callback}>
          <Content {...props} />
        </Profiler>
      </Panel>
    </Profiler>
  </App>
);
```

```

        <Profiler id="PreviewPane" onRender={callback}>
          <PreviewPane {...props} />
        </Profiler>
      </Panel>
    </Profiler>
  </App>
);

```

注意

尽管 Profiler 是一个轻量级组件，我们依然应该在需要时才去使用它。对一个应用来说，每添加一些都会给 CPU 和内存带来一些负担。

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。

```

function onRenderCallback(
  id, // 发生提交的 Profiler 树的 "id"
  phase, // "mount" (如果组件树刚加载) 或者 "update" (如果它重渲染了) 之一
  actualDuration, // 本次更新 committed 花费的渲染时间
  baseDuration, // 估计不使用 memoization 的情况下渲染整棵子树需要的时间
  startTime, // 本次更新中 React 开始渲染的时间
  commitTime, // 本次更新中 React committed 的时间
  interactions // 属于本次更新的 interactions 的集合
) {
  // 合计或记录渲染时间...
}

```

让我们来仔细研究一下各个 prop:

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`, `useMemo`, `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`”的集合会被追踪。(例如当 `render` 或者 `setState` 被调用时)。

注意

Interactions 能用来识别更新是由什么引起的，尽管这个追踪更新的 API 依然是实验性质的。

从 fb.me/react-interaction-tracing 了解更多

[Go to TOC](#)

PureRenderMixin

Note

The `PureRenderMixin` mixin predates `React.PureComponent`. This reference doc is provided for legacy purposes, and you should consider using `React.PureComponent` instead.

If your React component's render function renders the same result given the same props and state, you can use this mixin for a performance boost in some cases.

Example:

```
var PureRenderMixin = require('react-addons-pure-render-mixin');
var createReactClass = require('create-react-class');
createReactClass({
  mixins: [PureRenderMixin],

  render: function() {
    return <div className={this.props.className}>foo</div>;
  }
});
```

Example using ES6 class syntax:

```
import PureRenderMixin from 'react-addons-pure-render-mixin';
class FooComponent extends React.Component {
  constructor(props) {
    super(props);
    this.shouldComponentUpdate = PureRenderMixin.shouldComponentUpdate.bind(this);
  }

  render() {
    return <div className={this.props.className}>foo</div>;
  }
}
```

Under the hood, the mixin implements `shouldComponentUpdate`, in which it compares the current props and state with the next ones and returns `false` if the equalities pass.

Note:

This only shallowly compares the objects. If these contain complex data structures, it may produce false-negatives for deeper differences. Only mix into components which have simple props and state, or use `forceUpdate()` when you know deep data structures have changed. Or, consider using [immutable objects](#) to facilitate fast comparisons of nested data.

Furthermore, `shouldComponentUpdate` skips updates for the whole component subtree. Make sure all the children components are also "pure".

React.Component

本章节提供了 React class 组件的详细 API 参考。本章节默认你已熟悉基本的 React 概念，例如 [组件 & Props](#)，以及 [State & 生命周期](#) 等。如果你还未熟悉，请先阅读之前章节进行学习。

概览

React 的组件可以定义为 class 或函数的形式。class 组件目前提供了更多的功能，这些功能将在此章节中详细介绍。如需定义 class 组件，需要继承 `React.Component`：

```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

在 `React.Component` 的子类中有个必须定义的 `render()` 函数。本章节介绍其他方法均为可选。

我们强烈建议你不要创建自己的组件基类。在 React 组件中，代码重用的主要方式是组合而不是继承。

注意:

React 并不会强制你使用 ES6 的 class 语法。如果你倾向于不使用它，你可以使用 `create-react-class` 模块或类似的自定义抽象来代替。请查阅[不使用 ES6](#) 了解更多。

组件的生命周期

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。**你可以使用此**生命周期图谱**作为速查表。**在下述列表中，常用的生命周期方法会被加粗。其余生命周期函数的使用则相对罕见。

挂载

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()`
- `static getDerivedStateFromProps()`
- `render()`
- `componentDidMount()`

注意:

下述生命周期方法即将过时，在新代码中应该[避免使用它们](#)：

- `UNSAFE_componentWillMount()`

更新

当组件的 props 或 state 发生变化时会触发更新。组件更新的生命周期调用顺序如下：

- `static getDerivedStateFromProps()`
- `shouldComponentUpdate()`
- `render()`
- `getSnapshotBeforeUpdate()`
- `componentDidUpdate()`

注意：

下述方法即将过时，在新代码中应该[避免使用它们](#)：

- `UNSAFE_componentWillUpdate()`
- `UNSAFE_componentWillReceiveProps()`

卸载

当组件从 DOM 中移除时会调用如下方法：

- `componentWillUnmount()`

错误处理

当渲染过程、生命周期，或子组件的构造函数中抛出错误时，会调用如下方法：

- `static getDerivedStateFromError()`
- `componentDidCatch()`

其他 APIs

组件还提供了一些额外的 API：

- `setState()`
- `forceUpdate()`

class 属性

- `defaultProps`
- `displayName`

实例属性

- `props`
- `state`

参考

常用的生命周期方法

本节中的方法涵盖了创建 React 组件时能遇到的绝大多数用例。想要更好了解这些方法，可以参考[生命周期图谱](#)。

`render()` {#render}

`render()`

`render()` 方法是 class 组件中唯一必须实现的方法。

当 `render` 被调用时，它会检查 `this.props` 和 `this.state` 的变化并返回以下类型之一：

- **React 元素**。通常通过 JSX 创建。例如，`<div />` 会被 React 渲染为 DOM 节点，`<MyComponent />` 会被 React 渲染为自定义组件，无论是`<div />` 还是`<MyComponent />` 均为 React 元素。
- **数组或 fragments**。使得 `render` 方法可以返回多个元素。欲了解更多详细信息，请参阅 [fragments](#) 文档。
- **Portals**。可以渲染子节点到不同的 DOM 子树中。欲了解更多详细信息，请参阅有关 [portals](#) 的文档。
- **字符串或数值类型**。它们在 DOM 中会被渲染为文本节点。
- **布尔类型或 null**。什么都不渲染。(主要用于支持返回 `test && <Child />` 的模式，其中 `test` 为布尔类型。)

`render()` 函数应该为纯函数，这意味着在不修改组件 `state` 的情况下，每次调用时都返回相同的结果，并且它不会直接与浏览器交互。

如需与浏览器进行交互，请在 `componentDidMount()` 或其他生命周期方法中执行你的操作。保持 `render()` 为纯函数，可以使组件更容易思考。

注意

如果 `shouldComponentUpdate()` 返回 `false`，则不会调用 `render()`。

`constructor()` {#constructor}

`constructor(props)`

如果不初始化 state 或不进行方法绑定，则不需要为 React 组件实现构造函数。

在 React 组件挂载之前，会调用它的构造函数。在为 `React.Component` 子类实现构造函数时，应在其他语句之前调用 `super(props)`。否则，`this.props` 在构造函数中可能会出现未定义的 bug。

通常，在 React 中，构造函数仅用于以下两种情况：

- 通过给 `this.state` 赋值对象来初始化**内部 state**。
- 为**事件处理函数绑定实例**

在 `constructor()` 函数中不要调用 `setState()` 方法。如果你的组件需要使用内部 state，请直接在构造函数中为 `this.state` 赋值初始 state：

```
constructor(props) {
  super(props);
  // 不要在这里调用 this.setState()
  this.state = { counter: 0 };
  this.handleClick = this.handleClick.bind(this);
}
```

只能在构造函数中直接为 `this.state` 赋值。如需在其他方法中赋值，你应使用 `this.setState()` 替代。

要避免在构造函数中引入任何副作用或订阅。如遇到此场景，请将对应的操作放置在 `componentDidMount` 中。

注意

避免将 `props` 的值复制给 `state`！这是一个常见的错误：

```
constructor(props) {
  super(props);
  // 不要这样做
  this.state = { color: props.color };
}
```

如此做毫无必要（你可以直接使用 `this.props.color`），同时还产生了 bug（更新 prop 中的 `color` 时，并不会影响 `state`）。

只有在你刻意忽略 prop 更新的情况下使用。此时，应将 prop 重命名为 `initialColor` 或 `defaultColor`。必要时，你可以修改它的 `key`，以强制“重置”其内部 `state`。

请参阅关于[避免派生状态的博文](#)，以了解出现 `state` 依赖 `props` 的情况该如何处理。

componentDidMount() {#componentdidmount}

componentDidMount()

`componentDidMount()` 会在组件挂载后（插入 DOM 树中）立即调用。依赖于 DOM 节点的初始化应该放在这里。如需通过网络请求获取数据，此处是实例化请求的好地方。

这个方法是比较适合添加订阅的地方。如果添加了订阅，请不要忘记在 `componentWillUnmount()` 里取消订阅。

你可以在 `componentDidMount()` 里直接调用 `setState()`。它将触发额外渲染，但此渲染会发生在浏览器更新屏幕之前。如此保证了即使在 `render()` 两次调用的情况下，用户也不会看到中间状态。请谨慎使用该模式，因为它会导致性能问题。通常，你应该在 `constructor()` 中初始化 `state`。如果你的渲染依赖于 DOM 节点的大小或位置，比如实现 `modals` 和 `tooltips` 等情况下，你可以使用此方式处理。

componentDidUpdate() {#componentdidupdate}

componentDidUpdate(prevProps, prevState, snapshot)

`componentDidUpdate()` 会在更新后会被立即调用。首次渲染不会执行此方法。

当组件更新后，可以在此处对 DOM 进行操作。如果你对更新前后的 props 进行了比较，也可以选择在此处进行网络请求。（例如，当 props 未发生变化时，则不会执行网络请求）。

```
componentDidUpdate(prevProps) {
  // 典型用法（不要忘记比较 props）：
  if (this.props.userID !== prevProps.userID) {
    this.fetchData(this.props.userID);
  }
}
```

你也可以在 `componentDidUpdate()` 中直接调用 `setState()`，但请注意它必须被包裹在一个条件语句里，正如上述的例子那样进行处理，否则会导致死循环。它还会导致额外的重新渲染，虽然用户不可见，但会影响组件性能。不要将 props “镜像”给 state，请考虑直接使用 props。欲了解更多有关内容，请参阅[为什么 props 复制给 state 会产生 bug](#)。

如果组件实现了 `getSnapshotBeforeUpdate()` 生命周期（不常用），则它的返回值将作为 `componentDidUpdate()` 的第三个参数“snapshot”参数传递。否则此参数将为 `undefined`。

注意

如果 `shouldComponentUpdate()` 返回值为 `false`，则不会调用 `componentDidUpdate()`。

componentWillUnmount() {#componentwillunmount}

componentWillUnmount()

`componentWillUnmount()` 会在组件卸载及销毁之前直接调用。在此方法中执行必要的清理操作，例如，清除 timer，取消网络请求或清除在 `componentDidMount()` 中创建的订阅等。

`componentWillUnmount()` 中不应调用 `setState()`，因为该组件将永远不会再重新渲染。组件实例卸载后，将永远不会再挂载它。

不常用的生命周期方法

本节中的生命周期方法并不太常用。它们偶尔会很方便，但是大部分情况下组件可能都不需要它们。你可以在[生命周期图谱](#)中，选择“显示不常用的生命周期”复选框，即可看到下述相关方法。

shouldComponentUpdate() {#shouldcomponentupdate}

shouldComponentUpdate(nextProps, nextState)

根据 `shouldComponentUpdate()` 的返回值，判断 React 组件的输出是否受当前 state 或 props 更改的影响。默认行为是 state 每次发生变化组件都会重新渲染。大部分情况下，你应该遵循默认行为。

当 props 或 state 发生变化时，`shouldComponentUpdate()` 会在渲染执行之前被调用。返回值默认为 `true`。首次渲染或使用 `forceUpdate()` 时不会调用该方法。

此方法仅作为**性能优化的方式**而存在。不要企图依靠此方法来“阻止”渲染，因为这可能会产生 bug。你应该**考虑使用内置的 `PureComponent` 组件**，而不是手动编写 `shouldComponentUpdate()`。`PureComponent` 会对 props 和 state 进行浅层比较，并减少了跳过必要更新的可能性。

如果你一定要手动编写此函数，可以将 `this.props` 与 `nextProps` 以及 `this.state` 与 `nextState` 进行比较，并返回 `false` 以告知 React 可以跳过更新。请注意，返回 `false` 并不会阻止子组件在 state 更改时重新渲染。

我们不建议在 `shouldComponentUpdate()` 中进行深层比较或使用 `JSON.stringify()`。这样非常影响效率，且会损害性能。

目前，如果 `shouldComponentUpdate()` 返回 `false`，则不会调用 `UNSAFE_componentWillUpdate()`，`render()` 和 `componentDidUpdate()`。后续版本，React 可能会将 `shouldComponentUpdate` 视为提示而不是严格的指令，并且，当返回 `false` 时，仍可能导致组件重新渲染。

`static getDerivedStateFromProps() {#static-getderivedstatefromprops}`

`static getDerivedStateFromProps(props, state)`

`getDerivedStateFromProps` 会在调用 `render` 方法之前调用，并且在初始挂载及后续更新时都会被调用。它应返回一个对象来更新 state，如果返回 `null` 则不更新任何内容。

此方法适用于**罕见的用例**，即 state 的值在任何时候都取决于 props。例如，实现 `<Transition>` 组件可能很方便，该组件会比较当前组件与下一组件，以决定针对哪些组件进行转场动画。

派生状态会导致代码冗余，并使组件难以维护。**确保你已熟悉这些简单的替代方案**：

- 如果你需要**执行副作用**（例如，数据提取或动画）以响应 props 中的更改，请改用 `componentDidUpdate`。
- 如果只想在 **prop 改变时重新计算某些数据**，**请使用 memoization helper 代替**。
- 如果你想**在 prop 改变时“重置”某些 state**，请考虑使组件**完全受控**或使用 `key` **使组件完全不受控** 代替。

此方法无权访问组件实例。如果你需要，可以通过提取组件 props 的纯函数及 class 之外的状态，在 `getDerivedStateFromProps()` 和其他 class 方法之间重用代码。

请注意，不管原因是什么，都会在每次渲染前触发此方法。这与 `UNSAFE_componentWillReceiveProps` 形成对比，后者仅在父组件重新渲染时触发，而不是在内部调用 `setState` 时。

`getSnapshotBeforeUpdate() {#getsnapshotbeforeupdate}`

`getSnapshotBeforeUpdate(prevProps, prevState)`

`getSnapshotBeforeUpdate()` 在最近一次渲染输出（提交到 DOM 节点）之前调用。它使得组件能在发生更改之前从 DOM 中捕获一些信息（例如，滚动位置）。此生命周期方法的任何返回值将作为参数传递给 `componentDidUpdate()`。

此用法并不常见，但它可能出现在 UI 处理中，如需要以特殊方式处理滚动位置的聊天线程等。

应返回 `snapshot` 的值（或 `null`）。

例如：

```
embed:react-component-reference/get-snapshot-before-update.js
```

在上述示例中，重点是从 `getSnapshotBeforeUpdate` 读取 `scrollHeight` 属性，因为“render”阶段生命周期（如 `render`）和“commit”阶段生命周期（如 `getSnapshotBeforeUpdate` 和 `componentDidUpdate`）之间可能存在延迟。

Error boundaries

[Error boundaries](#) 是 React 组件，它会在其子组件树中的任何位置捕获 JavaScript 错误，并记录这些错误，展示降级 UI 而不是崩溃的组件树。Error boundaries 组件会捕获在渲染期间，在生命周期方法以及其整个树的构造函数中发生的错误。

如果 class 组件定义了生命周期方法 `static getDerivedStateFromError()` 或 `componentDidCatch()` 中的任何一个（或两者），它就成为了 Error boundaries。通过生命周期更新 state 可让组件捕获树中未处理的 JavaScript 错误并展示降级 UI。

仅使用 Error boundaries 组件来从意外异常中恢复的情况；**不要将它们用于流程控制**。

欲了解更多详细信息，请参阅 [React 16 中的错误处理](#)。

注意

Error boundaries 仅捕获组件树中**以下**组件中的错误。但它本身的错误无法捕获。

```
static getDerivedStateFromError() {#static-
getderivedstatefromerror}
```

```
static getDerivedStateFromError(error)
```

此生命周期会在后代组件抛出错误后被调用。它将抛出的错误作为参数，并返回一个值以更新 state

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    // 更新 state 使下一次渲染可以显降级 UI
    return { hasError: true };
  }
}
```

```

render() {
  if (this.state.hasError) {
    // 你可以渲染任何自定义的降级 UI
    return <h1>Something went wrong.</h1>;
  }

  return this.props.children;
}
}

```

注意

`getDerivedStateFromError()` 会在渲染阶段调用，因此不允许出现副作用。如遇此类情况，请改用 `componentDidCatch()`。

componentDidCatch() {#componentdidcatch}

`componentDidCatch(error, info)`

此生命周期在后代组件抛出错误后被调用。它接收两个参数：

1. `error` —— 抛出的错误。
2. `info` —— 带有 `componentStack` key 的对象，其中包含有关组件引发错误的栈信息。

`componentDidCatch()` 会在“提交”阶段被调用，因此允许执行副作用。它应该用于记录错误之类的情况：

```

class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    // 更新 state 使下一次渲染可以显示降级 UI
    return { hasError: true };
  }

  componentDidCatch(error, info) {
    // "组件堆栈" 例子：
    //   in ComponentThatThrows (created by App)
    //   in ErrorBoundary (created by App)
    //   in div (created by App)
    //   in App
    logComponentStackToMyService(info.componentStack);
  }

  render() {
    if (this.state.hasError) {
      // 你可以渲染任何自定义的降级 UI
      return <h1>Something went wrong.</h1>;
    }
  }
}

```

