

第一章 核心

Hello World

如何阅读本指南

预备知识

JSX 简介

为什么使用 JSX?

在 JSX 中嵌入表达式

JSX 也是一个表达式

JSX 特定属性

使用 JSX 指定子元素

JSX 防止注入攻击

JSX 表示对象

元素渲染

将一个元素渲染为 DOM

更新已渲染的元素

React 只更新它需要更新的部分

组件 & Props

函数组件与 class 组件

渲染组件

组合组件

提取组件

Props 的只读性

State & 生命周期

将函数组件转换成 class 组件

向 class 组件中添加局部的 state

将生命周期方法添加到 Class 中

正确地使用 State

不要直接修改 State

State 的更新可能是异步的

State 的更新会被合并

数据是向下流动的

事件处理

向事件处理程序传递参数

条件渲染

元素变量

与运算符 &&

三目运算符

阻止组件渲染

列表 & Key

渲染多个组件

基础列表组件

key

用 key 提取组件

key 只是在兄弟节点之间必须唯一

在 JSX 中嵌入 map()

表单

受控组件

textarea 标签

select 标签

文件 input 标签

处理多个输入

受控输入空值

受控组件的替代品

成熟的解决方案

状态提升

添加第二个输入框

编写转换函数

状态提升

学习小结

组合 vs 继承

包含关系

特例关系

那么继承呢?

React 哲学

从设计稿开始

第一步：将设计好的 UI 划分为组件层级

第二步：用 React 创建一个静态版本

补充说明: 有关 props 和 state

第三步：确定 UI state 的最小（且完整）表示

第四步：确定 state 放置的位置

第五步：添加反向数据流

这就是全部了

第二章 高级指引

无障碍辅助功能

为什么我们需要无障碍辅助功能？

标准和指南

WCAG

WAI-ARIA

语义化的 HTML

无障碍表单

标记

在出错时提醒用户

控制焦点

键盘焦点及焦点轮廓

跳过内容机制

使用程序管理焦点

鼠标和指针事件

更复杂的部件

其他考虑因素

设置语言

设置文档标题

色彩对比度

开发及测试

键盘

开发辅助

eslint-plugin-jsx-a11y

- 在浏览器中测试无障碍辅助功能
 - aXe,aXe-core 以及 react-axe
 - WebAIM WAVE
- 无障碍辅助功能检测器和无障碍辅助功能树
- 屏幕朗读器
 - 常用屏幕朗读器
 - 火狐中的 NVDA
 - Safari 中的 VoiceOver
 - Internet Explorer 中的 JAWS
 - 其他屏幕朗读器
 - Google Chrome 中的 ChromeVox

代码分割

- 打包
 - 示例

代码分割

- `import()`

- `React.lazy`

- 异常捕获边界 (Error boundaries)

- 基于路由的代码分割

- 命名导出 (Named Exports)

Context

- 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
 - 消费多个 Context
- 注意事项
 - 过时的 API

错误边界

- 错误边界 (Error Boundaries)
 - 在线演示
 - 错误边界应该放置在哪?
 - 未捕获错误 (Uncaught Errors) 的新行为
 - 组件栈追踪
 - 关于 try/catch ?
 - 关于事件处理器
 - 自 React 15 的命名更改

Refs 转发

- 转发 refs 到 DOM 组件
 - 组件库维护者的注意事项
 - 在高阶组件中转发 refs
 - 在 DevTools 中显示自定义名称

Fragments

动机

用法

短语法

带 key 的 Fragments

在线 Demo

高阶组件

使用 HOC 解决横切关注点问题

不要改变原始组件。使用组合。

约定：将不相关的 props 传递给被包裹的组件

约定：最大化可组合性

约定：包装显示名称以便轻松调试

注意事项

不要在 render 方法中使用 HOC

务必复制静态方法

Refs 不会被传递

与第三方库协同

集成带有 DOM 操作的插件

如何解决这个问题

集成 jQuery Chosen 插件

和其他视图库集成

利用 React 替换基于字符串的渲染

把 React 嵌入到 Backbone 视图

和 Model 层集成

在 React 组件中使用 Backbone 的 Model

从 Backbone Model 提取数据

深入 JSX

指定 React 元素类型

React 必须在作用域内

在 JSX 类型中使用点语法

用户定义的组件必须以大写字母开头

在运行时选择类型

JSX 中的 Props

JavaScript 表达式作为 Props

字符串字面量

Props 默认值为 “True”