```

    return this.props.children;
}
}

```

React 的开发和生产构建版本在 `componentDidCatch()` 的方式上有轻微差别。

在 开 发 模 式 下 , 错 误 会 冒 泡 至 `window` , 这 意 味 着 任 何 `window.onerror` 或 `window.addEventListener('error', callback)` 会 中 断 这 些 已 经 被 `componentDidCatch()` 捕 获 的 错 误。

相 反 , 在 生 产 模 式 下 , 错 误 不 会 冒 泡 , 这 意 味 着 任 何 根 错 误 处 理 器 只 会 接 受 那 些 没 有 显 式 地 被 `componentDidCatch()` 捕 获 的 错 误。

注意

如 果 发 生 错 误 , 你 可 以 通 过 调 用 `setState` 使 用 `componentDidCatch()` 渲 染 降 级 UI , 但 在 未 来 的 版 本 中 将 不 推 荐 这 样 做。可 以 使用 静 态 `getDerivedStateFromError()` 来 处 理 降 级 渲 染。

过时的生命周期方法

以下生命周期方法标记为“过时”。这些方法仍然有效，但不建议在新代码中使用它们。参阅此[博客文章](#)以了解更多有关迁移旧版生命周期方法的信息。

UNSAFE_componentWillMount() {#unsafe_componentwillmount}

`UNSAFE_componentWillMount()`

注意

此 生命周期 之 前 名 为 `componentWillMount` 。该 名 称 将 继 续 使 用 至 React 17。可 以 使用 `rename-unsafe-lifecycles` [codemod](#) 自 动 更新 你 的 组件。

`UNSAFE_componentWillMount()` 在 挂 载 之 前 被 调 用。它 在 `render()` 之 前 调 用, 因 此 在 此 方 法 中 同 步 调 用 `setState()` 不 会 触 发 额 外 渲 染。通 常, 我 们 建 议 使 用 `constructor()` 来 初始化 `state`。

避 免 在 此 方 法 中 引 入 任 何 副 作 用 或 订 阅。如 遇 此 种 情 况, 请 改 用 `componentDidMount()` 。

此 方法 是 服 务 端 渲 染 唯 一 会 调 用 的 生 命 周 期 函 数。

UNSAFE_componentWillReceiveProps() {#unsafe_componentwillreceiveprops}

`UNSAFE_componentWillReceiveProps(nextProps)`

注意

此生命周期之前名为 `componentWillReceiveProps`。该名称将继续使用至 React 17。可以使用 `rename-unsafe-lifecycles codemod` 自动更新你的组件。

注意：

使用此生命周期方法通常会出现 bug 和不一致性：

- 如果你需要执行副作用（例如，数据提取或动画）以响应 props 中的更改，请改用 `componentDidUpdate` 生命周期。
- 如果你使用 `componentWillReceiveProps` 仅在 prop 更改时重新计算某些数据，请[使用 memoization helper 代替](#)。
- 如果你使用 `componentWillReceiveProps` 是为了在 prop 更改时“重置”某些 state，请考虑使组件[完全受控](#)或[使用 key 使组件完全不受控](#)代替。

对于其他使用场景，请[遵循此博客文章中有关派生状态的建议](#)。

`UNSAFE_componentWillReceiveProps()` 会在已挂载的组件接收新的 props 之前被调用。如果你需要更新状态以响应 prop 更改（例如，重置它），你可以比较 `this.props` 和 `nextProps` 并在此方法中使用 `this.setState()` 执行 state 转换。

请注意，如果父组件导致组件重新渲染，即使 props 没有更改，也会调用此方法。如果只想处理更改，请确保进行当前值与变更值的比较。

在[挂载](#)过程中，React 不会针对初始 props 调用 `UNSAFE_componentWillReceiveProps()`。组件只会在组件的 props 更新时调用此方法。调用 `this.setState()` 通常不会触发 `UNSAFE_componentWillReceiveProps()`。

`UNSAFE_componentWillUpdate()` {#unsafe_componentwillupdate}**`UNSAFE_componentWillUpdate(nextProps, nextState)`****注意**

此生命周期之前名为 `componentWillUpdate`。该名称将继续使用至 React 17。可以使用 `rename-unsafe-lifecycles codemod` 自动更新你的组件。

当组件收到新的 props 或 state 时，会在渲染之前调用 `UNSAFE_componentWillUpdate()`。使用此作为在更新发生之前执行准备更新的机会。初始渲染不会调用此方法。

注意，你不能在此方法中调用 `this.setState()`；在 `UNSAFE_componentWillUpdate()` 返回之前，你也不应该执行任何其他操作（例如，`dispatch Redux 的 action`）触发对 React 组件的更新。

通常，此方法可以替换为 `componentDidUpdate()`。如果你在此方法中读取 DOM 信息（例如，为了保存滚动位置），则可以将此逻辑移至 `getSnapshotBeforeUpdate()` 中。

注意

如果 `shouldComponentUpdate()` 返回 `false`，则不会调用 `UNSAFE_componentWillUpdate()`。

其他 API

不同于上述生命周期方法（React 主动调用），以下方法是你可以调用的方法。

只有两个方法：`setState()` 和 `forceUpdate()`。

`setState()` {#setstate}

`setState(updater[, callback])`

`setState()` 将对组件 state 的更改排入队列，并通知 React 需要使用更新后的 state 重新渲染此组件及其子组件。这是用于更新用户界面以响应事件处理器和处理服务器数据的主要方式。

将 `setState()` 视为请求而不是立即更新组件的命令。为了更好的感知性能，React 会延迟调用它，然后通过一次传递更新多个组件。在罕见的情况下，你需要强制 DOM 更新同步应用，你可以使用 `flushSync` 来包装它，但这可能会损害性能。

`setState()` 并不总是立即更新组件。它会批量推迟更新。这使得在调用 `setState()` 后立即读取 `this.state` 成为了隐患。为了消除隐患，请使用 `componentDidUpdate` 或者 `setState` 的回调函数 (`setState(updater, callback)`)，这两种方式都可以保证在应用更新后触发。如需基于之前的 state 来设置当前的 state，请阅读下述关于参数 `updater` 的内容。

除非 `shouldComponentUpdate()` 返回 `false`，否则 `setState()` 将始终执行重新渲染操作。如果可变对象被使用，且无法在 `shouldComponentUpdate()` 中实现条件渲染，那么仅在新旧状态不一时调用 `setState()` 可以避免不必要的重新渲染。

参数一为带有形式参数的 `updater` 函数：

`(state, props) => stateChange`

`state` 是对应用变化时组件状态的引用。当然，它不应直接被修改。你应该使用基于 `state` 和 `props` 构建的新对象来表示变化。例如，假设我们想根据 `props.step` 来增加 `state`：

```
this.setState((state, props) => {
  return {counter: state.counter + props.step};
});
```

`updater` 函数中接收的 `state` 和 `props` 都保证为最新。`updater` 的返回值会与 `state` 进行浅合并。

`setState()` 的第二个参数为可选的回调函数，它将在 `setState` 完成合并并重新渲染组件后执行。通常，我们建议使用 `componentDidUpdate()` 来代替此方式。

`setState()` 的第一个参数除了接受函数外，还可以接受对象类型：

```
  setState(stateChange[, callback])
```

`stateChange` 会将传入的对象浅层合并到新的 `state` 中，例如，调整购物车商品数：

```
  this.setState({quantity: 2})
```

这种形式的 `setState()` 也是异步的，并且在同一周期内会对多个 `setState` 进行批处理。例如，如果在同一周期内多次设置商品数量增加，则相当于：

```
  Object.assign(
    previousState,
    {quantity: state.quantity + 1},
    {quantity: state.quantity + 1},
    ...
)
```

后调用的 `setState()` 将覆盖同一周期内先调用 `setState` 的值，因此商品数仅增加一次。如果后续状态取决于当前状态，我们建议使用 `updater` 函数的形式代替：

```
  this.setState((state) => {
  return {quantity: state.quantity + 1};
});
```

有关更多详细信息，请参阅：

- [State 和生命周期指南](#)
- [深入学习：何时以及为什么 `setState\(\)` 会批量执行？](#)
- [深入：为什么不直接更新 `this.state`？](#)

forceUpdate() {#forceupdate}

```
  component.forceUpdate(callback)
```

默认情况下，当组件的 `state` 或 `props` 发生变化时，组件将重新渲染。如果 `render()` 方法依赖于其他数据，则可以调用 `forceUpdate()` 强制让组件重新渲染。

调用 `forceUpdate()` 将致使组件调用 `render()` 方法，此操作会跳过该组件的 `shouldComponentUpdate()`。但其子组件会触发正常的生命周期方法，包括 `shouldComponentUpdate()` 方法。如果标记发生变化，React 仍将只更新 DOM。

通常你应该避免使用 `forceUpdate()`，尽量在 `render()` 中使用 `this.props` 和 `this.state`。

Class 属性

defaultProps {#defaultprops}

`defaultProps` 可以为 Class 组件添加默认 props。这一般用于 props 未赋值，但又不能为 `null` 的情况。例如：

```
class CustomButton extends React.Component {
  // ...
}

CustomButton.defaultProps = {
  color: 'blue'
};
```

如果未提供 `props.color`，则默认设置为 `'blue'`

```
render() {
  return <CustomButton />; // props.color 将设置为 'blue'
}
```

如果 `props.color` 被设置为 `null`，则它将保持为 `null`

```
render() {
  return <CustomButton color={null} />; // props.color 将保持是 null
}
```

displayName {#displayname}

`displayName` 字符串多用于调试消息。通常，你不需要设置它，因为它可以根据函数组件或 class 组件的名称推断出来。如果调试时需要显示不同的名称或创建高阶组件，请参阅[使用 displayName 轻松进行调试](#)了解更多。

实例属性

props {#props}

`this.props` 包括被该组件调用者定义的 props。欲了解 props 的详细介绍，请参阅[组件 & Props](#)。

需特别注意，`this.props.children` 是一个特殊的 prop，通常由 JSX 表达式中的子组件组成，而非组件本身定义。

state {#state}

组件中的 state 包含了随时可能发生变化的数据。state 由用户自定义，它是一个普通 JavaScript 对象。

如果某些值未用于渲染或数据流（例如，计时器 ID），则不必将其设置为 state。此类值可以在组件实例上定义。

欲了解关于 state 的更多信息，请参阅[State & 生命周期](#)。

永远不要直接改变 `this.state`，因为后续调用的 `setState()` 可能会替换掉你的改变。请把 `this.state` 看作是不可变的。

react

[Go to TOC](#)

ReactDOMClient

The `react-dom/client` package provides client-specific methods used for initializing an app on the client. Most of your components should not need to use this module.

```
import * as ReactDOM from 'react-dom/client';
```

If you use ES5 with npm, you can write:

```
var ReactDOM = require('react-dom/client');
```

Overview

The following methods can be used in client environments:

- `createRoot()`
- `hydrateRoot()`

Browser Support

React supports all modern browsers, although [some polyfills are required](#) for older versions.

Note

We do not support older browsers that don't support ES5 methods or microtasks such as Internet Explorer. You may find that your apps do work in older browsers if polyfills such as [es5-shim](#) and [es5-sham](#) are included in the page, but you're on your own if you choose to take this path.

Reference

`createRoot()` [\[#createroot\]](#)

```
createRoot(container[, options]);
```

Create a React root for the supplied `container` and return the root. The root can be used to render a React element into the DOM with `render`:

```
const root = createRoot(container);
root.render(element);
```

`createRoot` accepts two options:

- `onRecoverableError` : optional callback called when React automatically recovers from errors.
- `identifierPrefix` : optional prefix React uses for ids generated by `React.useId`. Useful to avoid conflicts when using multiple roots on the same page. Must be the same prefix used on the server.

The root can also be unmounted with `unmount`:

```
root.unmount();
```

Note:

`createRoot()` controls the contents of the container node you pass in. Any existing DOM elements inside are replaced when `render` is called. Later calls use React's DOM diffing algorithm for efficient updates.

`createRoot()` does not modify the container node (only modifies the children of the container). It may be possible to insert a component to an existing DOM node without overwriting the existing children.

Using `createRoot()` to hydrate a server-rendered container is not supported. Use `hydrateRoot()` instead.

hydrateRoot() {#hydrateroot}

```
hydrateRoot(container, element[, options])
```

Same as `createRoot()`, but is used to hydrate a container whose HTML contents were rendered by `ReactDOMServer`. React will attempt to attach event listeners to the existing markup.

`hydrateRoot` accepts two options:

- `onRecoverableError`: optional callback called when React automatically recovers from errors.
- `identifierPrefix`: optional prefix React uses for ids generated by `React.useId`. Useful to avoid conflicts when using multiple roots on the same page. Must be the same prefix used on the server.

Note

React expects that the rendered content is identical between the server and the client. It can patch up differences in text content, but you should treat mismatches as bugs and fix them. In development mode, React warns about mismatches during hydration. There are no guarantees that attribute differences will be patched up in case of mismatches. This is important for performance reasons because in most apps, mismatches are rare, and so validating all markup would be prohibitively expensive.

[Go to TOC](#)

ReactDOMServer

`ReactDOMServer` 对象允许你将组件渲染成静态标记。通常，它被使用在 Node 服务端上：

```
// ES modules
import * as ReactDOMServer from 'react-dom/server';
// CommonJS
var ReactDOMServer = require('react-dom/server');
```

概览

这些方法只在支持 **Node.js Streams** 的环境下可用：

- `renderToPipeableStream()`
- `renderToNodeStream()` (Deprecated)
- `renderToStaticNodeStream()`

这些方法只在支持 **Web Streams** 的环境中可用（这包含浏览器，Deno 和一些现代边缘 runtime）：

- `renderToReadableStream()`

以下方法可以在不支持 Stream 的环境中使用：

- `renderToString()`
- `renderToStaticMarkup()`

参考

`renderToPipeableStream()` {#rendertopipeablestream}

`ReactDOMServer.renderToPipeableStream(element, options)`

将一个 React 元素渲染为初始 HTML。返回一个带有 `pipe(res)` 方法的流，用于管道输出。`abort()` 用于中止请求。完美支持了 `suspense` 和 HTML 流，“延迟”的内容块会通过内联的 `<script>` 标签嵌入。[了解更多](#)

如果你在一个已经被服务端渲染标记的节点上调用 `ReactDOM.hydrateRoot()`，React 会保留它，只附加事件处理程序，让你有一个非常高性能的首次加载体验。

```
let didError = false;
const stream = renderToPipeableStream(
  <App />,
  {
    onShellReady() {
      // The content above all Suspense boundaries is ready.
      // If something errored before we started streaming, we set the error code
      // appropriately.
      res.statusCode = didError ? 500 : 200;
      res.setHeader('Content-type', 'text/html');
      stream.pipe(res);
    },
  }
);
```

```

onShellError(error) {
  // Something errored before we could complete the shell so we emit an
  // alternative shell.
  res.statusCode = 500;
  res.send(
    '<!doctype html><p>Loading...</p><script src="clientrender.js"></script>'
  );
},
onAllReady() {
  // If you don't want streaming, use this instead of onShellReady.
  // This will fire after the entire page content is ready.
  // You can use this for crawlers or static generation.

  // res.statusCode = didError ? 500 : 200;
  // res.setHeader('Content-type', 'text/html');
  // stream.pipe(res);
},
onError(err) {
  didError = true;
  console.error(err);
},
}
);

```

具体参见 [完整的选项列表](#).

注意：

这是一个针对 Node.js 的 API。只在支持 [Web Streams](#) 的环境下可用，如 Deno 以及现代边缘 runtimes，应使用 `renderToReadableStream` 代替。

`renderToReadableStream()` [#rendertoreadablestream](#)

`ReactDOMServer.renderToReadableStream(element, options);`

将一个 React 元素通过流的形式注入初始的 HTML 中。返回值为 Promise, resolve 一个 [可读 Stream](#)。完美支持了 `suspense` 和 HTML 流。[了解更多](#)

如果你在一个已经被服务端渲染标记的节点上调用 `ReactDOM.hydrateRoot()`，React 会保留它，只为其附加事件处理程序。让你拥有一个非常良好的首次加载体验。

```

let controller = new AbortController();
let didError = false;
try {
  let stream = await renderToReadableStream(
    <html>
      <body>Success</body>
    </html>,
    {
      signal: controller.signal,
      onError(error) {
        didError = true;
        console.error(error);
      }
    }
  );
}

```

```

    });

    // This is to wait for all Suspense boundaries to be ready. You can uncomment
    // this line if you want to buffer the entire HTML instead of streaming it.
    // You can use this for crawlers or static generation:

    // await stream.allReady;

    return new Response(stream, {
      status: didError ? 500 : 200,
      headers: {'Content-Type': 'text/html'},
    });
  } catch (error) {
    return new Response(
      `<!doctype html><p>Loading...</p><script src="clientrender.js"></script>`,
    {
      status: 500,
      headers: {'Content-Type': 'text/html'},
    }
  );
}

```

具体参见 [完整的选项列表](#)。

注意：

该 API 依赖 [Web Streams](#)。对于，请使用 `renderToPipeableStream` 代替。

`renderToString()` (Deprecated)

`ReactDOMServer.renderToString(element)`

将一个 React 元素渲染成其初始 HTML。返回一个可输出 HTML 字符串的 [Node.js 可读流](#)。通过可读流输出的 HTML 完全等同于 `ReactDOMServer.renderToString` 返回的 HTML。你可以使用本方法在服务器上生成 HTML，并在初始请求时将标记下发，以加快页面加载速度，并允许搜索引擎抓取你的页面以达到 SEO 优化的目的。

如果你在已有服务端渲染标记的节点上调用 `ReactDOM.hydrateRoot()` 方法，React 将会保留该节点且只进行事件处理绑定，从而让你有一个非常高性能的首次加载体验。

注意：

这个 API 仅允许在服务端使用。不允许在浏览器使用。

通过本方法返回的流会返回一个由 utf-8 编码的字节流。如果你需要另一种编码的流，请查看像 [iconv-lite](#) 这样的项目，它为转换文本提供了转换流。

renderToStaticNodeStream() {#rendertostaticnodestream}

ReactDOMServer.renderToStaticNodeStream(element)

此方法与 `renderToNodeStream` 相似，但此方法不会在 React 内部创建的额外 DOM 属性，例如 `data-reactroot`。如果你希望把 React 当作静态页面生成器来使用，此方法会非常有用，因为去除额外的属性可以节省一些字节。

通过可读流输出的 HTML，完全等同于 `ReactDOMServer.renderToStaticMarkup` 返回的 HTML。

如果你计划在前端使用 React 以使得标记可交互，请不要使用此方法。你可以在服务端上使用 `renderToNodeStream` 或在前端上使用 `ReactDOM.hydrateRoot()` 来代替此方法。

注意：

此 API 仅限于服务端使用，在浏览器中是不可用的。

通过本方法返回的流会返回一个由 utf-8 编码的字节流。如果你需要另一种编码的流，请查看像 `iconv-lite` 这样的项目，它为转换文本提供了转换流。

renderToString() {#rendertostring}

ReactDOMServer.renderToString(element)

将一个 React 元素渲染成其初始的 HTML。React 将返回一个 HTML 字符串。你可以使用这种方法在服务器上生产 HTML，并在初始请求中发送标记。以加快页面加载速度，并允许搜索引擎以 SEO 为目的抓取你的页面。

如果你在一个已被服务端渲染标记的节点上调用 `ReactDOM.hydrateRoot()`，React 会保留它，只附加事件处理程序，让你有一个非常良好的首次加载体验。

注意：

此 API 对 Suspense 支持有限，并且不支持流。

在服务端，建议使用 `renderToPipeableStream` (Node.js) 或者 `renderToReadableStream` (for Web Streams) 代替。

renderToStaticMarkup() {#rendertostaticmarkup}

ReactDOMServer.renderToStaticMarkup(element)

与 `renderToString` 相似，只是该方法不会创建 React 内部使用的额外 DOM 属性，如 `data-reactroot`。如果你只想把 React 作为简单的静态页面生成器使用，此方法会非常实用。因为剥离多余的属性可以节省一些字节占用。

如果你打算在客户端使用 React 进行交互标记，那请不要使用这个方法。作为替代，你可以在服务器使用 `renderToString`，同时在客户端使用 `ReactDOM.hydrateRoot()`。

ReactDOM

`react-dom` 包提供了用户 DOM 的特定方法，可以在你应用程序的顶层进行使用，如果你有需要的话，还可以作为应急方案，在 React 模型以外的地方使用。

```
import * as ReactDOM from 'react-dom';
```

如果你使用 npm 和 ES5，你可以用：

```
var ReactDOM = require('react-dom');
```

`react-dom` 包还提供了客户端和服务器应用程序的特定模块：

- `react-dom/client`
- `react-dom/server`

概览

`react-dom` 包导出了如下这些方法：

- `createPortal()`
- `flushSync()`

如下这些方法也会被 `react-dom` 导出，但会被认为是遗弃：

- `render()`
- `hydrate()`
- `findDOMNode()`
- `unmountComponentAtNode()`

注意：

在 React 18，`render` 和 `hydrate` 都被新的 [客户端方法](#) 所取代。这些方法将警告你的应用程序将表现得像运行 React 17 一样（欲了解更多请参阅 [此处](#)）。

浏览器支持

React 支持所有的现代浏览器，尽管对于旧版本来说，可能需要引入 [相关的 polyfills 依赖](#)。

注意：

我们不支持那些不兼容 ES5 方法的旧版浏览器，但如果的应用包含了 polyfill，例如 `es5-shim` 和 `es5-sham` 你可能会发现你的应用仍然可以在这些浏览器中正常运行。但是如果你选择这种方法，你便需要孤军奋战了。

参考

`createPortal()` [\[#createportal\]](#)

`createPortal(child, container)`

创建 portal。Portal 提供了一种将子节点渲染到已 DOM 节点中的方式，该节点存在于 DOM 组件的层次结构之外。

`flushSync()` [\[#flushsync\]](#)

`flushSync(callback)`

强制 React 同步刷新提供的回调函数中的任何更新。这确保了 DOM 会被立即 更新。