属性展开

JSX 中的子元素

字符串字面量

JSX 子元素

JavaScript 表达式作为子元素

函数作为子元素

布尔类型、Null 以及 Undefined 将会忽略

Optimizing Performance

使用生产版本

Create React App

单文件构建

Brunch

Browserify

Rollup

webpack

使用 Chrome Performance 标签分析组件

使用开发者工具中的分析器对组件进行分析

虚拟化长列表

避免调停

shouldComponentUpdate 的作用

示例

不可变数据的力量

Portals

用法

通过 Portal 进行事件冒泡

Profiler API

用法

`onRender` 回调

不使用 ES6

声明默认属性

初始化 State

自动绑定

Mixins

不使用 JSX 的 React

协调

设计动力

Diffing 算法

比对不同类型的元素

比对同一类型的元素

比对同类型的组件元素

对子节点进行递归

Keys

权衡

Refs and the DOM

何时使用 Refs

勿过度使用 Refs

创建 Refs

访问 Refs

为 DOM 元素添加 ref

为 class 组件添加 Ref

Refs 与函数组件

将 DOM Refs 暴露给父组件

回调 Refs

过时 API: String 类型的 Refs

关于回调 refs 的说明

Render Props

使用 Render Props 来解决横切关注点 (Cross-Cutting Concerns)

使用 Props 而非 `render`

注意事项

将 Render Props 与 React.PureComponent 一起使用时要小心

静态类型检查

Flow

[在项目中添加 Flow](#)

[从编译后的代码中去除 Flow 语法](#)

[Create React App](#)

[Babel](#)

[其他构建工具设置](#)

[运行 Flow](#)

[添加 Flow 类型注释](#)

[TypeScript](#)

[在 Create React App 中使用 TypeScript](#)

[添加 TypeScript 到现有项目中](#)

[配置 TypeScript 编译器](#)

[文件扩展名](#)

[运行 TypeScript](#)

[类型定义](#)

[Reason](#)

[Kotlin](#)

[其他语言](#)

[严格模式](#)

[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](#)

[使用 PropTypes 进行类型检查](#)

[PropTypes](#)

[限制单个元素](#)

[默认 Prop 值](#)

[非受控组件](#)

[默认值](#)

[文件输入](#)

[Web Components](#)

[在 React 中使用 Web Components](#)

[在 Web Components 中使用 React](#)

[第三章 Hook](#)

[Hook 简介](#)

[视频介绍](#)

[没有破坏性改动](#)

[动机](#)

[在组件之间复用状态逻辑很难](#)

[复杂组件变得难以理解](#)

[难以理解的 class](#)

[渐进策略](#)

[FAQ](#)

[下一步](#)

[Hook 概览](#)

 [State Hook](#)

[声明多个 state 变量](#)

[那么，什么是 Hook?](#)

 [Effect Hook](#)

Hook 使用规则

自定义 Hook

其他 Hook

下一步

使用 State Hook

等价的 class 示例

Hook 和函数组件

Hook 是什么?

声明 State 变量

读取 State

更新 State

总结

提示: 方括号有什么用?

提示: 使用多个 state 变量

下一步

使用 Effect Hook

无需清除的 effect

使用 class 的示例

使用 Hook 的示例

详细说明

需要清除的 effect

使用 Class 的示例

使用 Hook 的示例

小结

使用 Effect 的提示

提示: 使用多个 Effect 实现关注点分离

解释: 为什么每次更新的时候都要运行 Effect

提示: 通过跳过 Effect 进行性能优化

下一步

Hook 规则

只在最顶层使用 Hook

只在 React 函数中调用 Hook

ESLint 插件

说明

下一步

自定义 Hook

提取自定义 Hook

使用自定义 Hook

提示: 在多个 Hook 之间传递信息

`useYourImagination()`

Hook API 索引

基础 Hook

`useState`

函数式更新

惰性初始 state

跳过 state 更新

`useEffect`

清除 effect

effect 的执行时机

effect 的条件执行

`useContext`

额外的 Hook

`useReducer`

指定初始 state

惰性初始化

跳过 dispatch

`useCallback`

`useMemo`

`useRef`

`useImperativeHandle`

`useLayoutEffect`

`useDebugValue`

延迟格式化 debug 值

Hooks FAQ

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?

第一章 核心

Hello World

最简易的 React 示例如下：

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

它将在页面上展示一个“Hello, world!”的标题。

[在 CodePen 上尝试](#)

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。

JSX 简介

考虑如下变量声明：

```
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版语言，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进行分离到不同文件这种人为地分离方式，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>;  
ReactDOM.render(  
  element,  
  document.getElementById('root')  
>;
```

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>
);

ReactDOM.render(
  element,
  document.getElementById('root')
);
```

[在 CodePen 上尝试](#)

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 特定属性

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 = <div tabIndex="0"></div>;
```

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。本网站使用与其兼容的[Oceanic Next](#) 配色方案。

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.render\(\)](#)：

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

[在 CodePen 上尝试](#)

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

[ReactDOM.render\(\)](#)。

考虑一个计时器的例子：

```
function tick() {  
  const element = (  
    <div>  
      <h1>Hello, world!</h1>  
      <h2>It is {new Date().toLocaleTimeString()}.</h2>  
    </div>  
  );  
  ReactDOM.render(element, document.getElementById('root'));  
  
  setInterval(tick, 1000);
```

[在 CodePen 上尝试](#)

这个例子会在 [setInterval\(\)](#) 回调函数，每秒都调用 [ReactDOM.render\(\)](#)。

注意：

在实践中，大多数 React 应用只会调用一次 [ReactDOM.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。

你可以使用浏览器的检查元素工具查看[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>
```

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，考虑 UI 在任意给定时刻的状态，而不是随时间变化的过程，能够消灭一整类的 bug。

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 element = <welcome name="Sara" />;ReactDOM.render(  
  element,  
  document.getElementById('root')  
,
```

[在 CodePen 上尝试](#)

让我们来回顾一下这个例子中发生了什么：

1. 我们调用 `ReactDOM.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>
    );
}

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

[在 CodePen 上尝试](#)

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，并自下而上地将这些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>
      <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 上尝试](#)

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} />           className="UserInfo-name">  
            {props.author.name}  
          </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">       <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 上尝试](#)

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`），那么它就是一个可复用组件的候选项。

Props 的只读性

组件无论是使用[函数声明还是通过 class 声明](#)，都决不能修改自身的 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。

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 `ReactDOM.render()` 来修改我们想要渲染的元素：

```
function tick() {
  const element = (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {new Date().toLocaleTimeString()}.</h2>
    </div>
  );
  ReactDOM.render(element, document.getElementById('root'))
}

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：

```
function Clock(props) {
  return (
    <div>      <h1>Hello, world!</h1>      <h2>It is
    {props.date.toLocaleTimeString()}.</h2>      </div>  );
}

function tick() {
  ReactDOM.render(
    <Clock date={new Date()} />,      document.getElementById('root')
  );
}

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：

```
ReactDOM.render(
  <Clock />,  document.getElementById('root')
);
```

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>
    );
  }
}
```

1. 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。

1. 移除 `<clock />` 元素中的 `date` 属性：

```
ReactDOM.render(
  <Clock />,  document.getElementById('root')
);
```

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>
    );
  }
}

ReactDOM.render(
  <Clock />,  document.getElementById('root')
);
```

[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>
    );
  }
}

ReactDOM.render(
  <Clock />,
  document.getElementById('root')
);
```

[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 `ReactDOM.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，如此以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：

```
<h2>It is {this.state.date.toLocaleTimeString()}.</h2>
```

这对于自定义组件同样适用：

```
<FormattedMessage date={this.state.date} />
```

`FormattedMessage` 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>
  );
}

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

[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。

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。你必须显式的使用 `preventDefault`。例如，传统的 HTML 中阻止链接默认打开一个新页面，你可以这样写：

```
<a href="#" onClick="console.log('The link was clicked.');// return  
false">  
  click me  
</a>
```

在 React 中，可能是这样的：

```
function ActionLink() {  
  function handleClick(e) {    e.preventDefault();  
  console.log('The link was clicked.');//  
  }  
  return (  
    <a href="#" onClick={handleClick}>      click me  
    </a>  
  );  
}
```

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。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(state => ({        isToggleOn:  
!state.isToggleOn        }));  }  
  render() {  
    return (
```

```
        <button onClick={this.handleClick}>
{this.state.isToggleOn ? 'ON' : 'OFF'}
    </button>
);
}
}

ReactDOM.render(
<Toggle />,
document.getElementById('root')
);
```