```
// Force this state update to be synchronous.
flushSync(() => {
  setCount(count + 1);
});
// By this point, DOM is updated.
```

注意：

`flushSync` 会对性能产生很大影响。尽量少用。

`flushSync` 可能会迫使悬而未决的 Suspense 边界显示其 `fallback` 的状态。

`flushSync` 也可以运行待定副作用，并在返回之前同步应用它们所包含的任何更新。

当需要刷新内部的更新时，`flushSync` 也可以在回调外部刷新更新。例如，如果有来自点击的未决更新。React 可能会在刷新回调之前刷新这些更新。

遗留方法参考

`render()` [\[#render\]](#)

`render(element, container[, callback])`

注意：

在 React 18 中，`render` 函数已被 `createRoot` 函数所取代。具体请参阅 [createRoot](#) 以了解更多。

在提供的 `container` 里渲染一个 React 元素，并返回对该组件的引用（或者针对无状态组件返回 `null`）。

如果 React 元素之前已经在 `container` 里渲染过，这将会对其执行更新操作，并仅会在必要时改变 DOM 以映射最新的 React 元素。

如果提供了可选的回调函数，该回调将在组件被渲染或更新之后被执行。

注意：

`render()` 会控制你传入容器节点里的内容。当首次调用时，容器节点里的所有 DOM 元素都会被替换，后续的调用则会使用 React 的 DOM 差分算法（DOM differencing algorithm）进行高效的更新。

`render()` 不会修改容器节点（只会修改容器的子节点）。可以在不覆盖现有子节点的情况下，将组件插入已有的 DOM 节点中。

`render()` 目前会返回对根组件 `ReactComponent` 实例的引用。但是，目前应该避免使用返回的引用，因为它是历史遗留下来的内容，而且在未来版本的 React 中，组件渲染在某些情况下可能会是异步的。如果你真的需要获得对根组件 `ReactComponent` 实例的引用，那么推荐为根元素添加 `callback ref`。

使用 `render()` 对服务端渲染容器进行 `hydrate` 操作的方式已经被废弃，并且会在 React 17 被移除。作为替代，请使用 `hydrateRoot()`。

hydrate() {#hydrate}

`hydrate(element, container[, callback])`

注意：

在 React 18 中，请使用 `hydrateRoot` 来替代 `hydrate`。请参阅 `hydrateRoot` 以了解更多。

与 `render()` 相同，但它用于在 `ReactDOMServer` 渲染的容器中对 HTML 的内容进行 `hydrate` 操作。React 会尝试在已有标记上绑定事件监听器。

React 希望服务端与客户端渲染的内容完全一致。React 可以弥补文本内容的差异，但是你需要将不匹配的地方作为 bug 进行修复。在开发者模式下，React 会对 hydration 操作过程中的不匹配进行警告。但并不能保证在不匹配的情况下，修补属性的差异。由于性能的关系，这一点非常重要，因为大多是应用中不匹配的情况很少见，并且验证所有标记的成本非常昂贵。

如果单个元素的属性或者文本内容，在服务端和客户端之间有无法避免差异（比如：时间戳），则可以为元素添加 `suppressHydrationWarning={true}` 来消除警告。这种方式只在一级深度上有效，应只作为一种应急方案（escape hatch）。请不要过度使用！除非它是文本内容，否则 React 仍不会尝试修补差异，因此在未来的更新之前，仍会保持不一致。

如果你执意要在服务端与客户端渲染不同内容，你可以采用双重（two-pass）渲染。在客户端渲染不同内容的组件可以读取类似于 `this.state.isClient` 的 state 变量，你可以在 `componentDidMount()` 里将它设置为 `true`。这种方式在初始渲染过程中会与服务端渲染相同的内容，从而避免不匹配的情况出现，但在 hydration 操作之后，会同步进行额外的渲染操作。注意，因为进行了两次渲染，这种方式会使得组件渲染变慢，请小心使用。

记得保证弱网环境下的用户体验。JavaScript 代码的加载要比最初的 HTML 渲染晚的多。因此如果你只在客户端渲染不同的内容，其转换可能会不稳定。但是，如果执行顺利，那么在服务端负责渲染的 shell 会对渲染提供帮助，并且只显示客户端上额外的小组件。欲了解如何在不出现标记不匹配的情况下执行此操作，请参考上一段的解释。

unmountComponentAtNode() {#unmountcomponentatnode}

`unmountComponentAtNode(container)`

注意：

在 React 18 中，`unmountComponentAtNode` 已被 `root.unmount()` 取代。具体请参阅 [createRoot](#) 以了解更多。

从 DOM 中卸载组件，会将其事件处理器（event handlers）和 state 一并清除。如果指定容器上没有对应已挂载的组件，这个函数什么也不会做。如果组件被移除将会返回 `true`，如果没有组件可被移除将会返回 `false`。

findDOMNode() {#finddomnode}

注意：

`findDOMNode` 是一个访问底层 DOM 节点的应急方案（escape hatch）。在大多数情况下，不推荐使用该方法，因为它会破坏组件的抽象结构。[严格模式下该方法已弃用](#)。

`findDOMNode(component)`

如果组件已经被挂载到 DOM 上，此方法会返回浏览器中相应的原生 DOM 元素。此方法对于从 DOM 中读取值很有用，例如获取表单字段的值或者执行 DOM 检测（performing DOM measurements）。**大多数情况下，你可以绑定一个 ref 到 DOM 节点上，可以完全避免使用 `findDOMNode`。**

当组件渲染的内容为 `null` 或 `false` 时，`findDOMNode` 也会返回 `null`。当组件渲染的是字符串时，`findDOMNode` 返回的是字符串对应的 DOM 节点。从 React 16 开始，组件可能会返回有多个子节点的 fragment，在这种情况下，`findDOMNode` 会返回第一个非空子节点对应的 DOM 节点。

注意：

`findDOMNode` 只在已挂载的组件上可用（即，已经放置在 DOM 中的组件）。如果你尝试调用未挂载的组件（例如在一个还未创建的组件上调用 `render()` 中的 `findDOMNode()`）将会引发异常。

`findDOMNode` 不能用于函数组件。

React 顶层 API

`React` 是 React 库的入口。如果你通过使用 `<script>` 标签的方式来加载 React，则可以通过 `React` 全局变量对象来获得 React 的顶层 API。当你使用 ES6 与 npm 时，可以通过编写 `import React from 'react'` 来引入它们。当你使用 ES5 与 npm 时，则可以通过编写 `var React = require('react')` 来引入它们。

概览

组件

使用 React 组件可以将 UI 拆分为独立且复用的代码片段，每部分都可独立维护。你可以通过子类 `React.Component` 或 `React.PureComponent` 来定义 React 组件。

- `React.Component`
- `React.PureComponent`

如果你不使用 ES6 的 class，则可以使用 `create-react-class` 模块来替代。请参阅[不使用 ES6](#) 以获取更多详细信息。

React 组件也可以被定义为可被包装的函数：

- `React.memo`

创建 React 元素

我们建议[使用 JSX](#) 来编写你的 UI 组件。每个 JSX 元素都是调用 `React.createElement()` 的语法糖。一般来说，如果你使用了 JSX，就不再需要调用以下方法。

- `createElement()`
- `createFactory()`

请参阅[不使用 JSX](#) 以获取更多详细信息。

转换元素

`React` 提供了几个用于操作元素的 API：

- `cloneElement()`
- `isValidElement()`
- `React.Children`

Fragments

`React` 还提供了用于减少不必要的嵌套的组件。

- `React.Fragment`

Refs

- `React.createRef`
- `React.forwardRef`

Suspense

`Suspense` 使得组件可以“等待”某些操作结束后，再进行渲染。目前，`Suspense` 仅支持的使用场景是：[通过 `React.lazy` 动态加载组件](#)。它将在未来支持其它使用场景，如数据获取等。

- `React.lazy`
- `React.Suspense`

Transitions

`Transitions` 是 React 18 引入的一个全新的并发特性。它允许你将标记更新作为一个 `transitions`，这会告诉 React 它们可以被中断执行，并避免回到已经可见内容的 `Suspense` 降级方案。

- `React.startTransition`
- `React.useTransition`

Hooks

`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 React 特性。`Hook` 拥有[专属文档章节](#)和单独的 API 参考文档：

- [基础 Hook](#)
 - `useState`
 - `useEffect`
 - `useContext`
 - [额外的 Hook](#)
 - `useReducer`
 - `useCallback`
 - `useMemo`
 - `useRef`
 - `useImperativeHandle`
 - `useLayoutEffect`
 - `useDebugValue`
 - `useDeferredValue`
 - `useTransition`
 - `useId`
 - [Library Hooks](#)
 - `useSyncExternalStore`
 - `useInsertionEffect`
-

参考

React.Component {#reactcomponent}

`React.Component` 是使用 `ES6 classes` 方式定义 React 组件的基类：

```
class Greeting extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

请参阅 [React.Component API 参考](#)，获取与基类 `React.Component` 相关方法和属性的详细列表。

React.PureComponent {#reactpurecomponent}

`React.PureComponent` 与 `React.Component` 很相似。两者的区别在于 `React.Component` 并未实现 `shouldComponentUpdate()`，而 `React.PureComponent` 中以浅层对比 prop 和 state 的方式来实现了该函数。

如果赋予 React 组件相同的 props 和 state，`render()` 函数会渲染相同的内容，那么在某些情况下使用 `React.PureComponent` 可提高性能。

注意

`React.PureComponent` 中的 `shouldComponentUpdate()` 仅作对象的浅层比较。如果对象中包含复杂的数据结构，则有可能因为无法检查深层的差别，产生错误的比对结果。仅在你的 props 和 state 较为简单时，才使用 `React.PureComponent`，或者在深层数据结构发生变化时调用 `forceUpdate()` 来确保组件被正确地更新。你也可以考虑使用 [immutable 对象](#) 加速嵌套数据的比较。

此外，`React.PureComponent` 中的 `shouldComponentUpdate()` 将跳过所有子组件树的 prop 更新。因此，请确保所有子组件也都是“纯”的组件。

React.memo {#reactmemo}

```
const MyComponent = React.memo(function MyComponent(props) {
  /* 使用 props 渲染 */
});
```

`React.memo` 为高阶组件。

如果你的组件在相同 props 的情况下渲染相同的结果，那么你可以通过将其包装在 `React.memo` 中调用，以此通过记忆组件渲染结果的方式来提高组件的性能表现。这意味着在这种情况下，React 将跳过渲染组件的操作并直接复用最近一次渲染的结果。

`React.memo` 仅检查 props 变更。如果函数组件被 `React.memo` 包裹，且其实现中拥有 `useState`，`useReducer` 或 `useContext` 的 Hook，当 state 或 context 发生变化时，它仍会重新渲染。

默认情况下其只会对复杂对象做浅层对比，如果你想要控制对比过程，那么请将自定义的比较函数通过第二个参数传入来实现。

```
function MyComponent(props) {
  /* 使用 props 渲染 */
}

function areEqual(prevProps, nextProps) {
  /*
  如果把 nextProps 传入 render 方法的返回结果与
  将 prevProps 传入 render 方法的返回结果一致则返回 true,
  否则返回 false
  */
}

export default React.memo(MyComponent, areEqual);
```

此方法仅作为[性能优化](#)的方式而存在。但请不要依赖它来“阻止”渲染，因为这会产生 bug。

注意

与 class 组件中 `shouldComponentUpdate()` 方法不同的是，如果 `props` 相等，`areEqual` 会返回 `true`；如果 `props` 不相等，则返回 `false`。这与 `shouldComponentUpdate` 方法的返回值相反。

`createElement()` [#createelement](#)

```
React.createElement(
  type,
  [props],
  [...children]
)
```

创建并返回指定类型的新 `React` 元素。其中的类型参数既可以是标签名字符串（如 `'div'` 或 `'span'`），也可以是 `React` 组件类型（class 组件或函数组件），或是 `React fragment` 类型。

使用 `JSX` 编写的代码将会被转换成使用 `React.createElement()` 的形式。如果使用了 `JSX` 方式，那么一般来说就不需要直接调用 `React.createElement()`。请查阅[不使用 JSX](#) 章节获得更多信息。

`cloneElement()` [#cloneelement](#)

```
React.cloneElement(
  element,
  [config],
  [...children]
)
```

以 `element` 元素为样板克隆并返回新的 `React` 元素。`config` 中应包含新的 `props`、`key` 或 `ref`。返回元素的 `props` 是将新的 `props` 与原始元素的 `props` 浅层合并后的结果。新的子元素将取代现有的子元素，如果在 `config` 中未出现 `key` 或 `ref`，那么原始元素的 `key` 和 `ref` 将被保留。

`React.cloneElement()` 几乎等同于：

```
<element.type {...element.props} {...props}>{children}</element.type>
```

但是，这也保留了组件的 `ref`。这意味着当通过 `ref` 获取子节点时，你将不会意外地从你祖先节点上窃取它。相同的 `ref` 将添加到克隆后的新元素中。如果存在新的 `ref` 或 `key` 将覆盖之前的。

引入此 API 是为了替换已弃用的 `React.addons.cloneWithProps()`。

`createFactory()` [#createfactory](#)

```
React.createFactory(type)
```

返回用于生成指定类型 React 元素的函数。与 `React.createElement()` 相似的是，类型参数既可以是标签名字符串（像是 `'div'` 或 `'span'`），也可以是 React 组件 类型（class 组件或函数组件），或是 React fragment 类型。

此辅助函数已废弃，建议使用 JSX 或直接调用 `React.createElement()` 来替代它。

如果你使用 JSX，通常不会直接调用 `React.createFactory()`。请参阅[不使用 JSX](#) 以获得更多信息。

`isValidElement()` [#isvalidelement](#)

```
React.isValidElement(object)
```

验证对象是否为 React 元素，返回值为 `true` 或 `false`。

`React.Children` [#reactchildren](#)

`React.Children` 提供了用于处理 `this.props.children` 不透明数据结构的实用方法。

`React.Children.map` [#reactchildrenmap](#)

```
React.Children.map(children, function[(thisArg)])
```

在 `children` 里的每个直接子节点上调用一个函数，并将 `this` 设置为 `thisArg`。如果 `children` 是一个数组，它将被遍历并为数组中的每个子节点调用该函数。如果子节点为 `null` 或是 `undefined`，则此方法将返回 `null` 或是 `undefined`，而不会返回数组。

注意

如果 `children` 是一个 Fragment 对象，它将被视为单一子节点的情况处理，而不会被遍历。

`React.Children.forEach` [#reactchildrenforeach](#)

```
React.Children.forEach(children, function[(thisArg)])
```

与 `React.Children.map()` 类似，但它不会返回一个数组。

React.Children.count {#reactchildrencount}**React.Children.count(children)**

返回 `children` 中的组件总数量，等同于通过 `map` 或 `forEach` 调用回调函数的次数。

React.Children.only {#reactchildrenonly}**React.Children.only(children)**

验证 `children` 是否只有一个子节点（一个 React 元素），如果有则返回它，否则此方法会抛出错误。

注意：

`React.Children.only()` 不接受 `React.Children.map()` 的返回值，因为它是一个数组而并不是 React 元素。

React.Children.toArray {#reactchildrentoarray}**React.Children.toArray(children)**

将 `children` 这个复杂的数据结构以数组的方式扁平展开并返回，并为每个子节点分配一个 `key`。当你想要在渲染函数中操作子节点的集合时，它会非常实用，特别是当你想要在向下传递 `this.props.children` 之前对内容重新排序或获取子集时。

注意：

`React.Children.toArray()` 在拉平展开子节点列表时，更改 `key` 值以保留嵌套数组的语义。也就是说，`toArray` 会为返回数组中的每个 `key` 添加前缀，以使得每个元素 `key` 的范围都限定在此函数入参数组的对象内。

React.Fragment {#reactfragment}

`React.Fragment` 组件能够在不额外创建 DOM 元素的情况下，让 `render()` 方法中返回多个元素。

```
render() {
  return (
    <React.Fragment>
      Some text.
      <h2>A heading</h2>
    </React.Fragment>
  );
}
```

你也可以使用其简写语法 `<></>`。欲了解更多相关信息，请参阅 [React v16.2.0: Fragments 支持改进](#)。

React.createRef {#reactcreateref}

`React.createRef` 创建一个能够通过 `ref` 属性附加到 React 元素的 `ref`。 embed:16-3-release-blog-post/create-ref-example.js

React.forwardRef {#reactforwardref}

`React.forwardRef` 会创建一个React组件，这个组件能够将其接受的 `ref` 属性转发到其组件树下的另一个组件中。这种技术并不常见，但在以下两种场景中特别有用：

- 转发 refs 到 DOM 组件
- 在高阶组件中转发 refs

`React.forwardRef` 接受渲染函数作为参数。React 将使用 `props` 和 `ref` 作为参数来调用此函数。此函数应返回 React 节点。

embed:reference-react-forward-ref.js

在上述的示例中，React 会将 `<FancyButton ref={ref}>` 元素的 `ref` 作为第二个参数传递给 `React.forwardRef` 函数中的渲染函数。该渲染函数会将 `ref` 传递给 `<button ref={ref}>` 元素。

因此，当 React 附加了 `ref` 属性之后，`ref.current` 将直接指向 `<button>` DOM 元素实例。

欲了解更多相关信息，请参阅 [refs 转发](#)。

React.lazy {#reactlazy}

`React.lazy()` 允许你定义一个动态加载的组件。这有助于缩减 bundle 的体积，并延迟加载在初次渲染时未用到的组件。

你可以在[代码分割文档](#)中学习如何使用它。查阅[此文章](#)可以了解更多用法细节。

```
// 这个组件是动态加载的
const SomeComponent = React.lazy(() => import('./SomeComponent'));
```

请注意，渲染 `lazy` 组件依赖该组件渲染树上层的 `<React.Suspense>` 组件。这是指定加载指示器（loading indicator）的方式。

React.Suspense {#reactsuspense}

`React.Suspense` 可以指定加载指示器（loading indicator），以防其组件树中的某些子组件尚未具备渲染条件。在未来，我们计划让 `Suspense` 处理更多的场景，如数据获取等。你可以在[我们的路线图](#)了解这一点。

如今，懒加载组件是 `<React.Suspense>` 支持的唯一用例：

```
// 该组件是动态加载的
const OtherComponent = React.lazy(() => import('./OtherComponent'));

function MyComponent() {
  return (
    // 显示 <Spinner> 组件直至 OtherComponent 加载完成
    <React.Suspense fallback={<Spinner />}>
      <div>
```

```

        <OtherComponent />
      </div>
    </React.Suspense>
  );
}

```

它已被收录在了我们的[代码分割指南](#)中。请注意，`lazy` 组件可以位于 `Suspense` 组件树的深处——它不必包装树中的每一个延迟加载组件。最佳实践是将 `<Suspense>` 置于你想展示加载指示器（loading indicator）的位置，而 `lazy()` 则可被放置于任何你想要做代码分割的地方。

注意

对于已经展示给用户的内容来说，在切换回去时，展示加载指示器可能会让人困惑。有时，在准备新的 UI 时，展示“旧”的 UI 可能会更加友好。要做到这一点，你可以使用新的 transition API `startTransition` 和 `useTransition` 来将标记更新为 `transitions`，同时避免意外的兜底方案。

服务端渲染中的 `React.Suspense` [#reactsuspense-in-server-side-rendering](#)

在服务端渲染过程中，`Suspense` 边界允许你挂起，通过较小的块来刷新应用程序。当组件挂起时，我们会安排一个低优先级的任务来渲染最近的 `Suspense` 边界的 fallback。如果组件在我们刷新 fallback 之前取消挂起，那么我们会发送实际内容并丢弃 fallback。

hydrate 过程中的 `React.Suspense` [#reactsuspense-during-hydration](#)

`Suspense` 边界依赖于它们的父边界，在它们可以 `hydrate` 前被 `hydrate`，但是它们可以独立于兄弟边界进行 `hydrate`。边界 `hydrate` 前发生的事件将导致边界 `hydrate` 的优先级高于相邻边界的优先级。具体请参阅[讨论](#)。

`React.startTransition` [#starttransition](#)

`React.startTransition(callback)`

`React.startTransition` 让你把提供的 `fallback` 里面的更新标记为 `transitions`。这个方法是为了在 `React.useTransition` 不可用时使用。

注意：

过渡期的更新会被更紧急的更新取代，如点击操作。

过渡期的更新不会显示重新挂起内容的 `fallback`，允许用户在渲染更新时继续进行交互。

`React.startTransition` 不提供 `isPending` 的标志。要跟踪过渡的待定状态，请参阅 `React.useTransition`。

Test Renderer

如何引入

```
import TestRenderer from 'react-test-renderer'; // ES6
const TestRenderer = require('react-test-renderer'); // ES5 with npm
```

概览

这个 package 提供了一个 React 渲染器，用于将 React 组件渲染成纯 JavaScript 对象，无需依赖 DOM 或原生移动环境。

这个 package 提供的主要功能是在不依赖浏览器或 `jsdom` 的情况下，返回某个时间点由 React DOM 或者 React Native 平台渲染出的视图结构（类似与 DOM 树）快照。

示例：

```
import TestRenderer from 'react-test-renderer';

function Link(props) {
  return <a href={props.page}>{props.children}</a>;
}

const testRenderer = TestRenderer.create(
  <Link page="https://www.facebook.com/">Facebook</Link>
);

console.log(testRenderer.toJSON());
// { type: 'a',
//   props: { href: 'https://www.facebook.com/' },
//   children: [ 'Facebook' ] }
```

你可以使用 Jest 的快照测试功能来自动保存当前 `JSON` 树结构到一个文件中，并在测试中检查它是否被修改：[了解更多](#)。

你也可以通过遍历输出来查找特定节点，并对它们进行断言。

```
import TestRenderer from 'react-test-renderer';

function MyComponent() {
  return (
    <div>
      <SubComponent foo="bar" />
      <p className="my">Hello</p>
    </div>
  )
}

function SubComponent() {
  return (
    <p className="sub">Sub</p>
  );
}

const testRenderer = TestRenderer.create(<MyComponent />);
```

```
const testInstance = testRenderer.root;
expect(testInstance.findByType(SubComponent).props.foo).toBe('bar');
expect(testInstance.findByProps({className: "sub"}).children).toEqual(['Sub']);
```

TestRenderer

- `TestRenderer.create()`
- `TestRenderer.act()`

TestRenderer instance

- `testRenderer.toJSON()`
- `testRenderer.toTree()`
- `testRenderer.update()`
- `testRenderer.unmount()`
- `testRenderer.getInstance()`
- `testRenderer.root`

TestInstance

- `testInstance.find()`
- `testInstance.findByType()`
- `testInstance.findByProps()`
- `testInstance.findAll()`
- `testInstance.findAllByType()`
- `testInstance.findAllByProps()`
- `testInstance.instance`
- `testInstance.type`
- `testInstance.props`
- `testInstance.parent`
- `testInstance.children`

参考

TestRenderer.create() `{#testrenderercreate}`

```
TestRenderer.create(element, options);
```

通过传来的 React 元素创建一个 `TestRenderer` 实例。它并不使用真实的 DOM，但是它依然将组件树完整地渲染到内存，以便于你对它进行断言。此时将返回一个 `TestRenderer` 实例。

TestRenderer.act() `{#testrendereract}`

```
TestRenderer.act(callback);
```

与 `react-dom/test-utils` 中的 `act()` 相似，`TestRender.act` 为断言准备一个组件。可以使用 `act()` 来包装 `TestRenderer.create` 和 `testRenderer.update`。

```

import {create, act} from 'react-test-renderer';
import App from './app.js'; // The component being tested

// 渲染组件
let root;
act(() => {
  root = create(<App value={1}>/)
});

// 对根元素进行断言
expect(root.toJSON()).toMatchSnapshot();

// 更新 props
act(() => {
  root.update(<App value={2}>);
})

// 对根元素进行断言
expect(root.toJSON()).toMatchSnapshot();

```

testRenderer.toJSON() {#testrenderertojson}

`testRenderer.toJSON()`

返回一个已渲染的树对象。该树仅包含特定平台的节点，例如 `<div>` 或 `<View>` 和它们的 props，但并不包含任何用户编写的组件。这对于快照测试非常方便。

testRenderer.toTree() {#testrenderertotree}

`testRenderer.toTree()`

返回一个已渲染的树对象。它所表示的内容比 `toJSON()` 提供的内容要更加详细，并且包含用户编写的组件。除非你要在测试渲染器 (test renderer) 之上编写自己的断言库，否则你可能并不需要这个方法。

testRenderer.update() {#testrendererupdate}

`testRenderer.update(element)`

使用新的根元素重新渲染内存中的树。它模拟根元素的一次 React 更新。如果新的元素和之前的元素有相同的 type 和 key，该树将被更新；否则，它将重挂载一个新树。

testRenderer.unmount() {#testrendererunmount}

`testRenderer.unmount()`

卸载内存中的树，会触发相应的生命周期事件。

testRenderer.getInstance() {#testrenderergetinstance}

`testRenderer.getInstance()`

如果可用的话，返回与根元素相对应的实例。如果根元素是函数定义组件，该方法无效，因为函数定义组件没有实例。

testRenderer.root {#testrendererroot}**testRenderer.root**

返回根元素“测试实例”对象，它对于断言树中的特定节点十分有用。你可以利用它来查找其他更深层的“测试实例”。

testInstance.find() {#testinstancefind}**testInstance.find(test)**

找到一个 `test(testInstance)` 返回 `true` 的后代测试实例。如果不只有一个测试实例匹配，将会报错。

testInstance.findByType() {#testinstancefindbytype}**testInstance.findByType(type)**

找到匹配指定 `type` 的后代测试实例，如果不是只有一个测试实例匹配指定的 `type`，将会报错。

testInstance.findByProps() {#testinstancefindbyprops}**testInstance.findByProps(props)**

找到匹配指定 `props` 的后代测试实例，如果不是正好只有一个测试实例匹配指定的 `props`，将会报错。

testInstance.findAll() {#testinstancefindall}**testInstance.findAll(test)**

找到所有 `test(testInstance)` 返回 `true` 的后代测试实例。

testInstance.findAllByType() {#testinstancefindallbytype}**testInstance.findAllByType(type)**

找到所有匹配指定 `type` 的后代测试实例。

testInstance.findAllByProps() {#testinstancefindallbyprops}**testInstance.findAllByProps(props)**

找到所有匹配指定 `props` 的后代测试实例。

testInstance.instance {#testinstanceinstance}**testInstance.instance**

该测试实例相对应的组件实例。它只能用于类定义组件，因为函数定义组件没有实例。它匹配给定的组件内部的 `this` 的值。

testInstance.type {#testinstancetype}

testInstance.type

该测试实例相对应的组件的类型。例如，一个 `<Button />` 组件有一个 `Button` 类型。

testInstance.props {#testinstanceprops}

testInstance.props

该测试实例相对应的组件的 props。例如，一个 `<Button size="small" />` 组件的 props 为 `{size: 'small'}`。

testInstance.parent {#testinstanceparent}

testInstance.parent

该测试实例的父测试实例。

testInstance.children {#testinstancechildren}

testInstance.children

该测试实例的子测试实例。

想法

你可以把 `createNodeMock` 函数作为选项 (option) 传递给 `TestRenderer.create`，进行自定义 refs 模拟。

`createNodeMock` 接受当前元素作为参数，并且返回一个模拟 ref 对象的。这十分有利于依赖 refs 组件的测试。

```
import TestRenderer from 'react-test-renderer';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.input = null;
  }
  componentDidMount() {
    this.input.focus();
  }
  render() {
    return <input type="text" ref={el => this.input = el} />
  }
}

let focused = false;
TestRenderer.create(
  <MyComponent />,
  {
    createNodeMock: (element) => {
      if (element.type === 'input') {
        // 模拟 focus 函数
        return {
          focus: () => {
            focused = true;
          }
        };
      }
    }
  }
);
```

```
    };
}
return null;
}
);
expect(focused).toBe(true);
```

Refs and the DOM

Refs 提供了一种方式，允许我们访问 DOM 节点或在 render 方法中创建的 React 元素。

在典型的 React 数据流中，`props` 是父组件与子组件交互的唯一方式。要修改一个子组件，你需要使用新的 `props` 来重新渲染它。但是，在某些情况下，你需要在典型数据流之外强制修改子组件。被修改的子组件可能是一个 React 组件的实例，也可能是一个 DOM 元素。对于这两种情况，React 都提供了解决办法。

何时使用 Refs

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。

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。

勿过度使用 Refs

你可能首先会想到使用 `refs` 在你的 app 中“让事情发生”。如果是这种情况，请花一点时间，认真再考虑一下 `state` 属性应该被安排在哪个组件层中。通常你会想明白，让更高的组件层级拥有这个 `state`，是更恰当的。查看 [状态提升](#) 以获取更多有关示例。

注意

下面的例子已经更新为使用在 React 16.3 版本引入的 `React.createRef()` API。如果你正在使用一个较早版本的 React，我们推荐你使用[回调形式的 refs](#)。

创建 Refs

Refs 是使用 `React.createRef()` 创建的，并通过 `ref` 属性附加到 React 元素。在构造组件时，通常将 Refs 分配给实例属性，以便可以在整个组件中引用它们。

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.myRef = React.createRef();
  }
  render() {
    return <div ref={this.myRef} />;
  }
}
```

访问 Refs

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。

```
const node = this.myRef.current;
```

`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。

以下例子说明了这些差异。

为 DOM 元素添加 ref

以下代码使用 `ref` 去存储 DOM 节点的引用：

```
class CustomTextInput extends React.Component {
  constructor(props) {
    super(props);
    // 创建一个 ref 来存储 textInput 的 DOM 元素
    this.textInput = React.createRef();
    this.focusTextInput = this.focusTextInput.bind(this);
  }

  focusTextInput() {
    // 直接使用原生 API 使 text 输入框获得焦点
    // 注意：我们通过 "current" 来访问 DOM 节点
    this.textInput.current.focus();
  }

  render() {
    // 告诉 React 我们想把 <input> ref 关联到
    // 构造器里创建的 `textInput` 上
    return (
      <div>
        <input
          type="text"
          ref={this.textInput} />
        <input
          type="button"
          value="Focus the text input"
          onClick={this.focusTextInput}
        />
      </div>
    );
  }
}
```

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。

为 class 组件添加 Ref

如果我们想包装上面的 `CustomTextInput`，来模拟它挂载之后立即被点击的操作，我们可以使用 `ref` 来获取这个自定义的 `input` 组件并手动调用它的 `focusTextInput` 方法：

```
class AutoFocusTextInput extends React.Component {
  constructor(props) {
    super(props);
    this.textInput = React.createRef();
  }

  componentDidMount() {
    this.textInput.current.focusTextInput();
  }

  render() {
    return (
      <CustomTextInput ref={this.textInput} />
    );
  }
}
```

请注意，这仅在 `CustomTextInput` 声明为 `class` 时才有效：

```
class CustomTextInput extends React.Component {
  // ...
}
```

Refs 与函数组件

默认情况下，你不能在函数组件上使用 `ref` 属性，因为它们没有实例：

```
function MyFunctionComponent() {
  return <input />;
}

class Parent extends React.Component {
  constructor(props) {
    super(props);
    this.textInput = React.createRef();
  }

  render() {
    // This will *not* work!
    return (
      <MyFunctionComponent ref={this.textInput} />
    );
  }
}
```

如果要在函数组件中使用 `ref`，你可以使用 `forwardRef`（可与 `useImperativeHandle` 结合使用），或者可以将该组件转化为 `class` 组件。

不管怎样，你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：

```
function CustomTextInput(props) {
  // 这里必须声明 textInput，这样 ref 才可以引用它
  const textInput = useRef(null);
```

```

function handleClick() {
 textInput.current.focus();
}

return (
  <div>
    <input
      type="text"
      ref={textInput} />
    <input
      type="button"
      value="Focus the text input"
      onClick={handleClick}
    />
  </div>
);
}

```

将 DOM Refs 暴露给父组件

在极少数情况下，你可能希望在父组件中引用子节点的 DOM 节点。通常不建议这样做，因为它会打破组件的封装，但它偶尔可用于触发焦点或测量子 DOM 节点的大小或位置。

虽然你可以[向子组件添加 ref](#)，但这不是一个理想的解决方案，因为你只能获取组件实例而不是 DOM 节点。并且，它还在函数组件上无效。

如果你使用 16.3 或更高版本的 React，这种情况下我们推荐使用[ref 转发](#)。**Ref 转发使组件可以像暴露自己的 ref 一样暴露子组件的 ref**。关于怎样对父组件暴露子组件的 DOM 节点，在[ref 转发文档](#)中有一个详细的例子。

如果你使用 16.2 或更低版本的 React，或者你需要比 ref 转发更高的灵活性，你可以使用[这个替代方案](#)将 ref 作为特殊名字的 prop 直接传递。

可能的话，我们不建议暴露 DOM 节点，但有时候它会成为救命稻草。注意这个方案需要你在子组件中增加一些代码。如果你对子组件的实现没有控制权的话，你剩下的选择是使用[findDOMNode\(\)](#)，但在严格模式下已被废弃且不推荐使用。

回调 Refs

React 也支持另一种设置 refs 的方式，称为“回调 refs”。它能助你更精细地控制何时 refs 被设置和解除。

不同于传递 `createRef()` 创建的 `ref` 属性，你会传递一个函数。这个函数中接受 React 组件实例或 HTML DOM 元素作为参数，以使它们能在其他地方被存储和访问。

下面的例子描述了一个通用的范例：使用 `ref` 回调函数，在实例的属性中存储对 DOM 节点的引用。

```

class CustomTextInput extends React.Component {
  constructor(props) {
    super(props);

    this.textInput = null;

    this.setTextInputRef = element => {
      this.textInput = element;
    };

    this.focusTextInput = () => {
      // 使用原生 DOM API 使 text 输入框获得焦点
    };
}

```

```

        if (this.TextInput) this.TextInput.focus();
    };

}

componentDidMount() {
    // 组件挂载后，让文本框自动获得焦点
    this.focusTextInput();
}

render() {
    // 使用 `ref` 的回调函数将 text 输入框 DOM 节点的引用存储到 React
    // 实例上（比如 this.TextInput）
    return (
        <div>
            <input
                type="text"
                ref={this.setTextInputRef}
            />
            <input
                type="button"
                value="Focus the text input"
                onClick={this.focusTextInput}
            />
        </div>
    );
}
}

```

React 将在组件挂载时，会调用 `ref` 回调函数并传入 DOM 元素，当卸载时调用它并传入 `null`。在 `componentDidMount` 或 `componentDidUpdate` 触发前，React 会保证 refs 一定是最新的。

你可以在组件间传递回调形式的 refs，就像你可以传递通过 `React.createRef()` 创建的对象 refs 一样。

```

function CustomTextInput(props) {
    return (
        <div>
            <input ref={props.inputRef} />
        </div>
    );
}

class Parent extends React.Component {
    render() {
        return (
            <CustomTextInput
                inputRef={el => this.inputElement = el}
            />
        );
    }
}

```

在上面的例子中，`Parent` 把它的 refs 回调函数当作 `inputRef` props 传递给了 `CustomTextInput`，而且 `CustomTextInput` 把相同的函数作为特殊的 `ref` 属性传递给了 `<input>`。结果是，在 `Parent` 中的 `this.inputElement` 会被设置为与 `CustomTextInput` 中的 `input` 元素相对应的 DOM 节点。

过时 API: String 类型的 Refs

如果你之前使用过 React，你可能了解过之前的 API 中的 string 类型的 ref 属性，例如 `"textInput"`。你可以通过 `this.refs.textInput` 来访问 DOM 节点。我们不建议使用它，因为 string 类型的 refs 存在 [一些问题](#)。它已过时并可能会在未来的版本被移除。

注意

如果你目前还在使用 `this.refs.textInput` 这种方式访问 refs，我们建议用[回调函数](#)或 [createRef API](#) 的方式代替。

关于回调 refs 的说明

如果 `ref` 回调函数是以内联函数的方式定义的，在更新过程中它会被执行两次，第一次传入参数 `null`，然后第二次会传入参数 DOM 元素。这是因为在每次渲染时会创建一个新的函数实例，所以 React 清空旧的 ref 并且设置新的。通过将 ref 的回调函数定义成 class 的绑定函数的方式可以避免上述问题，但是大多数情况下它是无关紧要的。

发布渠道

React 依靠强大的开源社区收集 bug 报告，发起 pull request 和 [提交 RFC](#)。为了鼓励大家反馈，我们打算共享一些特殊的 React 版本，其中包括未发布的功能。

此文章与从事框架、库或开发工具的开发人员息息相关。而主要使用 React 来构建应用程序的开发者无需担心此预发布渠道。

React 的每个发布渠道都是针对不同的用例进行设计的：

- **最新**版本用于稳定的 semver React 版本。此版本可通过 npm 安装获取。此渠道为目前大家已经在用的方式。**其主要用于所有面向用户的 React 应用程序。**
- **Next** 版本主要用于追踪 React 源码仓库的 main 分支。我们会将其视为下一个次要版本发布的候选版本。使用它可以进行 React 与第三方项目间的集成测试。
- **实验阶段**版本包含稳定版本中不提供的实验阶段的 API 与功能。同时它也追踪了 main 分支，但启用了附加新功能的标志。使用此渠道可以尝试即将发布的功能。

所有版本都将发布到 npm，但只有最新版本遵循[语义版本控制](#)。预发布版本（应用于 Next 和实验渠道的版本）会根据其内容的哈希值和提交日期生成版本号，例如，Next 的版本为 `0.0.0-68053d940-20210623`，实验版则为 `0.0.0-experimental-68053d940-20210623`。

最新版是面向用户应用程序的唯一官方支持发布渠道。提供 Next 和实验版本的目的是用于测试，我们并不保证功能在这两个版本中不发生变化。因为它们并不遵循用于最新版发布的 semver 协议。

将预发布版本发布到与稳定版本相同的注册表，我们可以利用许多支持 npm 工作流的工具，比如：[unpkg](#) 和 [CodeSandbox](#)。

最新版渠道

最新版是用于稳定 React 版本的渠道。它对应是 npm 中 `latest` 标签。此版本是所有交付给真实用户的 React 应用程序的推荐版本。

如果你不确定应该使用哪个版本，那就用最新版。如果你是 React 开发人员，那么这就是你正确的选择。

你可以认为最新版的更新是非常稳定的。版本遵循语义版本控制方案。在[版本政策](#)中了解更多关于我们对稳定性和增量迁移的承诺。

Next 渠道

Next 属于预发布渠道，用于追踪 React 仓库的 main 分支。我们使用在 Next 渠道的预发布版本作为最新版发布渠道的候选版本。你可以将 Next 视为最新版的超集，它的更新频率更高。

最近的 Next 版本和最近的最新版本之间的变化程度，与两个次要的 semver 版本之间的变化程度大致相同。但是，**Next 渠道不遵循语义版本控制。**在 Next 渠道中，你应该预期到后续的版本中偶尔会有不兼容的改动。

不要在面向用户的应用程序中使用预发布版本。

Next 渠道中的预发布版本在 npm 中携带 `next` 标签发布。版本号是根据其构建内容的哈希值和提交日期生成的，例如：
`0.0.0-68053d940-20210623`。

使用 Next 渠道进行集成测试

Next 渠道用于支持 React 与其他项目之间的集成测试。

React 的所有更改在发布之前都要经过大量的内部测试。然而，React 的整个生态系统使用了无数的环境和配置，我们不可能针对每一项进行测试。

如果你是 React 第三方框架、库、开发者工具或类似基础设施项目的作者，则可以通过定期针对最新版本运行的测试用例，帮助我们一起维持 React 稳定，为你的用户和整个 React 社区保驾护航。如果你对此有兴趣，请按照下列步骤进行操作：

- 在你喜欢的持续集成平台上设置 cron job。[CircleCI](#) 和 [Travis CI](#) 均支持 cron job。
- 在 cron job 中，使用 npm 的 `next` 标签将 React 版本更新至 Next 渠道中的最新版本。使用 npm cli：

```
npm update react@next react-dom@next
```

或者 yarn：

```
yarn upgrade react@next react-dom@next
```

- 针对更新的 packages 运行你的测试用例。
- 如果均通过，那么恭喜你！你的项目可以与下个小版本的 React 一起使用。
- 如果发生意外中断，请通过[提交 issues](#) 告知我们。

Next.js 项目使用了这个工作流。你可以参考他们的 [CircleCI 配置](#)作为示例。

实验版渠道

与 Next 相似，实验版渠道是一个预发布渠道，用于追踪 React 仓库 main 分支。但不同于 Next 的是，实验版包含尚未准备好广泛推广的功能及 API。

通常，对 Next 更新时也会对实验版本进行更新。它们基于相同的源，但是构建时会使用不同的功能标记。

实验阶段发布的版本可能与 Next 和最新版本的发布均不相同。**不要在面向用户的应用程序中使用实验版。** 你应该能够想象到实验渠道中发布的版本会频繁进行破坏性的更新。

实验版本会在 npm 上会以 `experimental` 标签的形式发布。版本会根据构建内容的哈希值和提交日期生成，例如，
`0.0.0-experimental-68053d940-20210623`。

实验阶段发布包含哪些内容？

实验阶段的功能并未打算公开发布，在最终确定之前可能会发生重大变化。有些实验功能可能永远不会完成——我们进行实验的目的是为了测试变更提案的可行性。

例如，如果我们在宣布发布 Hook 时，其已经在实验渠道中存在，我们会在最新版本发布 Hook 的前几周，将其发布到实验渠道中。

你可能会发现在实验阶段进行集成测试是很有必要的。但是，请注意，实验阶段版本的稳定性不如 Next 版本。**我们并不保证实验版本之间的稳定性。**

如何了解有关实验功能的更多信息？

实验性的功能可能会有文档，也可能不会有文档。通常，在实验渠道的内容发布到 Next 或 Latest 中之前，才会编写文档。

如果找不到文档，则可能会附有 [RFC](#) 说明。

当我们准备发布新的实验内容时，我们会发布到 [React 博客](#) 中，但这并不意味着我们将公开发布每个实验的内容。

你可以参考 Github 公开仓库的[历史记录](#)以查看完整的变更列表。

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 的组件接受一个返回 React 元素的函数，并在组件内部通过调用此函数来实现自己的渲染逻辑。