[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。如果你正在使用实验性的 [public class fields 语法](#)，你可以使用 class fields 正确的绑定回调函数：

```
class LoggingButton extends React.Component {
// 此语法确保 `handleClick` 内的 `this` 已被绑定。 // 注意：这是 *实验性
* 语法。 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被隐式的进行传递。

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 />;}  
ReactDOM.render(  
  // Try changing to isLoggedIn={true}:  
  <Greeting isLoggedIn={false} />,  document.getElementById('root'));
```

[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}>
```

```
<button onClick={props.onClick}>
  Logout
</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>
    );
  }
}

ReactDOM.render(
  <LoginControl />,
  document.getElementById('root')
);
```

[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 中嵌入任何表达式](#)。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'];
ReactDOM.render(
  <Mailbox unreadMessages={messages} />,
  document.getElementById('root')
);
```

[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。

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.
    );
}
```

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>
);
}
}

ReactDOM.render(
  <Page />,
  document.getElementById('root')
);

```

[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。

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>);
```

我们把整个 `listItems` 插入到 `` 元素中，然后渲染进 DOM：

```
ReactDOM.render(
  <ul>{listItems}</ul>,  document.getElementById('root')
);
```

[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];
ReactDOM.render(
  <NumberList numbers={numbers} />,  document.getElementById('root')
);
```

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>
  );
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('root')
);
```

[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>
  );
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('root')
);

```

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>
  );
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('root')
);

```

[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 只是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 = [
  {id: 1, title: 'Hello World', content: 'Welcome to learning React!'},
  {id: 2, title: 'Installation', content: 'You can install React from npm.'}
];
ReactDOM.render(
  <Blog posts={posts} />,
  document.getElementById('root')
);
```

[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。

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>`）之类的表单元素通常自己维护 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>  
    <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} />
    </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 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.name === 'isGoing' ? 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.render(<input value="hi" />, mountNode);  
  
setTimeout(function() {  
  ReactDOM.render(<input value={null} />, mountNode);  
}, 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。

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} />
      </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>
```

```

<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。

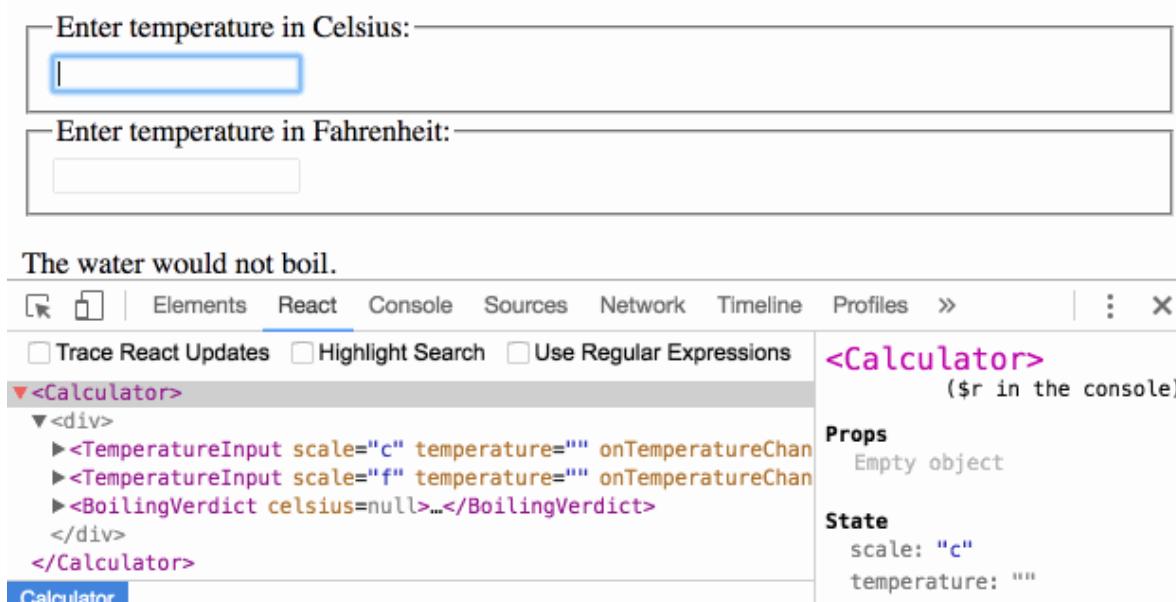
学习小结

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：



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。

React 哲学

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：

Search...	
<input type="checkbox"/>	Only show products in stock
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