```
<DataProvider render={data => (
  <h1>Hello {data.target}</h1>
)} />
```

使用 render prop 的库有 [React Router](#)、[Downshift](#) 以及 [Formik](#)。

在这个文档中，我们将讨论为什么 render prop 是有用的，以及如何写一个自己的 render prop 组件。

使用 Render Props 来解决横切关注点 (Cross-Cutting Concerns)

组件是 React 代码复用的主要单元，但如何将一个组件封装的状态或行为共享给其他需要相同状态的组件并不总是显而易见。

例如，以下组件跟踪 Web 应用程序中的鼠标位置：

```
class MouseTracker extends React.Component {
  constructor(props) {
    super(props);
    this.handleMouseMove = this.handleMouseMove.bind(this);
    this.state = { x: 0, y: 0 };
  }

  handleMouseMove(event) {
    this.setState({
      x: event.clientX,
      y: event.clientY
    });
  }

  render() {
    return (
      <div style={{ height: '100vh' }} onMouseMove={this.handleMouseMove}>
        <h1>移动鼠标!</h1>
        <p>当前的鼠标位置是 ({this.state.x}, {this.state.y})</p>
      </div>
    );
  }
}
```

当光标在屏幕上移动时，组件在 `<p>` 中显示其 (x, y) 坐标。

现在的问题是：我们如何在另一个组件中复用这个行为？换个说法，若另一个组件需要知道鼠标位置，我们能否封装这一行为，以便轻松地与其他组件共享它？

由于组件是 React 中最基础的代码复用单元，现在尝试重构一部分代码使其能够在 `<Mouse>` 组件中封装我们需要共享的行为。

```
// <Mouse> 组件封装了我们需要的行为...
class Mouse extends React.Component {
  constructor(props) {
    super(props);
    this.handleMouseMove = this.handleMouseMove.bind(this);
    this.state = { x: 0, y: 0 };
  }

  handleMouseMove(event) {
    this.setState({
      x: event.clientX,
      y: event.clientY
    });
  }

  render() {
    return (
      <div style={{ height: '100vh' }} onMouseMove={this.handleMouseMove}>
        {/* ...但我们如何渲染 <p> 以外的东西? */}
        <p>The current mouse position is ({this.state.x}, {this.state.y})</p>
      </div>
    );
  }
}

class MouseTracker extends React.Component {
  render() {
    return (
      <>
        <h1>移动鼠标!</h1>
        <Mouse />
      </>
    );
  }
}
```

现在 `<Mouse>` 组件封装了所有关于监听 `mousemove` 事件和存储鼠标 (x, y) 位置的行为，但其仍不是真正的可复用。

举个例子，假设我们有一个 `<Cat>` 组件，它可以呈现一张在屏幕上追逐老鼠的猫的图片。我们或许会使用 `<Cat mouse={{ x, y }}>` prop 来告诉组件鼠标的坐标以让它知道图片应该在屏幕哪个位置。

首先，你或许会像这样，尝试在 `<Mouse>` 内部的渲染方法渲染 `<Cat>` 组件：

```
class Cat extends React.Component {
  render() {
    const mouse = this.props.mouse;
    return (
      
    );
  }
}

class MouseWithCat extends React.Component {
  constructor(props) {
```

```

super(props);
this.handleMouseMove = this.handleMouseMove.bind(this);
this.state = { x: 0, y: 0 };
}

handleMouseMove(event) {
  this.setState({
    x: event.clientX,
    y: event.clientY
  });
}

render() {
  return (
    <div style={{ height: '100vh' }} onMouseMove={this.handleMouseMove}>
      /*
        在这里，我们可以简单地使用 <Cat> 来替换 <p>。但是如果这样
        做的话，当我们每次遇到这样的情况时，就需要创建一个单独的
        <MouseWithSomethingElse>。所以 <MouseWithCat> 并不是真正的可复用组件。
      */
      <Cat mouse={this.state} />
    </div>
  );
}

class MouseTracker extends React.Component {
  render() {
    return (
      <div>
        <h1>移动鼠标!</h1>
        <MouseWithCat />
      </div>
    );
  }
}

```

这种方法适用于我们的特定用例，但我们还没有达到以可复用的方式真正封装行为的目标。现在，每当我们想要鼠标位置用于不同的用例时，我们必须创建一个新的组件（本质上是另一个 `<MouseWithCat>`），它专门为该用例呈现一些东西。

这也是 render prop 的来历：相比于直接将 `<Cat>` 写死在 `<Mouse>` 组件中，并且有效地更改渲染的结果，我们可以为 `<Mouse>` 提供一个函数 prop 来动态的确定要渲染什么——一个 render prop。

```

class Cat extends React.Component {
  render() {
    const mouse = this.props.mouse;
    return (
      
    );
  }
}

class Mouse extends React.Component {
  constructor(props) {
    super(props);
    this.handleMouseMove = this.handleMouseMove.bind(this);
    this.state = { x: 0, y: 0 };
  }
}

```

```

handleMouseMove(event) {
  this.setState({
    x: event.clientX,
    y: event.clientY
  });
}

render() {
  return (
    <div style={{ height: '100vh' }} onMouseMove={this.handleMouseMove}>
      /*
        使用 `render` prop 动态决定要渲染的内容,
        而不是给出一个 <Mouse> 渲染结果的静态表示
      */
      {this.props.render(this.state)}
    </div>
  );
}
}

class MouseTracker extends React.Component {
  render() {
    return (
      <div>
        <h1>移动鼠标!</h1>
        <Mouse render={mouse => (
          <Cat mouse={mouse} />
        )}>
        </div>
      );
    }
}

```

现在，我们提供了一个 `render` 方法 让 `<Mouse>` 能够动态决定什么需要渲染，而不是克隆 `<Mouse>` 组件然后硬编码来解决特定的用例。

更具体地说，**render prop 是一个用于告知组件需要渲染什么内容的函数 prop**。

这项技术使我们共享行为非常容易。要获得这个行为，只要渲染一个带有 `render` prop 的 `<Mouse>` 组件就能够告诉它当前鼠标坐标 (x, y) 要渲染什么。

关于 `render prop` 一个有趣的事情是你可以使用带有 `render prop` 的常规组件来实现大多数高阶组件 (HOC)。例如，如果你更喜欢使用 `withMouse` HOC 而不是 `<Mouse>` 组件，你可以使用带有 `render prop` 的常规 `<Mouse>` 轻松创建一个：

```

// 如果你出于某种原因真的想要 HOC，那么你可以轻松实现
// 使用具有 render prop 的普通组件创建一个!
function withMouse(Component) {
  return class extends React.Component {
    render() {
      return (
        <Mouse render={mouse => (
          <Component {...this.props} mouse={mouse} />
        )}>
      );
    }
  }
}

```

因此，你可以将任一模式与 render prop 一起使用。

使用 Props 而非 render {#using-props-other-than-render}

重要的是要记住，render prop 是因为模式才被称为 render prop，你不一定要用名为 `render` 的 prop 来使用这种模式。事实上，[任何被用于告知组件需要渲染什么内容的函数 prop 在技术上都可以被称为“render prop”](#)。

尽管之前的例子使用了 `render`，我们也可以简单地使用 `children` prop！

```
<Mouse children={mouse => (
  <p>鼠标的位置是 {mouse.x}, {mouse.y}</p>
)}>
```

记住，`children` prop 并不真正需要添加到 JSX 元素的“attributes”列表中。相反，你可以直接放置到元素的内部！

```
<Mouse>
  {mouse => (
    <p>鼠标的位置是 {mouse.x}, {mouse.y}</p>
  )}
</Mouse>
```

你将在 [react-motion](#) 的 API 中看到此技术。

由于这一技术的特殊性，当你在设计一个类似的 API 时，你或许会要直接地在你的 `propTypes` 里声明 `children` 的类型应为一个函数。

```
Mouse.propTypes = {
  children: PropTypes.func.isRequired
};
```

注意事项

将 Render Props 与 React.PureComponent 一起使用时要小心

如果你在 `render` 方法里创建函数，那么使用 render prop 会抵消使用 `React.PureComponent` 带来的优势。因为浅比较 props 的时候总会得到 `false`，并且在这种情况下每一个 `render` 对于 render prop 将会生成一个新的值。

例如，继续我们之前使用的 `<Mouse>` 组件，如果 `Mouse` 继承自 `React.PureComponent` 而不是 `React.Component`，我们的例子看起来就像这样：

```
class Mouse extends React.PureComponent {
  // 与上面相同的代码.....
}

class MouseTracker extends React.Component {
  render() {
    return (
      <div>
        <h1>Move the mouse around!</h1>
        {/* 这是不好的！ */}
      </div>
    );
  }
}
```

```

    每个渲染的 `render` prop 的值将会是不同的。
  */
  <Mouse render={mouse => (
    <Cat mouse={mouse} />
  )}>
  </div>
);
}
}

```

在这样例子中，每次 `<MouseTracker>` 渲染，它会生成一个新的函数作为 `<Mouse render>` 的 prop，因而在同时也抵消了继承自 `React.PureComponent` 的 `<Mouse>` 组件的效果！

为了绕过这一问题，有时你可以定义一个 prop 作为实例方法，类似这样：

```

class MouseTracker extends React.Component {
  // 定义为实例方法，`this.renderTheCat`始终
  // 当我们在渲染中使用它时，它指的是相同的函数
  renderTheCat(mouse) {
    return <Cat mouse={mouse} />;
  }

  render() {
    return (
      <div>
        <h1>Move the mouse around!</h1>
        <Mouse render={this.renderTheCat} />
      </div>
    );
  }
}

```

如果你无法静态定义 prop（例如，因为你需要控制组件 props 和/或 state 的暴露程度），则 `<Mouse>` 应该继承自 `React.Component`。

元素渲染

元素是构成 React 应用的最小砖块。

元素描述了你在屏幕上想看到的内容。

```
const element = <h1>Hello, world</h1>;
```

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。

注意：

你可能会将元素与另一个被熟知的概念——“组件”混淆起来。我们会在[下一个章节](#)介绍组件。组件是由元素构成的。我们强烈建议你不要觉得繁琐而跳过本章节，应当深入阅读这一章节。

将一个元素渲染为 DOM

假设你的 HTML 文件某处有一个 `<div>`：

```
<div id="root"></div>
```

我们将其称为“根”DOM 节点，因为该节点内的所有内容都将由 React DOM 管理。

仅使用 React 构建的应用通常只有单一的根 DOM 节点。如果你在将 React 集成进一个已有应用，那么你可以在应用中包含任意多的独立根 DOM 节点。

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.createRoot()`：

```
embed:rendering-elements/render-an-element.js
```

[在 CodePen 上试试](#)

页面上会展示出 "Hello, world"。

更新已渲染的元素

React 元素是[不可变对象](#)。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。

根据我们已有的知识，更新 UI 唯一的方式是创建一个全新的元素，并将其传入 `root.render()`。

考虑一个计时器的例子：

```
embed:rendering-elements/update-rendered-element.js
```

[在 CodePen 上试试](#)

这个例子会在 `setInterval()` 回调函数，每秒都调用 `root.render()`。

注意：

在实践中，大多数 React 应用只会调用一次 `root.render()`。在下一个章节，我们将学习如何将这些代码封装到有状态组件中。

我们建议你不要跳跃着阅读，因为每个话题都是紧密联系的。

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。

你可以通过查看 [上一个例子](#) 来确认这一点。

Hello, world!

It is 12:26:46 PM.

```

Console Sources Network Timeline
▼<div id="root">
  ▼<div data-reactroot>
    <h1>Hello, world!</h1>
    ▼<h2>
      <!-- react-text: 4 -->
      "It is "
      <!-- /react-text -->
      <!-- react-text: 5 -->
      "12:26:46 PM"
      <!-- /react-text -->
      <!-- react-text: 6 -->
      ""
      <!-- /react-text -->
    </h2>
  </div>
</div>

```

DOM inspector showing granular updates

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。

根据我们的经验，应该专注于 UI 在任意给定时刻的状态，而不是一视同仁地随着时间修改整个界面。

[Go to TOC](#)

State & 生命周期

本页面介绍了 React 组件中 state 和生命周期的概念。你可以查阅[详细的组件 API 参考文档](#)。

请参考[前一章节](#)中时钟的例子。在[元素渲染](#)章节中，我们只了解了一种更新 UI 界面的方法。通过调用 `root.render()` 来修改我们想要渲染的元素：

```
const root = ReactDOM.createRoot(document.getElementById('root'));

function tick() {
  const element = (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {new Date().toLocaleTimeString()}.</h2>
    </div>
  );
  root.render(element);
}

setInterval(tick, 1000);
```

[在 CodePen 上尝试](#)

在本章节中，我们将学习如何封装真正可复用的 `Clock` 组件。它将设置自己的计时器并每秒更新一次。

我们可以从封装时钟的外观开始：

```
const root = ReactDOM.createRoot(document.getElementById('root'));

function Clock(props) {
  return (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {props.date.toLocaleTimeString()}.</h2>
    </div>
  );
}

function tick() {
  root.render(<Clock date={new Date()} />);
}

setInterval(tick, 1000);
```

[在 CodePen 上尝试](#)

然而，它忽略了一个关键的技术细节：`Clock` 组件需要设置一个计时器，并且需要每秒更新 UI。

理想情况下，我们希望只编写一次代码，便可以让 `Clock` 组件自我更新：

```
root.render(<Clock />);
```

我们需要在 `Clock` 组件中添加 "state" 来实现这个功能。

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

将函数组件转换成 class 组件

通过以下五步将 `clock` 的函数组件转成 class 组件：

1. 创建一个同名的 `ES6 class`，并且继承于 `React.Component`。
2. 添加一个空的 `render()` 方法。
3. 将函数体移动到 `render()` 方法之中。
4. 在 `render()` 方法中使用 `this.props` 替换 `props`。
5. 删除剩余的空函数声明。

```
class Clock extends React.Component {
  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.props.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

[在 CodePen 上尝试](#)

现在 `Clock` 组件被定义为 class，而不是函数。

每次组件更新时 `render` 方法都会被调用，但只要在相同的 DOM 节点中渲染 `<Clock />`，就仅有一个 `Clock` 组件的 class 实例被创建使用。这就使得我们可以使用如 state 或生命周期方法等很多其他特性。

向 class 组件中添加局部的 state

我们通过以下三步将 `date` 从 props 移动到 state 中：

1. 把 `render()` 方法中的 `this.props.date` 替换成 `this.state.date`：

```
class Clock extends React.Component {
  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

2. 添加一个 `class 构造函数`，然后在该函数中为 `this.state` 赋初值：

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

通过以下方式将 `props` 传递到父类的构造函数中：

```
constructor(props) {
  super(props);
  this.state = {date: new Date()};
}
```

Class 组件应该始终使用 `props` 参数来调用父类的构造函数。

3. 移除 `<Clock />` 元素中的 `date` 属性：

```
root.render(<Clock />);
```

我们之后会将计时器相关的代码添加到组件中。

代码如下：

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Clock />);
```

[在 CodePen 上尝试](#)

接下来，我们会设置 `Clock` 的计时器并每秒更新它。

将生命周期方法添加到 Class 中

在具有许多组件的应用程序中，当组件被销毁时释放所占用的资源是非常重要的。

当 `Clock` 组件第一次被渲染到 DOM 中的时候，就为其[设置一个计时器](#)。这在 React 中被称为“挂载（mount）”。

同时，当 DOM 中 `Clock` 组件被删除的时候，应该[清除计时器](#)。这在 React 中被称为“卸载（unmount）”。

我们可以为 class 组件声明一些特殊的方法，当组件挂载或卸载时就会去执行这些方法：

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  componentDidMount() {}

  componentWillUnmount() {}

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

这些方法叫做“生命周期方法”。

`componentDidMount()` 方法会在组件已经被渲染到 DOM 中后运行，所以，最好在这里设置计时器：

```
componentDidMount() {
  this.timerID = setInterval(
    () => this.tick(),
    1000
  );
}
```

接下来把计时器的 ID 保存在 `this` 之中 (`this.timerID`)。

尽管 `this.props` 和 `this.state` 是 React 本身设置的，且都拥有特殊的含义，但是其实你可以向 class 中随意添加不参与数据流（比如计时器 ID）的额外字段。

我们会在 `componentWillUnmount()` 生命周期方法中清除计时器：

```
componentWillUnmount() {
  clearInterval(this.timerID);
}
```

最后，我们会实现一个叫 `tick()` 的方法，`Clock` 组件每秒都会调用它。

使用 `this.setState()` 来时刻更新组件 state：

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  componentDidMount() {
    this.timerID = setInterval(
      () => this.tick(),
      1000
    );
  }

  componentWillUnmount() {
    clearInterval(this.timerID);
  }

  tick() {
    this.setState({
      date: new Date()
    });
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Clock />);
```

[在 CodePen 上尝试](#)

现在时钟每秒都会刷新。

让我们来快速概括一下发生了什么和这些方法的调用顺序：

1. 当 `<Clock />` 被传给 `root.render()` 的时候，React 会调用 `Clock` 组件的构造函数。因为 `Clock` 需要显示当前的时间，所以它会用一个包含当前时间的对象来初始化 `this.state`。我们会在之后更新 state。
2. 之后 React 会调用组件的 `render()` 方法。这就是 React 确定该在页面上展示什么的方式。然后 React 更新 DOM 来匹配 `Clock` 渲染的输出。
3. 当 `Clock` 的输出被插入到 DOM 中后，React 就会调用 `ComponentDidMount()` 生命周期方法。在这个方法中，`Clock` 组件向浏览器请求设置一个计时器来每秒调用一次组件的 `tick()` 方法。

4. 浏览器每秒都会调用一次 `tick()` 方法。在这方法之中，`Clock` 组件会通过调用 `setState()` 来计划进行一次 UI 更新。得益于 `setState()` 的调用，React 能够知道 state 已经改变了，然后会重新调用 `render()` 方法来确定页面上该显示什么。这一次，`render()` 方法中的 `this.state.date` 就不一样了，如此一来就会渲染输出更新过的时间。React 也会相应的更新 DOM。

5. 一旦 `Clock` 组件从 DOM 中被移除，React 就会调用 `componentWillUnmount()` 生命周期方法，这样计时器就停止了。

正确地使用 State

关于 `setState()` 你应该了解三件事：

不要直接修改 State

例如，此代码不会重新渲染组件：

```
// Wrong
this.state.comment = 'Hello';
```

而是应该使用 `setState()`：

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

构造函数是唯一可以给 `this.state` 赋值的地方。

State 的更新可能是异步的

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。

例如，此代码可能会无法更新计数器：

```
// Wrong
this.setState({
  counter: this.state.counter + this.props.increment,
});
```

要解决这个问题，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：

```
// Correct
this.setState((state, props) => ({
  counter: state.counter + props.increment
}));
```

上面使用了[箭头函数](#)，不过使用普通的函数也同样可以：

```
// Correct
this.setState(function(state, props) {
  return {
```

```

        counter: state.counter + props.increment
    };
};

}

```

State 的更新会被合并

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

例如, 你的 state 包含几个独立的变量 :

```

constructor(props) {
  super(props);
  this.state = {
    posts: [],
    comments: []
  };
}

```

然后你可以分别调用 `setState()` 来单独地更新它们 :

```

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, 所以 `this.setState({comments})` 完整保留了 `this.state.posts`, 但是完全替换了 `this.state.comments`。

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 :

```
<FormattedDate date={this.state.date} />
```

`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 :

```

function FormattedDate(props) {
  return <h2>It is {props.date.toLocaleTimeString()}</h2>;
}

```

[在 CodePen 上尝试](#)

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。

为了证明每个组件都是真正独立的，我们可以创建一个渲染三个 `Clock` 的 `App` 组件：

```
function App() {
  return (
    <div>
      <Clock />
      <Clock />
      <Clock />
    </div>
  );
}
```

[在 CodePen 上尝试](#)