该 JSON API 会返回以下数据：

```

[

  {category: "Sporting Goods", price: "$49.99", stocked: true, name: "Football"},

  {category: "Sporting Goods", price: "$9.99", stocked: true, name: "Baseball"},

  {category: "Sporting Goods", price: "$29.99", stocked: false, name: "Basketball"},

  {category: "Electronics", price: "$99.99", stocked: true, name: "iPod Touch"},

  {category: "Electronics", price: "$399.99", stocked: false, name: "iPhone 5"},

  {category: "Electronics", price: "$199.99", stocked: true, name: "Nexus 7"}
```

];

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，所以如果你的数据模型设计得恰当，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。

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

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。

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`

第二步：用 React 创建一个静态版本

参阅 [CodePen](#) 上的 [React 哲学：第二步](#)。

现在我们已经确定了组件层级，可以编写对应的应用了。最容易的方式，是先用已有的数据模型渲染一个不包含交互功能的 UI。最好将渲染 UI 和添加交互这两个过程分开。这是因为，编写一个应用的静态版本时，往往要编写大量代码，而不需要考虑太多交互细节；添加交互功能时则要考虑大量细节，而不需要编写太多代码。所以，将这两个过程分开进行更为合适。我们会在接下来的代码中体会到其中的区别。

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 `ReactDOM.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

参阅 [CodePen](#) 上的 [React 哲学：第四步](#)。

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`。
- 因此，搜索词和复选框的值应该很自然地存放在 `FilterableProductTable` 组件中。

很好，我们已经决定把这些 state 存放在 `FilterableProductTable` 组件中。首先，将实例属性 `this.state = {filterText: '', instockOnly: false}` 添加到 `FilterableProductTable` 的 `constructor` 中，设置应用的初始 state；接着，将 `filterText` 和 `instockOnly` 作为 props 传入 `ProductTable` 和 `SearchBar`；最后，用这些 props 筛选 `ProductTable` 中的产品信息，并设置 `SearchBar` 的表单值。

你现在可以看到应用的变化了：将 `filterText` 设置为 `"ball"` 并刷新应用，你能发现表格中的数据已经更新了。

第五步：添加反向数据流

参阅 [CodePen](#) 上的 [React 哲学：第五步](#)。

到目前为止，我们已经借助自上而下传递的 props 和 state 渲染了一个应用。现在，我们将尝试让数据反向传递：处于较低层级的表单组件更新较高层级的 `FilterableProductTable` 中的 state。

React 通过一种比传统的双向绑定略微繁琐的方法来实现反向数据传递。尽管如此，但这种需要显式声明的方法更有助于人们理解程序的运作方式。

如果你在这时尝试在搜索框输入或勾选复选框，React 不会产生任何响应。这是正常的，因为我们之前已经将 `input` 的值设置为了从 `FilterableProductTable` 的 `state` 传递而来的固定值。

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。:)

第二章 高级指引

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

```
        <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 [react-axe](#) 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](#)

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:

最终你的打包文件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，你会拥有一个可以直接使用的 Webpack 配置来进行打包工作。

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()

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

注意：

`React.lazy` 和 `Suspense` 技术还不支持服务端渲染。如果你想要在使用服务端渲染的应用中使用，我们推荐 [Loadable Components](#) 这个库。它有一个很棒的[服务端渲染打包指南](#)。

`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>
  );
}
```

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, Route, Switch } from 'react-router-dom';

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

const App = () => (
  <Router>
    <Suspense fallback={<div>Loading...</div>}>
      <Switch>
        <Route exact path="/" component={Home}/>
        <Route path="/about" component={About}/>
      </Switch>
    </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"));
```

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，但这种做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：

```
class App extends React.Component {  
  render() {  
    return <Toolbar theme="dark" />;  
  }  
}  
  
function Toolbar(props) {  
  // Toolbar 组件接受一个额外的“theme”属性，然后传递给 ThemedButton 组件。  
  // 如果应用中每一个单独的按钮都需要知道 theme 的值，这会是一件很麻烦的事， //  
  // 因为必须将这个值层层传递所有组件。  return (<div>  
    <ThemedButton theme={props.theme} />  
  </div> );
```