每个 `Clock` 组件都会单独设置它自己的计时器并且更新它。

在 React 应用中，组件是有状态组件还是无状态组件属于组件实现的细节，它可能会随着时间的推移而改变。你可以在有状态的组件中使用无状态的组件，反之亦然。

[Go to TOC](#)

静态类型检查

像 [Flow](#) 和 [TypeScript](#) 等这些静态类型检查器，可以在运行前识别某些类型的问题。他们还可以通过增加自动补全等功能来改善开发者的工作流程。出于这个原因，我们建议在大型代码库中使用 Flow 或 TypeScript 来代替 `PropTypes`。

Flow

[Flow](#) 是一个针对 JavaScript 代码的静态类型检测器。Flow 由 Facebook 开发，经常与 React 一起使用。Flow 通过特殊的类型语法为变量，函数，以及 React 组件提供注解，帮助你尽早地发现错误。你可以阅读 [introduction to Flow](#) 来了解它的基础知识。

完成以下步骤，便可开始使用 Flow：

- 将 Flow 添加到你的项目依赖中。
- 确保编译后的代码中去除了 Flow 语法。
- 添加类型注解并且运行 Flow 来检查它们。

下面我们会详细解释这些步骤。

在项目中添加 Flow

首先，在终端中进入到项目根目录下。然后你需要执行以下命令：

如果你使用 [Yarn](#)，执行：

```
yarn add --dev flow-bin
```

如果你使用 [npm](#)，执行：

```
npm install --save-dev flow-bin
```

这个命令将在你的项目中安装最新版的 Flow。

接下来，将 `flow` 添加到项目 `package.json` 的 `"scripts"` 部分，以便能够从终端命令行中使用它：

```
{
  // ...
  "scripts": {
    "flow": "flow",
    // ...
  },
  // ...
}
```

最后，执行以下命令之一：

如果你使用 [Yarn](#)，执行：

```
yarn run flow init
```

如果你使用 `npm`, 执行 :

```
npm run flow init
```

这条命令将生成你需要提交的 Flow 配置文件。

从编译后的代码中去除 Flow 语法

Flow 通过这种类型注释的特殊语法扩展了 JavaScript 语言。但是, 浏览器不能够解析这种语法, 所以我们需要确保它不会被编译到在浏览器执行的 JavaScript bundle 中。

具体方法取决于你使用的 JavaScript 编译工具。

Create React App

如果你的项目使用的是 [Create React App](#), 那么 Flow 注解默认会被去除, 所以在这一步你不需要做任何事情。

Babel

注意 :

这些说明不适用于使用 Create React App 的用户。虽然 Create React App 底层也使用了 Babel, 但它已经配置了去除 Flow。如果你没有使用 Create React App, 请执行此步骤。

如果你的项目手动配置了 Babel, 你需要为 Flow 安装一个特殊的 preset。

如果你使用 Yarn, 执行 :

```
yarn add --dev @babel/preset-flow
```

如果你使用 npm, 执行 :

```
npm install --save-dev @babel/preset-flow
```

接下来将 `flow` preset 添加到你的 [Babel 配置](#) 配置中。例如, 如果你通过 `.babelrc` 文件配置 Babel, 它可能会如下所示 :

```
{
  "presets": [
    "@babel/preset-flow",
    "react"
  ]
}
```

这将让你可以在代码中使用 Flow 语法。

注意 :

Flow 不需要 react preset, 但他们经常一起使用。Flow 内置了 JSX 的语法识别。

其他构建工具设置

如果没有使用 Create React App 或 Babel 来构建项目, 可以通过 [flow-remove-types](#) 去除类型注解。

运行 Flow

如果你按照上面的说明操作, 你应该能运行 Flow 了。

```
yarn flow
```

如果你使用 npm, 执行 :

```
npm run flow
```

你应该会看到如下消息 :

```
No errors!
★ Done in 0.17s.
```

添加 Flow 类型注释

默认情况下, Flow 仅检查包含此注释的文件 :

```
// @flow
```

通常, 它位于文件的顶部。试着将其添加到项目的某些文件中, 然后运行 `yarn flow` 或 `npm run flow` 来查看 Flow 是否已经发现了一些问题。

还可以通过[这个选项](#)开启所有文件 (包括没有注解的文件) 的强制检查。通过 Flow 来检查全部文件对于现有的项目来说, 可能导致大量修改, 但对于希望完全集成 Flow 的新项目来说开启这个选项比较合理。

现在一切就绪 ! 我们建议你查看以下资源来了解有关 Flow 的更多信息 :

- [Flow 文档 : 类型注解](#)
- [Flow 文档 : 编辑器](#)
- [Flow 文档 : React](#)
- [在 Flow 中进行 lint](#)

TypeScript

TypeScript 是一种由微软开发的编程语言。它是 JavaScript 的一个类型超集, 包含独立的编译器。作为一种类型语言, TypeScript 可以在构建时发现 bug 和错误, 这样程序运行时就可以避免此类错误。您可以通过[此文档](#) 了解更多有关在 React 中使用 TypeScript 的知识。

完成以下步骤, 便可开始使用 TypeScript :

- 将 TypeScript 添加到你的项目依赖中

- 配置 TypeScript 编译选项
- 使用正确的文件扩展名
- 为你使用的库添加定义

下面让我们详细地介绍一下这些步骤：

在 Create React App 中使用 TypeScript

Create React App 内置了对 TypeScript 的支持。

需要创建一个使用 TypeScript 的新项目，在终端运行：

```
npx create-react-app my-app --template typescript
```

如需将 TypeScript 添加到现有的 **Create React App** 项目中，[请参考此文档](#).

注意：

如果你使用的是 Create React App，可以跳过本节的其余部分。其余部分讲述了不使用 Create React App 脚手架，手动配置项目的用户。

添加 TypeScript 到现有项目中

这一切都始于在终端中执行的一个命令。

如果你使用 [Yarn](#)，执行：

```
yarn add --dev typescript
```

如果你使用 [npm](#)，执行：

```
npm install --save-dev typescript
```

恭喜！你已将最新版本的 TypeScript 安装到项目中。安装 TypeScript 后我们就可以使用 `tsc` 命令。在配置编译器之前，让我们将 `tsc` 添加到 `package.json` 中的 "scripts" 部分：

```
{
  // ...
  "scripts": {
    "build": "tsc",
    // ...
  },
  // ...
}
```

配置 TypeScript 编译器

没有配置项，编译器提供不了任何帮助。在 TypeScript 里，这些配置项都在一个名为 `tsconfig.json` 的特殊文件中定义。可以通过执行以下命令生成该文件：

如果你使用 [Yarn](#), 执行 :

```
yarn run tsc --init
```

如果你使用 [npm](#), 执行 :

```
npx tsc --init
```

`tsconfig.json` 文件中, 有许多配置项用于配置编译器。查看所有配置项的详细说明, [请参考此文档](#)。

我们来看一下 `rootDir` 和 `outDir` 这两个配置项。编译器将从项目中找到 TypeScript 文件并编译成相对应 JavaScript 文件。但我们不想混淆源文件和编译后的输出文件。

为了解决该问题, 我们将执行以下两个步骤 :

- 首先, 让我们重新整理下项目目录, 把所有的源代码放入 `src` 目录中。

```

└── package.json
└── src
    └── index.ts
└── tsconfig.json

```

- 其次, 我们将通过配置项告诉编译器源码和输出的位置。

```
// tsconfig.json

{
  "compilerOptions": {
    // ...
    "rootDir": "src",
    "outDir": "build"
    // ...
  },
}
```

很好! 现在, 当我们运行构建脚本时, 编译器会将生成的 javascript 输出到 `build` 文件夹。[TypeScript React Starter](#) 提供了一套默认的 `tsconfig.json` 帮助你快速上手。

通常情况下, 你不希望将编译后生成的 JavaScript 文件保留在版本控制内。因此, 应该把构建文件夹添加到 `.gitignore` 中。

文件扩展名

在 React 中, 你的组件文件大多数使用 `.js` 作为扩展名。在 TypeScript 中, 提供两种文件扩展名 :

`.ts` 是默认的文件扩展名, 而 `.tsx` 是一个用于包含 `JSX` 代码的特殊扩展名。

运行 TypeScript

如果你按照上面的说明操作, 现在应该能运行 TypeScript 了。

```
yarn build
```

如果你使用 npm，执行：

```
npm run build
```

如果你没有看到输出信息，这意味着它编译成功了。

类型定义

为了能够显示来自其他包的错误和提示，编译器依赖于声明文件。声明文件提供有关库的所有类型信息。这样，我们的项目就可以用上像 npm 这样的平台提供的三方 JavaScript 库。

获取一个库的声明文件有两种方式：

Bundled - 该库包含了自己的声明文件。这样很好，因为我们只需要安装这个库，就可以立即使用它了。要知道一个库是否包含类型，看库中是否有 `index.d.ts` 文件。有些库会在 `package.json` 文件的 `typings` 或 `types` 属性中指定类型文件。

DefinitelyTyped - DefinitelyTyped 是一个庞大的声明仓库，为没有声明文件的 JavaScript 库提供类型定义。这些类型定义通过众包的方式完成，并由微软和开源贡献者一起管理。例如，React 库并没有自己的声明文件。但我们可以从 DefinitelyTyped 获取它的声明文件。只要执行以下命令。

```
# yarn
yarn add --dev @types/react

# npm
npm i --save-dev @types/react
```

局部声明 有时，你要使用的包里没有声明文件，在 DefinitelyTyped 上也没有。在这种情况下，我们可以创建一个本地的定义文件。因此，在项目的根目录中创建一个 `declarations.d.ts` 文件。一个简单的声明可能是这样的：

```
declare module 'querystring' {
  export function stringify(val: object): string
  export function parse(val: string): object
}
```

你现在已做好编码准备了！我们建议你查看以下资源来了解有关 TypeScript 的更多知识：

- [TypeScript 文档：常用类型](#)
- [TypeScript 文档：JavaScript 迁移](#)
- [TypeScript 文档：React 与 Webpack](#)

ReScript

ReScript 是一门类型语言，可编译为 JavaScript。它的部分核心功能是为了保证 100% 的类型覆盖，完美支持 JSX 和 [独有的 React 绑定](#)，以允许开发者无缝集成到现有的 JS/TS 的 React 代码库中。

你可以在此处。了解更多关于如何将 ReScript 集成到现有的 JS / React 代码库的信息。

Kotlin

[Kotlin](#) 是由 JetBrains 开发的一门静态类型语言。其目标平台包括 JVM、Android、LLVM 和 [JavaScript](#)。

JetBrains 专门为 React 社区开发和维护了几个工具：[React bindings](#) 以及 [Create React Kotlin App](#)。后者可以通过 Kotlin 快速编写 React 应用程序，并且不需要构建配置。

其他语言

注意，还有其他静态类型语言可以编译成 JavaScript，也与 React 兼容。例如，和 [elmish-react](#) 一起使用的 [F#/Fable](#)。查看他们各自的网站以获取更多信息，并欢迎添加更多和与 React 结合的静态类型语言到这个页面！

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。

注意：

严格模式检查仅在开发模式下运行；它们不会影响生产构建。

你可以为应用程序的任何部分启用严格模式。例如：`embed:strict-mode/enabling-strict-mode.js`

在上述的示例中，不会对 `Header` 和 `Footer` 组件运行严格模式检查。但是，`ComponentOne` 和 `ComponentTwo` 以及它们的所有后代元素都将进行检查。

`StrictMode` 目前有助于：

- [识别不安全的生命周期](#)
- [关于使用过时字符串 ref API 的警告](#)
- [关于使用废弃的 findDOMNode 方法的警告](#)
- [检测意外的副作用](#)
- [检测过时的 context API](#)
- [确保可复用的状态](#)

未来的 React 版本将添加更多额外功能。

识别不安全的生命周期

正如[这篇博文](#)所述，某些过时的生命周期方法在异步 React 应用程序中使用是不安全的。但是，如果你的应用程序使用了第三方库，很难确保它们不使用这些生命周期方法。幸运的是，严格模式可以帮助解决这个问题！

当启用严格模式时，React 会列出使用了不安全生命周期方法的所有 class 组件，并打印一条包含这些组件信息的警告消息，如下所示：

```
▶ Warning: Unsafe lifecycle methods were found within a strict-mode tree:
  in div (created by ExampleApplication)
  in ExampleApplication

componentWillMount: Please update the following components to use componentDidMount instead: ThirdPartyComponent

Learn more about this warning here:
https://fb.me/react-strict-mode-warnings
```

此时解决项目中严格模式所识别出来的问题，会使得在未来的 React 版本中使用 concurrent 渲染变得更容易。

关于使用过时字符串 ref API 的警告

以前，React 提供了两种方法管理 refs 的方式：已过时的字符串 ref API 的形式及回调函数 API 的形式。尽管字符串 ref API 在两者中使用更方便，但是它有一些缺点，因此官方推荐采用[回调的方式](#)。

React 16.3 新增了第三种选择，它提供了使用字符串 ref 的便利性，并且不存在任何缺点：[embed:16-3-release-blog-post/create-ref-example.js](#)

由于对象 ref 主要是为了替换字符串 ref 而添加的，因此严格模式现在会警告使用字符串 ref。

注意：

除了新增加的 `createRef` API，回调 ref 依旧适用。

你无需替换组件中的回调 ref。它们更灵活，因此仍将作为高级功能保留。

[在此处了解有关 `createRef` API 的更多信息](#)

关于使用废弃的 `findDOMNode` 方法的警告

React 支持用 `findDOMNode` 来在给定 class 实例的情况下在树中搜索 DOM 节点。通常你不需要这样做，因为你可以[将 ref 直接绑定到 DOM 节点](#)。

`findDOMNode` 也可用于 class 组件，但它违反了抽象原则，它使得父组件需要单独渲染子组件。它会产生重构危险，你不能更改组件的实现细节，因为父组件可能正在访问它的 DOM 节点。`findDOMNode` 只返回第一个子节点，但是使用 Fragments，组件可以渲染多个 DOM 节点。`findDOMNode` 是一个只读一次的 API。调用该方法只会返回第一次查询的结果。如果子组件渲染了不同的节点，则无法跟踪此更改。因此，`findDOMNode` 仅在组件返回单个且不可变的 DOM 节点时才有效。

你可以通过将 ref 传递给自定义组件并使用 [ref 转发](#) 来将其传递给 DOM 节点。

你也可以在组件中创建一个 DOM 节点的 wrapper，并将 ref 直接绑定到它。

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.wrapper = React.createRef();
  }
  render() {
    return <div ref={this.wrapper}>{this.props.children}</div>;
  }
}
```

注意：

在 CSS 中，如果你不希望节点成为布局的一部分，则可以使用 `display: contents` 属性。

检测意外的副作用

从概念上讲，React 分两个阶段工作：

- **渲染** 阶段会确定需要进行哪些更改，比如 DOM。在此阶段，React 调用 `render`，然后将结果与上次渲染的结果进行比较。
- **提交** 阶段发生在当 React 应用变化时。（对于 React DOM 来说，会发生在 React 插入、更新及删除 DOM 节点的时候。）在此阶段，React 还会调用 `componentDidMount` 和 `componentDidUpdate` 之类的生命周期方法。

提交阶段通常会很快，但渲染过程可能很慢。因此，即将推出的 concurrent 模式（默认情况下未启用）将渲染工作分解为多个部分，对任务进行暂停和恢复操作以避免阻塞浏览器。这意味着 React 可以在提交之前多次调用渲染阶段生命周期的方法，或者在不提交的情况下调用它们（由于出现错误或更高优先级的任务使其中断）。

渲染阶段的生命周期包括以下 class 组件方法：

- `constructor`
- `componentWillMount` (or `UNSAFE_componentWillMount`)
- `componentWillReceiveProps` (or `UNSAFE_componentWillReceiveProps`)
- `componentWillUpdate` (or `UNSAFE_componentWillUpdate`)
- `getDerivedStateFromProps`
- `shouldComponentUpdate`
- `render`
- `setState` 更新函数（第一个参数）

因为上述方法可能会被多次调用，所以不要在它们内部编写副作用相关的代码，这点非常重要。忽略此规则可能会导致各种问题的产生，包括内存泄漏和或出现无效的应用程序状态。不幸的是，这些问题很难被发现，因为它们通常具有**非确定性**。

严格模式不能自动检测到你的副作用，但它可以帮助你发现它们，使它们更具确定性。通过故意重复调用以下函数来实现的该操作：

- class 组件的 `constructor`, `render` 以及 `shouldComponentUpdate` 方法
- class 组件的生命周期方法 `getDerivedStateFromProps`
- 函数组件体
- 状态更新函数（即 `setState` 的第一个参数）
- 函数组件通过使用 `useState`, `useMemo` 或者 `useReducer`

注意：

这仅适用于开发模式。生产模式下生命周期不会被调用两次。

例如，请考虑以下代码：`embed:strict-mode/side-effects-in-constructor.js`

这段代码看起来似乎没有问题。但是如果 `SharedApplicationState.recordEvent` 不是**幂等**的情况下，多次实例化此组件可能会导致应用程序状态无效。这种小 bug 可能在开发过程中不会表现出来，或者说表现出来但并不明显，并因此被忽视。

严格模式采用故意重复调用方法（如组件的构造函数）的方式，使得这种 bug 更容易被发现。

注意：

在 React 17 中，React 会自动修改 `console` 的方法，例如 `console.log()`，在第二次调用生命周期函数时，将日志静默。然而，在某些情况下，这可能会导致一些不符合期望的行为发生，此时，[可以使用替代解决方案](#)。

从 React 18 开始，React 不会抑制任何日志。不过，如果你安装了 React Dev Tools，第二次调用的日志会出现被轻微淡化。React DevTools 也提供了一个设置（默认关闭）来完全抑制它们。

检测过时的 context API

过时的 context API 容易出错，将在未来的主要版本中删除。在所有 16.x 版本中它仍然有效，但在严格模式下，将显示以下警告：

```
✖ ▶Warning: Legacy context API has been detected within a strict-mode tree:  
  in div (at App.js:32)  
  in App (at index.js:7)  
  
Please update the following components: LegacyContextConsumer, LegacyContextProvider  
  