```
}

class ThemedButton extends React.Component {
  render() {
    return <Button theme={this.props.theme} />;
  }
}
```

使用 context, 我们可以避免通过中间元素传递 props:

```
// Context 可以让我们无须明确地传遍每一个组件, 就能将值深入传递进组件树。// 为
当前的 theme 创建一个 context (“light”为默认值)。const ThemeContext =
React.createContext('light');class App extends React.Component {
  render() {
    // 使用一个 Provider 来将当前的 theme 传递给以下的组件树。      // 无论多
深, 任何组件都能读取这个值。      // 在这个例子中, 我们将 “dark” 作为当前的值传
递下去。    return (
      <ThemeContext.Provider value="dark">          <Toolbar />
      </ThemeContext.Provider>
    );
  }
}

// 中间的组件再也不必指明往下传递 theme 了。function Toolbar() {  return (
  <div>
    <ThemedButton />
  </div>
);
}

class ThemedButton extends React.Component {
  // 指定 contextType 读取当前的 theme context。 // React 会往上找到最近
的 theme Provider, 然后使用它的值。 // 在这个例子中, 当前的 theme 值为
“dark”。 static contextType = ThemeContext;
  render() {
    return <Button theme={this.context} />;  }
}
```

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](#)，因而中间组件无需知道 `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

```
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。这有助于在不使用 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

```
<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。Provider 及其内部 consumer 组件都不受制于 `shouldComponentUpdate` 函数，因此当 consumer 组件在其祖先组件退出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

```
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` 属性会被重赋值为一个由 `React.createContext()` 创建的 Context 对象。这能让你使用 `this.context` 来消费最近 Context 上的那个值。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

```
<MyContext.Consumer>  
  {value => /* 基于 context 值进行渲染 */}  
</MyContext.Consumer>
```

这里，React 组件也可以订阅到 context 变更。这能让你在[函数式组件](#)中完成订阅 context。

这需要[函数作为子元素 \(function as a child\)](#) 这种做法。这个函数接收当前的 context 值，返回一个 React 节点。传递给函数的 `value` 值等同于往上组件树离这个 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

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

对于上面的 theme 例子，使用动态值（dynamic values）后更复杂的用法：

theme-context.js

```
export const themes = {
  light: {
    foreground: '#000000',
    background: '#eeeeee',
  },
  dark: {
    foreground: '#ffffff',
    background: '#222222',
  },
};

export const ThemeContext = React.createContext(themes.dark // 默认
值);
```

themed-button.js

```
import {ThemeContext} from './theme-context';

class ThemedButton extends React.Component {
  render() {
    let props = this.props;
    let theme = this.context;    return (
      <button
        {...props}
        style={{backgroundColor: theme.background}}
      />
    );
  }
}

ThemedButton.contextType = ThemeContext;
export default ThemedButton;
```

app.js

```
import {ThemeContext, themes} from './theme-context';
import ThemedButton from './themed-button';

// 一个使用 ThemedButton 的中间组件
function Toolbar(props) {
  return (
    <ThemedButton onClick={props.changeTheme}>
      Change Theme
    </ThemedButton>
  );
}

class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      theme: themes.light,
    };

    this.toggleTheme = () => {
      this.setState(state => ({
        theme:
          state.theme === themes.dark
            ? themes.light
            : themes.dark,
      }));
    };
  }

  render() {
    // 在 ThemeProvider 内部的 ThemedButton 按钮组件使用 state 中的 theme
    // 值， // 而外部的组件使用默认的 theme 值
    return (
      <Page>
        <ThemeContext.Provider value={this.state.theme}>
          <Toolbar changeTheme={this.toggleTheme} />
        </ThemeContext.Provider>
        <Section>
          <ThemedButton />
        </Section>
      </Page>
    );
  }
}
```

```
ReactDOM.render(<App />, document.root);
```