Learn more about this warning here:  
https://fb.me/react-strict-mode-warnings
```

阅读[新的 context API 文档](#)以帮助你迁移到新版本。

确保可复用的 state

在未来，我们希望增加一个功能，允许 React 在保留 state 的同时对 UI 进行增删。例如，当用户从当前屏幕的标签离开并返回时，React 应该能立即展示之前屏幕的内容。为了做到这一点，React 将支持使用卸载前已有的组件状态重新挂载到树上。

该特性会给 React 带来更好的开箱即用性能，但需要组件对多次挂载和销毁的副作用具有弹性。大多数副作用将在不做任何改变的情况下工作，但有些副作用可能会在销毁回调中未正确的清理订阅，或者隐示的认为它们只被挂载或销毁一次。

为了帮助解决这些问题，React 18 为严格模式引入了一个全新的仅用于开发环境的检查操作。每当第一次安装组件时，这个新的检查将自动卸载并重新安装每个组件，并在第二次挂载时恢复之前的 state。

为了演示你在严格模式下看到的具有这一特性的开发行为，考虑一下当 React 挂载一个新组件时会发生什么？如果没有这个变化，当一个组件挂载时，React 会创建副作用：

- * React mounts the component.
- * Layout effects are created.
- * Effects are created.

从 React 18 开始的严格模式，每当组件在开发中挂载时，React 会模拟立即卸载和重新挂载组件：

- * React mounts the component.
- * Layout effects are created.
- * Effect effects are created.
- * React simulates effects being destroyed on a mounted component.
- * Layout effects are destroyed.
- * Effects are destroyed.

- * React simulates effects being re-created on a mounted component.
 - * Layout effects are created
 - * Effect setup code runs

在第二次挂载时，React 将恢复第一次装载时的状态。这个功能模拟了用户的行为，比如用户从屏幕上切换标签再回来，确保代码能正确处理状态恢复。

当组件卸载时，副作用会如常销毁：

- * React unmounts the component.
 - * Layout effects are destroyed.
 - * Effect effects are destroyed.

卸载和重新挂载的函数，包括：

- `componentDidMount`
- `componentWillUnmount`
- `useEffect`
- `useLayoutEffect`
- `useInsertionEffect`

注意：

这只适用于开发模式，生产环境没有变化。

如需了解更多常见问题，请参阅：

- [如何在 Effects 中支持可复用的 state](#)

测试环境

本章节介绍了可能会影响你测试环境的因素，并包含某些场景下的建议。

测试运行器

使用 [Jest](#), [mocha](#), [ava](#) 等测试运行器能像编写 JavaScript 一样编写测试套件，并将其作为开发过程的环节运行。此外，测试套件也将作为持续集成的环节运行。

- Jest 与 React 项目广泛兼容，支持诸如模拟 [模块](#)、[计时器](#) 和 [jsdom](#) 等特性。如果你使用 [Create React App](#), [Jest 已经能够开箱即用且包含许多实用的默认配置](#)。
- 像 mocha 这样的库在真实浏览器环境下运行良好，并且可以为明确需要它的测试提供帮助。
- 端对端测试用于测试跨多个页面的长流程，并且需要[不同的设置](#)。

模拟渲染表面

测试通常在无法访问真实渲染表面（如浏览器）的环境中运行。对于这些环境，我们建议使用 [jsdom](#) 来模拟浏览器，这是一个在 Node.js 内运行的轻量级浏览器实现。

在大多数情况下，jsdom 的行为类似于常规浏览器，但不具备如[布局和导航](#)的功能。这对于大多数基于 Web 的组件测试仍然有用，因为它的运行比为每个测试启动浏览器的方式效率更高。并且由于它与你编写的测试运行在同一个进程中，所以你能够编写代码来检查和断言渲染的 DOM。

就像在真实的浏览器中一样，jsdom 让我们模拟用户交互；测试可以在 DOM 节点上派发事件，然后观察并断言这些操作的副作用([例子](#))。

可以使用上述设置编写大部分 UI 测试：使用 Jest 作为测试运行器，渲染到 jsdom，使用 [act\(\)](#) 辅助函数([例子](#))提供的能力通过一系列的浏览器事件来模拟用户交互行为。例如，大量 React 自己的测试都是用这种组合编写的。

如果您正在编写一个主要测试浏览器特定行为的库，并且需要布局或真实输入等原生浏览器行为，那么你可以使用像 [mocha](#) 这样的框架。

在你 无法 模拟 DOM 环境（例如，在 Node.js 上测试 React Native 组件）的情况下，可以使用 [事件模拟辅助函数](#) 来模拟与元素的交互。或者，你也可以使用 [@testing-library/react-native](#) 中的 [fireEvent](#) 辅助函数。

诸如 [Cypress](#), [puppeteer](#) 和 [webdriver](#) 等框架对于运行[端对端测试](#) 都非常有用。

模拟功能

在编写测试的时候，我们希望模拟代码在测试环境较真实环境中缺失的等效部分（例如，在 Node.js 中检查 [navigator.onLine](#) 的状态）。测试还可以监视某些功能，并观察测试的其他部分如何与它们进行交互。有选择的将这些功能模拟为测试友好的版本是很有用的。

这对于数据获取尤其有用。通常最好使用“假”数据进行测试，以避免从实际 API 端获取数据可能导致的缓慢和不稳定 ([例子](#))。这样做有助于让测试变得可预测。像 [Jest](#) 与 [sinon](#) 这样的类库，支持模拟功能。对于端对端测试，虽然模拟网络可能更加困难，但你可能还想对真实的 API 端进行测试。

模拟模块

一些组件可能会依赖在测试环境中无法正常运行的模块，或者说这些模块对于我们的测试并不必要。那么，通过选择性地模拟来替换这些模块是很有用的（[例子](#)）。

在 Node.js 中，测试运行器如 Jest [支持模拟模块](#)。你也可以使用像 `mock-require` 这样的类库。

模拟计时器

组件可能会使用基于时间的函数如 `setTimeout`、`setInterval` 和 `Date.now` 等。在测试环境中，使用可以手动“推进”时间的替代物来模拟这些功能会很有帮助。它会确保你的测试快速运行！依赖于计时器的测试仍将按照顺序解析，但会更快（[例子](#)）。大部分测试框架，包括 Jest、sinon 和 Iolex 都允许你在测试中模拟计时器。

有些时候可能你不想要模拟计时器。例如，在你测试动画时，或是交互端对时间较为敏感的情况下（如 API 访问速率限制器）。具有计时器模拟的库允许你在每个测试/套件上启用或禁用这个功能，因此你可以明确地选择这些测试的运行方式。

端对端测试

端对端测试对于测试更长的工作流程非常有用，特别是当它们对于你的业务（例如付款或注册）特别重要时。对于这些测试，你可能会希望测试真实浏览器如何渲染整个应用、从真实的 API 端获取数据、使用 session 和 cookies 以及在不同的链接间导航等功能。你可能还希望不仅在 DOM 状态上进行断言，而同时也在后端数据上进行校验（例如，验证更新是否已经在数据库中持久化）。

在这种场景下，你可以使用像 Cypress、Playwright 等类似框架或者使用 Puppeteer 这样的库，这样你就可以在多个路由之间导航切换，并且不仅能够在浏览器中对副作用进行断言也能够在后端这么做。

测试技巧

React 组件的常见测试模式。

注意：

此章节假设你正在使用 [Jest](#) 作为测试运行器。如果你使用不同的测试运行器，你可能需要调整 API，但整体的解决方案是相同的。在[测试环境](#)章节阅读更多关于设置测试环境的细节。

在本章中，我们将主要使用函数组件。然而，这些测试策略并不依赖于实现细节，它对于 class 组件也同样有效。

- [创建/清理](#)
- [act\(\)](#)
- [渲染](#)
- [数据获取](#)
- [mock 模块](#)
- [事件](#)
- [计时器](#)
- [快照测试](#)
- [多渲染器](#)
- [缺少什么？](#)

创建/清理

对于每个测试，我们通常希望将 React 树渲染给附加到 `document` 的 DOM 元素。这点很重要，以便它可以接收 DOM 事件。当测试结束时，我们需要“清理”并从 `document` 中卸载树。

常见的方法是使用一对 `beforeEach` 和 `afterEach` 块，以便它们一直运行，并隔离测试本身造成的影响：

```
import { unmountComponentAtNode } from "react-dom";

let container = null;
beforeEach(() => {
  // 创建一个 DOM 元素作为渲染目标
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // 退出时进行清理
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});
```

你可以使用不同的测试模式，但请注意，即使测试失败，也需要执行清理。否则，测试可能会导致“泄漏”，并且一个测试可能会影响另一个测试的行为。这使得其难以调试。

act() {#act}

在编写 UI 测试时，可以将渲染、用户事件或数据获取等任务视为与用户界面交互的“单元”。`react-dom/test-utils` 提供了一个名为 `act()` 的 helper，它确保在进行任何断言之前，与这些“单元”相关的所有更新都已处理并应用于 DOM：

```
act(() => {
  // 渲染组件
});
// 进行断言
```

这有助于使测试运行更接近真实用户在使用应用程序时的体验。这些示例的其余部分使用 `act()` 来作出这些保证。

你可能会发现直接使用 `act()` 有点过于冗长。为了避免一些样板代码，你可以使用 [React 测试库](#)，这些 helper 是使用 `act()` 函数进行封装的。

注意：

`act` 名称来自 [Arrange-Act-Assert](#) 模式。

渲染

通常，你可能希望测试组件对于给定的 prop 渲染是否正确。此时应考虑实现基于 prop 渲染消息的简单组件：

```
// hello.js

import React from "react";

export default function Hello(props) {
  if (props.name) {
    return <h1>你好, {props.name}! </h1>;
  } else {
    return <span>嘿, 陌生人</span>;
  }
}
```

我们可以为这个组件编写测试：

```
// hello.test.js

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";

import Hello from "./hello";

let container = null;
beforeEach(() => {
  // 创建一个 DOM 元素作为渲染目标
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // 退出时进行清理
  unmountComponentAtNode(container);
  document.body.removeChild(container);
});
```

```

unmountComponentAtNode(container);
container.remove();
container = null;
});

it("渲染有或无名称", () => {
  act(() => {
    render(<Hello />, container);
  });
  expect(container.textContent).toBe("嘿, 陌生人");

  act(() => {
    render(<Hello name="Jenny" />, container);
  });
  expect(container.textContent).toBe("你好, Jenny! ");

  act(() => {
    render(<Hello name="Margaret" />, container);
  });
  expect(container.textContent).toBe("你好, Margaret! ");
});

```

数据获取

你可以使用假数据来 mock 请求，而不是在所有测试中调用真正的 API。使用“假”数据 mock 数据获取可以防止由于后端不可用而导致的测试不稳定，并使它们运行得更快。注意：你可能仍然希望使用一个“[端到端](#)”的框架来运行测试子集，该框架可显示整个应用程序是否一起工作。

```

// user.js

import React, { useState, useEffect } from "react";

export default function User(props) {
  const [user, setUser] = useState(null);

  async function fetchUserData(id) {
    const response = await fetch(`/${id}`);
    setUser(await response.json());
  }

  useEffect(() => {
    fetchUserData(props.id);
  }, [props.id]);

  if (!user) {
    return "加载中...";
  }

  return (
    <details>
      <summary>{user.name}</summary>
      <strong>{user.age}</strong> 岁
      <br />
      住在 {user.address}
    </details>
  );
}

```

我们可以为它编写测试：

```
// user.test.js

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";
import User from "./user";

let container = null;
beforeEach(() => {
  // 创建一个 DOM 元素作为渲染目标
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // 退出时进行清理
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});

it("渲染用户数据", async () => {
  const fakeUser = {
    name: "Joni Baez",
    age: "32",
    address: "123, Charming Avenue"
  };

  jest.spyOn(global, "fetch").mockImplementation(() =>
    Promise.resolve({
      json: () => Promise.resolve(fakeUser)
    })
  );

  // 使用异步的 act 应用执行成功的 promise
  await act(async () => {
    render(<User id="123" />, container);
  });

  expect(container.querySelector("summary").textContent).toBe(fakeUser.name);
  expect(container.querySelector("strong").textContent).toBe(fakeUser.age);
  expect(container.textContent).toContain(fakeUser.address);

  // 清理 mock 以确保测试完全隔离
  global.fetch.mockRestore();
});
```

mock 模块

有些模块可能在测试环境中不能很好地工作，或者对测试本身不是很重要。使用虚拟数据来 mock 这些模块可以使你为代码编写测试变得更容易。

考虑一个嵌入第三方 `GoogleMap` 组件的 `Contact` 组件：

```
// map.js

import React from "react";
import { LoadScript, GoogleMap } from "react-google-maps";
export default function Map(props) {
```

```

    return (
      <LoadScript id="script-loader" googleMapsApiKey="YOUR_API_KEY">
        <GoogleMap id="example-map" center={props.center} />
      </LoadScript>
    );
}

// contact.js

import React from "react";
import Map from "./map";

export default function Contact(props) {
  return (
    <div>
      <address>
        联系 {props.name}, 通过{" "}
        <a data-testid="email" href={"mailto:" + props.email}>
          email
        </a>
        或者他们的 <a data-testid="site" href={props.site}>
          网站
        </a>。
      </address>
      <Map center={props.center} />
    </div>
  );
}

```

如果不想在测试中加载这个组件，我们可以将依赖 mock 到一个虚拟组件，然后运行我们的测试：

```

// contact.test.js

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";

import Contact from "./contact";
import MockedMap from "./map";

jest.mock("./map", () => {
  return function DummyMap(props) {
    return (
      <div data-testid="map">
        {props.center.lat}:{props.center.long}
      </div>
    );
  };
});

let container = null;
beforeEach(() => {
  // 创建一个 DOM 元素作为渲染目标
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // 退出时进行清理
  unmountComponentAtNode(container);
  container.remove();
});

```

```

    container = null;
});

it("应渲染联系信息", () => {
  const center = { lat: 0, long: 0 };
  act(() => {
    render(
      <Contact
        name="Joni Baez"
        email="test@example.com"
        site="http://test.com"
        center={center}
      />,
      container
    );
  });

  expect(
    container.querySelector("[data-testid='email']").getAttribute("href")
  ).toEqual("mailto:test@example.com");

  expect(
    container.querySelector('[data-testid="site"]').getAttribute("href")
  ).toEqual("http://test.com");

  expect(container.querySelector('[data-testid="map"]').textContent).toEqual(
    "0:0"
);
});

```

Events

我们建议在 DOM 元素上触发真正的 DOM 事件，然后对结果进行断言。考虑一个 `Toggle` 组件：

```

// toggle.js

import React, { useState } from "react";

export default function Toggle(props) {
  const [state, setState] = useState(false);
  return (
    <button
      onClick={() => {
        setState(previousState => !previousState);
        props.onChange(!state);
      }}
      data-testid="toggle"
    >
      {state === true ? "Turn off" : "Turn on"}
    </button>
  );
}

```

我们可以为它编写测试：

```

// toggle.test.js

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";

```

```

import Toggle from "./toggle";

let container = null;
beforeEach(() => {
  // 创建一个 DOM 元素作为渲染目标
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // 退出时进行清理
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});

it("点击时更新值", () => {
  const onChange = jest.fn();
  act(() => {
    render(<Toggle onChange={onChange} />, container);
  });

  // 获取按钮元素，并触发点击事件
  const button = document.querySelector("[data-testid=toggle]");
  expect(button.innerHTML).toBe("Turn on");

  act(() => {
    button.dispatchEvent(new MouseEvent("click", { bubbles: true }));
  });

  expect(onChange).toHaveBeenCalledTimes(1);
  expect(button.innerHTML).toBe("Turn off");

  act(() => {
    for (let i = 0; i < 5; i++) {
      button.dispatchEvent(new MouseEvent("click", { bubbles: true }));
    }
  });

  expect(onChange).toHaveBeenCalledTimes(6);
  expect(button.innerHTML).toBe("Turn on");
});

```

[MDN](#)描述了不同的 DOM 事件及其属性。注意，你需要在创建的每个事件中传递 `{ bubbles: true }` 才能到达 React 监听器，因为 React 会自动将事件委托给 root。

注意：

React 测试库为触发事件提供了一个[更简洁 helper](#)。

计时器

你的代码可能会使用基于计时器的函数（如 `setTimeout`）来安排将来更多的工作。在这个例子中，多项选择面板等待选择并前进，如果在 5 秒内没有做出选择，则超时：

```
// card.js

import React, { useEffect } from "react";

export default function Card(props) {
  useEffect(() => {
    const timeoutID = setTimeout(() => {
      props.onSelect(null);
    }, 5000);
    return () => {
      clearTimeout(timeoutID);
    };
  }, [props.onSelect]);

  return [1, 2, 3, 4].map(choice => (
    <button
      key={choice}
      data-testid={choice}
      onClick={() => props.onSelect(choice)}
    >
      {choice}
    </button>
  ));
}
```

我们可以利用 [Jest 的计时器 mock](#) 为这个组件编写测试，并测试它可能处于的不同状态。

```
// card.test.js

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";

import Card from "./card";

let container = null;
beforeEach(() => {
  // 创建一个 DOM 元素作为渲染目标
  container = document.createElement("div");
  document.body.appendChild(container);
  jest.useFakeTimers();
});

afterEach(() => {
  // 退出时进行清理
  unmountComponentAtNode(container);
  container.remove();
  container = null;
  jest.useRealTimers();
});

it("超时后应选择 null", () => {
  const onSelect = jest.fn();
  act(() => {
    render(<Card onSelect={onSelect} />, container);
  });

  // 提前 100 毫秒执行
  act(() => {
    jest.advanceTimersByTime(100);
  });
});
```

```

expect(onSelect).not.toHaveBeenCalled();

// 然后提前 5 秒执行
act(() => {
  jest.advanceTimersByTime(5000);
});
expect(onSelect).toHaveBeenCalledWith(null);
});

it("移除时应进行清理", () => {
  const onSelect = jest.fn();
  act(() => {
    render(<Card onSelect={onSelect} />, container);
  });

  act(() => {
    jest.advanceTimersByTime(100);
  });
  expect(onSelect).not.toHaveBeenCalled();

  // 卸载应用程序
  act(() => {
    render(null, container);
  });

  act(() => {
    jest.advanceTimersByTime(5000);
  });
  expect(onSelect).not.toHaveBeenCalled();
});

it("应接受选择", () => {
  const onSelect = jest.fn();
  act(() => {
    render(<Card onSelect={onSelect} />, container);
  });

  act(() => {
    container
      .querySelector("[data-testid='2']")
      .dispatchEvent(new MouseEvent("click", { bubbles: true }));
  });

  expect(onSelect).toHaveBeenCalledWith(2);
});

```

你只能在某些测试中使用假计时器。在上面，我们通过调用 `jest.useFakeTimers()` 来启用它们。它们提供的主要优势是，你的测试实际上不需要等待 5 秒来执行，而且你也不需要为了测试而使组件代码更加复杂。

快照测试

像 Jest 这样的框架还允许你使用 `toMatchSnapshot` / `toMatchInlineSnapshot` 保存数据的“快照”。有了这些，我们可以“保存”渲染的组件输出，并确保对它的更新作为对快照的更新显式提交。

在这个示例中，我们渲染一个组件并使用 `pretty` 包对渲染的 HTML 进行格式化，然后将其保存为内联快照：

```

// hello.test.js, again
import React from "react";

```

```

import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";
import pretty from "pretty";

import Hello from "./hello";

let container = null;
beforeEach(() => {
  // 创建一个 DOM 元素作为渲染目标
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // 退出时进行清理
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});

it("应渲染问候语", () => {
  act(() => {
    render(<Hello />, container);
  });

  expect(
    pretty(container.innerHTML)
  ).toMatchSnapshot(); /* ... 由 jest 自动填充 ... */

  act(() => {
    render(<Hello name="Jenny" />, container);
  });

  expect(
    pretty(container.innerHTML)
  ).toMatchSnapshot(); /* ... 由 jest 自动填充 ... */

  act(() => {
    render(<Hello name="Margaret" />, container);
  });

  expect(
    pretty(container.innerHTML)
  ).toMatchSnapshot(); /* ... 由 jest 自动填充 ... */
});

```

通常，进行具体的断言比使用快照更好。这类测试包括实现细节，因此很容易中断，并且团队可能对快照中断不敏感。选择性地 [mock 一些子组件](#)可以帮助减小快照的大小，并使它们在代码评审中保持可读性。

多渲染器

在极少数情况下，你可能正在使用多个渲染器的组件上运行测试。例如，你可能正在使用 `react-test-renderer` 组件上运行快照测试，该组件内部使用 `react-dom` 的 `render` 来渲染子组件的一些内容。在这个场景中，你可以使用与它们的渲染器相对应的 `act()` 来包装更新。

```

import { act as domAct } from "react-dom/test-utils";
import { act as testAct, create } from "react-test-renderer";
// ...
let root;

```

```
domAct(() => {
  testAct(() => {
    root = create(<App />);
  });
})];
expect(root).toMatchSnapshot();
```

缺少什么？

如果有一些常见场景没有覆盖，请在文档网站的 [issue 跟踪器](#)上告诉我们。

测试概览

- "community/testing.html"

你可以用像测试其他 JavaScript 代码类似的方式测试 React 组件。

现在有许多种测试 React 组件的方法。大体上可以被分为两类：

- **渲染组件树** 在一个简化的测试环境中渲染组件树并对它们的输出做断言检查。
- **运行完整应用** 在一个真实的浏览器环境中运行整个应用（也被称为“端到端（end-to-end）”测试）。

本章节主要专注于第一种情况下的测试策略。虽然完整的端到端测试在防止对重要工作流的多次回归方面很有价值，但对 React 组件来说这类测试并不怎么需要关注，因此不在本章节的讨论范围之内。

权衡

当挑选测试工具的时候，有些细节值得我们权衡考虑：

- **迭代速度 vs 真实环境**：一些工具在做出改动和看到结果之间提供了非常快速的反馈循环，但没有精确的模拟浏览器的行为。另一些工具，也许使用了真实的浏览器环境，但却降低了迭代速度，而且在持续集成服务器中不太可靠。
- **mock 到什么程度**：对组件来说，“单元测试”和“集成测试”之间的差别可能会很模糊。如果你在测试一个表单，用例是否应该也测试表单里的按钮呢？一个按钮组件又需不需要有他自己的测试套件？重构按钮组件是否应该影响表单的测试用例？

不同的团队或产品可能会得出不同的答案。

推荐的工具

Jest 是一个 JavaScript 测试运行器。它允许你使用 `jsdom` 操作 DOM。尽管 `jsdom` 只是对浏览器工作表现的一个近似模拟，对测试 React 组件来说它通常也已经够用了。Jest 有着十分优秀的迭代速度，同时还提供了若干强大的功能，比如它可以模拟 `modules` 和 `timers` 让你更精细的控制代码如何执行。

React 测试库是一组能让你不依赖 React 组件具体实现对他们进行测试的辅助工具。它让重构工作变得轻而易举，还会推动你拥抱有关无障碍的最佳实践。虽然它不能让你省略子元素来浅（shallowly）渲染一个组件，但像 Jest 这样的测试运行器可以通过 `mocking` 让你做到。

了解更多

这一章节被划分成了两页内容：

- **技巧**：为 React 组件编写测试时的常见模式。
- **环境**：为 React 组件搭建测试环境的时候有哪些要考虑的东西。

React 哲学

- ['blog/2013/11/05/thinking-in-react.html'](#)
- ['docs/thinking-in-react-zh-CN.html'](#)

我们认为，React 是用 JavaScript 构建快速响应的大型 Web 应用程序的首选方式。它在 Facebook 和 Instagram 上表现优秀。

React 最棒的部分之一是引导我们思考如何构建一个应用。在这篇文档中，我们将会通过 React 构建一个可搜索的产品数据表格来更深刻地领会 React 哲学。

从设计稿开始

假设我们已经有了一个返回 JSON 的 API，以及设计师提供的组件设计稿。如下所示：

Name	Price
Sporting Goods	
Football	\$49.99
Baseball	\$9.99
Basketball	\$29.99
Electronics	
iPod Touch	\$99.99
iPhone 5	\$399.99
Nexus 7	\$199.99

Mockup

该 JSON API 会返回以下数据：

```
[ ];
```

第一步：将设计好的 UI 划分为组件层级

首先，你需要在设计稿上用方框圈出每一个组件（包括它们的子组件），并且以合适的名称命名。如果你是和设计师一起完成此任务，那么他们可能已经做过类似的工作，所以请和他们进行交流！他们的 Photoshop 的图层名称可能最终就是你编写的 React 组件的名称！

但你如何确定应该将哪些部分划分到一个组件中呢？你可以将组件当作一种函数或者是对象来考虑，根据[单一功能原则](#)来判定组件的范围。也就是说，一个组件原则上只能负责一个功能。如果它需要负责更多的功能，这时候就应该考虑将它拆分成更小的组件。

在实践中，因为你经常是在向用户展示 JSON 数据模型，所以如果你的模型设计得恰当，UI（或者说组件结构）便会与数据模型一一对应，这是因为 UI 和数据模型都会倾向于遵守相同的信息结构。将 UI 分离为组件，其中每个组件需与数据模型的某部分匹配。



组件嵌套图示

你会看到我们的应用中包含五个组件。我们已经将每个组件展示的数据标注为了斜体。图片中的序号与下方列表中的序号对应。

1. `FilterableProductTable` (橙色): 是整个示例应用的整体
2. `SearchBar` (蓝色): 接受所有的用户输入
3. `ProductTable` (绿色): 展示数据内容并根据用户输入筛选结果
4. `ProductCategoryRow` (天蓝色): 为每一个产品类别展示标题
5. `ProductRow` (红色): 每一行展示一个产品

你可能注意到，`ProductTable` 的表头（包含 "Name" 和 "Price" 的那一部分）并未单独成为一个组件。这仅仅是一种偏好选择，如何处理这一问题也一直存在争论。就这个示例而言，因为表头只起到了渲染数据集合的作用——这与 `ProductTable` 是一致的，所以我们仍然将其保留为 `ProductTable` 的一部分。但是，如果表头过于复杂（例如，我们需为其添加排序功能），那么将它作为一个独立的 `ProductTableHeader` 组件就显得很有必要了。

现在我们已经确定了设计稿中应该包含的组件，接下来我们将把它们描述为更加清晰的层级。设计稿中被其他组件包含的子组件，在层级上应该作为其子节点。

- `FilterableProductTable`
 - `SearchBar`
 - `ProductTable`
 - `ProductCategoryRow`
 - `ProductRow`

在构建应用的静态版本时，我们需要创建一些会重用其他组件的组件，然后通过 `props` 传入所需的数据。`props` 是父组件向子组件传递数据的方式。即使你已经熟悉了 `state` 的概念，也完全不应该使用 **state** 构建静态版本。`state` 代表了随时间会产生变化的数据，应当仅在实现交互时使用。所以构建应用的静态版本时，你不会用到它。

你可以自上而下或者自下而上构建应用：自上而下意味着首先编写层级较高的组件（比如 `FilterableProductTable`），自下而上意味着从最基本的组件开始编写（比如 `ProductRow`）。当你的应用比较简单时，使用自上而下的方式更方便；对于较为大型的项目来说，自下而上地构建，并同时为低层组件编写测试是更加简单的方式。

到此为止，你应该已经有了一个可重用的组件库来渲染你的数据模型。由于我们构建的是静态版本，所以这些组件目前只需提供 `render()` 方法用于渲染。最顶层的组件 `FilterableProductTable` 通过 `props` 接受你的数据模型。如果你的数据模型发生了改变，再次调用 `root.render()`，UI 就会相应地被更新。数据模型变化、调用 `render()` 方法、UI 相应变化，这个过程并不复杂，因此很容易看清楚 UI 是如何被更新的，以及是在哪里被更新的。React **单向数据流**（也叫单向绑定）的思想使得组件模块化，易于快速开发。

如果你在完成这一步骤时遇到了困难，可以参阅 [React 文档](#)。

补充说明：有关 `props` 和 `state`

在 React 中，有两类“模型”数据：`props` 和 `state`。清楚地理解两者的区别是十分重要的；如果你不太有把握，可以参阅 [React 官方文档](#)。你也可以查看 [FAQ: state 与 props 的区别是什么？](#)

第三步：确定 UI `state` 的最小(且完整)表示

想要使你的 UI 具备交互功能，需要有触发基础数据模型改变的能力。React 通过实现 **state** 来完成这个任务。

为了正确地构建应用，你首先需要找出应用所需的 `state` 的最小表示，并根据需要计算出其他所有数据。其中的关键正是 **DRY: Don't Repeat Yourself**。只保留应用所需的可变 `state` 的最小集合，其他数据均由它们计算产生。比如，你要编写一个任务清单应用，你只需要保存一个包含所有事项的数组，而无需额外保存一个单独的 `state` 变量（用于存储任务个数）。当你需要展示任务个数时，只需要利用该数组的 `length` 属性即可。

我们的示例应用拥有如下数据：

- 包含所有产品的原始列表
- 用户输入的搜索词
- 复选框是否选中的值
- 经过搜索筛选的产品列表

通过问自己以下三个问题，你可以逐个检查相应数据是否属于 `state`：

1. 该数据是否是由父组件通过 `props` 传递而来的？如果是，那它应该不是 `state`。
2. 该数据是否随时间的推移而保持不变？如果是，那它应该也不是 `state`。
3. 你能否根据其他 `state` 或 `props` 计算出该数据的值？如果是，那它也不是 `state`。

包含所有产品的原始列表是经由 `props` 传入的，所以它不是 `state`；搜索词和复选框的值应该是 `state`，因为它们随时间会发生改变且无法由其他数据计算而来；经过搜索筛选的产品列表不是 `state`，因为它的结果可以由产品的原始列表根据搜索词和复选框的选择计算出来。

综上所述，属于 `state` 的有：

- 用户输入的搜索词
- 复选框是否选中的值

第四步：确定 state 放置的位置

我们已经确定了应用所需的 state 的最小集合。接下来，我们需要确定哪个组件能够改变这些 state，或者说拥有这些 state。

注意：React 中的数据流是单向的，并顺着组件层级从上往下传递。哪个组件应该拥有某个 state 这件事，**对初学者来说往往是最难理解的部分**。尽管这可能在一开始不是那么清晰，但你可以尝试通过以下步骤来判断：

对于应用中的每一个 state：

- 找到根据这个 state 进行渲染的所有组件。
- 找到他们的共同所有者 (common owner) 组件（在组件层级上高于所有需要该 state 的组件）。
- 该共同所有者组件或者比它层级更高的组件应该拥有该 state。
- 如果你找不到一个合适的位置来存放该 state，就可以直接创建一个新的组件来存放该 state，并将这一新组件置于高于共同所有者组件层级的位置。

根据以上策略重新考虑我们的示例应用：

- `ProductTable` 需要根据 state 筛选产品列表。`SearchBar` 需要展示搜索词和复选框的状态。
- 他们的共同所有者是 `FilterableProductTable`。

让我们重新梳理一下需要实现的功能：每当用户改变表单的值，我们需要改变 state 来反映用户的当前输入。由于 state 只能由拥有它们的组件进行更改，`FilterableProductTable` 必须将一个能够触发 state 改变的回调函数（callback）传递给 `SearchBar`。我们可以使用输入框的 `onChange` 事件来监视用户输入的变化，并通知 `FilterableProductTable` 传递给 `SearchBar` 的回调函数。然后该回调函数将调用 `setState()`，从而更新应用。

这就是全部了

希望这篇文档能够帮助你建立起构建 React 组件和应用的一般概念。尽管你可能需要编写更多的代码，但是别忘了：比起写，代码更多地是给人看的。我们一起构建的这个模块化示例应用的代码就很易于阅读。当你开始构建更大的组件库时，你会意识到这种代码模块化和清晰度的重要性。并且随着代码重用程度的加深，你的代码行数也会显著地减少。:)

使用 PropTypes 进行类型检查

注意：

自 React v15.5 起，`React.PropTypes` 已移入另一个包中。请使用 `prop-types` 库代替。

我们提供了一个 [codemod 脚本](#) 来做自动转换。

随着你的应用程序不断增长，你可以通过类型检查捕获大量错误。对于某些应用程序来说，你可以使用 [Flow](#) 或 [TypeScript](#) 等 JavaScript 扩展来对整个应用程序做类型检查。但即使你不使用这些扩展，React 也内置了一些类型检查的功能。要在组件的 `props` 上进行类型检查，你只需配置特定的 `propTypes` 属性：

```
import PropTypes from 'prop-types';

class Greeting extends React.Component {
  render() {
    return (
      <h1>Hello, {this.props.name}</h1>
    );
  }
}

Greeting.propTypes = {
  name: PropTypes.string
};
```

在此示例中，我们使用的是 `class` 组件，但是同样的功能也可用于函数组件，或者是由 `React.memo` / `React.forwardRef` 创建的组件。

`PropTypes` 提供一系列验证器，可用于确保组件接收到的数据类型是有效的。在本例中，我们使用了 `PropTypes.string`。当传入的 `prop` 值类型不正确时，JavaScript 控制台将会显示警告。出于性能方面的考虑，`propTypes` 仅在开发模式下进行检查。

PropTypes

以下提供了使用不同验证器的例子：

```
import PropTypes from 'prop-types';

MyComponent.propTypes = {
  // 你可以将属性声明为 JS 原生类型，默认情况下
  // 这些属性都是可选的。
  optionalArray: PropTypes.array,
  optionalBool: PropTypes.bool,
  optionalFunc: PropTypes.func,
  optionalNumber: PropTypes.number,
  optionalObject: PropTypes.object,
  optionalString: PropTypes.string,
  optionalSymbol: PropTypes.symbol,
  // 任何可被渲染的元素（包括数字、字符串、元素或数组）
};
```

```

// (或 Fragment) 也包含这些类型。
optionalNode: PropTypes.node,

// 一个 React 元素。
optionalElement: PropTypes.element,

// 一个 React 元素类型 (即, MyComponent)。
optionalElementType: PropTypes.elementType,

// 你也可以声明 prop 为类的实例，这里使用
// JS 的 instanceof 操作符。
optionalMessage: PropTypes.instanceOf(Message),

// 你可以让你的 prop 只能是特定的值，指定它为
// 枚举类型。
optionalEnum: PropTypes.oneOf(['News', 'Photos']),

// 一个对象可以是几种类型中的任意一个类型
optionalUnion: PropTypes.oneOfType([
  PropTypes.string,
  PropTypes.number,
  PropTypes.instanceOf(Message)
]),

// 可以指定一个数组由某一类型的元素组成
optionalArrayOf: PropTypes.arrayOf(PropTypes.number),

// 可以指定一个对象由某一类型的值组成
optionalObjectOf: PropTypes.objectOf(PropTypes.number),

// 可以指定一个对象由特定的类型值组成
optionalObjectWithShape: PropTypes.shape({
  color: PropTypes.string,
  fontSize: PropTypes.number
}),

// An object with warnings on extra properties
optionalObjectWithStrictShape: PropTypes.exact({
  name: PropTypes.string,
  quantity: PropTypes.number
}),

// 你可以在任何 PropTypes 属性后面加上 `isRequired`，确保
// 这个 prop 没有被提供时，会打印警告信息。
requiredFunc: PropTypes.func.isRequired,

// 任意类型的必需数据
requiredAny: PropTypes.any.isRequired,

// 你可以指定一个自定义验证器。它在验证失败时应返回一个 Error 对象。
// 请不要使用 `console.warn` 或抛出异常，因为这在 `oneOfType` 中不会起作用。
customProp: function(props, propName, componentName) {
  if (!/matchme/.test(props[propName])) {
    return new Error(
      'Invalid prop `' + propName + '` supplied to' +
      ' `' + componentName + '`. Validation failed.'
    );
  }
}

// 你也可以提供一个自定义的 `arrayOf` 或 `objectOf` 验证器。
// 它应该在验证失败时返回一个 Error 对象。

```

```
// 验证器将验证数组或对象中的每个值。验证器的前两个参数
// 第一个是数组或对象本身
// 第二个是他们当前的键。
customArrayProp: PropTypes.arrayOf(function(propValue, key, componentName,
location, propFullName) {
  if (!/matchme/.test(propValue[key])) {
    return new Error(
      'Invalid prop `' + propFullName + '` supplied to' +
      ' `' + componentName + '`. Validation failed.'
    );
  }
});
```

限制单个元素

你可以通过 `PropTypes.element` 来确保传递给组件的 `children` 中只包含一个元素。

```
import PropTypes from 'prop-types';

class MyComponent extends React.Component {
  render() {
    // 这必须只有一个元素，否则控制台会打印警告。
    const children = this.props.children;
    return (
      <div>
        {children}
      </div>
    );
  }
}

MyComponent.propTypes = {
  children: PropTypes.element.isRequired
};
```

默认 Prop 值

你可以通过配置特定的 `defaultProps` 属性来定义 `props` 的默认值：

```
class Greeting extends React.Component {
  render() {
    return (
      <h1>Hello, {this.props.name}</h1>
    );
  }
}

// 指定 props 的默认值:
Greeting.defaultProps = {
  name: 'Stranger'
};

// 渲染出 "Hello, Stranger":
const root = ReactDOM.createRoot(document.getElementById('example'));
root.render(<Greeting />);
```

从 ES2022 开始，你也可以在 React 类组件中将 `defaultProps` 声明为静态属性。欲了解更多信息，请参阅 [class public static fields](#)。这种现代语法需要添加额外的编译步骤才能在老版浏览器中工作。

```
class Greeting extends React.Component {
  static defaultProps = {
    name: 'stranger'
  }

  render() {
    return (
      <div>Hello, {this.props.name}</div>
    )
  }
}
```

`defaultProps` 用于确保 `this.props.name` 在父组件没有指定其值时，有一个默认值。`propTypes` 类型检查发生在 `defaultProps` 赋值后，所以类型检查也适用于 `defaultProps`。

函数组件

如果你在常规开发中使用函数组件，那你可能需要做一些适当的改动，以保证 `PropTypes` 应用正常。

假设你有如下组件：

```
export default function HelloWorldComponent({ name }) {
  return (
    <div>Hello, {name}</div>
  )
}
```

如果要添加 `PropTypes`，你可能需要在导出之前以单独声明的一个函数的形式，声明该组件，具体代码如下：

```
function HelloWorldComponent({ name }) {
  return (
    <div>Hello, {name}</div>
  )
}

export default HelloWorldComponent
```

接着，可以直接在 `HelloWorldComponent` 上添加 `PropTypes`：

```
import PropTypes from 'prop-types'

function HelloWorldComponent({ name }) {
  return (
    <div>Hello, {name}</div>
  )
}

HelloWorldComponent.propTypes = {
  name: PropTypes.string
}

export default HelloWorldComponent
```

react

[Go to TOC](#)

非受控组件

在大多数情况下，我们推荐使用 [受控组件](#) 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](#) 来从 DOM 节点中获取表单数据。

例如，下面的代码使用非受控组件接受一个表单的值：

```
class NameForm extends React.Component {
  constructor(props) {
    super(props);
    this.handleSubmit = this.handleSubmit.bind(this);
    this.input = React.createRef();
  }

  handleSubmit(event) {
    alert('A name was submitted: ' + this.input.current.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name:
          <input type="text" ref={this.input} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

[在 CodePen 上尝试](#)

因为非受控组件将真实数据储存在 DOM 节点中，所以在使用非受控组件时，有时候反而更容易同时集成 React 和非 React 代码。如果你不介意代码美观性，并且希望快速编写代码，使用非受控组件往往可以减少你的代码量。否则，你应该使用受控组件。

如果你还是不清楚在某个特殊场景中应该使用哪种组件，那么 [这篇关于受控和非受控输入组件的文章](#) 会很有帮助。

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。

```
render() {
  return (
    <form onSubmit={this.handleSubmit}>
      <label>
        Name:
        <input
```

```
        defaultValue="Bob"
        type="text"
        ref={this.input} />
    </label>
    <input type="submit" value="Submit" />
</form>
);
}
```

同样，`<input type="checkbox">` 和 `<input type="radio">` 支持 `defaultChecked`，`<select>` 和 `<textarea>` 支持 `defaultValue`。

文件输入

在 HTML 中，`<input type="file">` 可以让用户选择一个或多个文件上传到服务器，或者通过使用 [File API](#) 进行操作。

```
<input type="file" />
```

在 React 中，`<input type="file" />` 始终是一个非受控组件，因为它的值只能由用户设置，而不能通过代码控制。

您应该使用 File API 与文件进行交互。下面的例子显示了如何创建一个 DOM 节点的 `ref` 从而在提交表单时获取文件的信息。

```
embed:uncontrolled-components/input-type-file.js
```

[Go to TOC](#)

Web Components

React 和 [Web Components](#) 为了解决不同的问题而生。Web Components 为可复用组件提供了强大的封装，而 React 则提供了声明式的解决方案，使 DOM 与数据保持同步。两者旨在互补。作为开发人员，可以自由选择在 Web Components 中使用 React，或者在 React 中使用 Web Components，或者两者共存。

大多数开发者在使用 React 时，不使用 Web Components，但可能你会需要使用，尤其是在使用 Web Components 编写的第三方 UI 组件时。

在 React 中使用 Web Components

```
class HelloMessage extends React.Component {
  render() {
    return <div>Hello <x-search>{this.props.name}</x-search>!</div>;
  }
}
```

注意：

Web Components 通常暴露的是命令式 API。例如，Web Components 的组件 `video` 可能会公开 `play()` 和 `pause()` 方法。要访问 Web Components 的命令式 API，你需要使用 `ref` 直接与 DOM 节点进行交互。如果你使用的是第三方 Web Components，那么最好的解决方案是编写 React 组件包装该 Web Components。

Web Components 触发的事件可能无法通过 React 渲染树正确的传递。你需要在 React 组件中手动添加事件处理器来处理这些事件。

常见的误区是在 Web Components 中使用的是 `class` 而非 `className`。

```
function BrickFlipbox() {
  return (
    <brick-flipbox class="demo">
      <div>front</div>
      <div>back</div>
    </brick-flipbox>
  );
}
```

在 Web Components 中使用 React

```
class XSearch extends HTMLElement {
  connectedCallback() {
    const mountPoint = document.createElement('span');
    this.attachShadow({ mode: 'open' }).appendChild(mountPoint);

    const name = this.getAttribute('name');
    const url = 'https://www.google.com/search?q=' + encodeURIComponent(name);
    const root = ReactDOM.createRoot(mountPoint);
    root.render(<a href={url}>{name}</a>);
  }
}
```

```
    }
customElements.define('x-search', XSearch);
```

注意：

如果使用 Babel 来转换 class，此代码将**不会**起作用。请查阅该 [issue](#) 了解相关讨论。在加载 Web Components 前请引入 [custom-elements-es5-adapter](#) 来解决该 issue。

Colophon

This book is created by using the following sources:

- React - 中文
- GitHub source: reactjs/zh-hans.reactjs.org/content
- Created: 2022-11-27
- Bash v5.2.2
- Vivliostyle, <https://vivliostyle.org/>
- By: @shinokada
- GitHub repo: <https://github.com/shinokada/markdown-docs-as-pdf>