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

```
// 确保传递给 createContext 的默认值数据结构是调用的组件（consumers）所能匹配的！
export const ThemeContext = React.createContext({
  theme: themes.dark,  toggleTheme: () => {}
});
```

theme-toggler-button.js

```
import {ThemeContext} from './theme-context';

function ThemeTogglerButton() {
  // Theme Toggler 按钮不仅仅只获取 theme 值，它也从 context 中获取到一个
  // toggleTheme 函数
  return (
    <ThemeContext.Consumer>
      {(theme, toggleTheme) => (
        <button
          onClick={toggleTheme}
          style={{backgroundColor: theme.background}}
        >
          Toggle Theme
        </button>
      )}
    </ThemeContext.Consumer>
  );
}

export default ThemeTogglerButton;
```

app.js

```
import {ThemeContext, themes} from './theme-context';
import ThemeTogglerButton from './theme-toggler-button';

class App extends React.Component {
```

```

constructor(props) {
  super(props);

  this.toggleTheme = () => {
    this.setState(state => ({
      theme:
        state.theme === themes.dark
          ? themes.light
          : themes.dark,
    }));
  };
}

// State 也包含了更新函数，因此它会被传递进 context provider。
this.state = { theme: themes.light,
  toggleTheme: this.toggleTheme,
}; }

render() {
  // 整个 state 都被传递进 provider    return (
    <ThemeContext.Provider value={this.state}>           <Content />
    </ThemeContext.Provider>
  );
}
}

function Content() {
  return (
    <div>
      <ThemeTogglerButton />
    </div>
  );
}

ReactDOM.render(<App />, document.root);

```

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。

```

// Theme context, 默认的 theme 是 “light” 值
const ThemeContext = React.createContext('light');

// 用户登录 context

```

```

const UserContext = React.createContext({
  name: 'Guest',
});

class App extends React.Component {
  render() {
    const {signedInUser, theme} = this.props;

    // 提供初始 context 值的 App 组件
    return (
      <ThemeContext.Provider value={theme}>
        <UserContext.Provider value={signedInUser}> <Layout />
          </UserContext.Provider> </ThemeContext.Provider>
      );
    }
  }

  function Layout() {
    return (
      <div>
        <Sidebar />
        <Content />
      </div>
    );
  }

  // 一个组件可能会消费多个 context
  function Content() {
    return (
      <ThemeContext.Consumer> {theme => (
        <UserContext.Consumer> {user => (
          <ProfilePage
            user={user} theme={theme} />
        )}
      )} </ThemeContext.Consumer>
    );
  }
}

```

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 会使用参考标识 (reference identity) 来决定何时进行渲染，这里可能会有一些陷阱，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，以下的代码会重渲染所有下面的 consumers 组件，因为 `value` 属性总是被赋值为新的对象：

```
class App extends React.Component {
  render() {
    return (
      <MyContext.Provider value={{something: 'something'}}>
        <Toolbar />
      </MyContext.Provider>
    );
  }
}
```

为了防止这种情况，将 value 状态提升到父节点的 state 里：

```
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: {something: 'something'},      };
  }

  render() {
    return (
      <Provider value={this.state.value}>          <Toolbar />
      </Provider>
    );
  }
}
```

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。

错误边界

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

错误边界 (Error Boundaries)

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

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

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

查看使用 [React 16 定义和使用错误边界的例子](#)。

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。

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：

```
function FancyButton(props) {
  return (
    <button className="FancyButton">
      {props.children}
    </button>
  );
}
```

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`：

```
const FancyButton = React.forwardRef((props, ref) => (
  <button ref={ref} className="FancyButton">{props.children}</button>
));
// 你可以直接获取 DOM button 的 ref:
const ref = React.createRef();
<FancyButton ref={ref}>Click me!</FancyButton>;
```

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：

```
function logProps(wrappedComponent) { class LogProps extends React.Component {
  componentDidUpdate(prevProps) {
    console.log('old props:', prevProps);
    console.log('new props:', this.props);
  }

  render() {
    return <wrappedComponent {...this.props} />;
  }
}

return LogProps;
}
```

“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`:

```
class FancyButton extends React.Component {
  focus() {
    // ...
  }

  // ...
}

// 我们导出 LogProps，而不是 FancyButton。
// 虽然它也会渲染一个 FancyButton。
export default logProps(FancyButton);
```

上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：

```
import FancyButton from './FancyButton';

const ref = React.createRef();
// 我们导入的 FancyButton 组件是高阶组件（HOC）LogProps。
// 尽管渲染结果将是一样的，
// 但我们的 ref 将指向 LogProps 而不是内部的 FancyButton 组件！
// 这意味着我们不能调用例如 ref.current.focus() 这样的方法
<FancyButton
  label="Click Me"
  handleClick={handleClick}
  ref={ref}/>;
```

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：

```
function LogProps(Component) {
  class LogProps extends React.Component {
    componentDidUpdate(prevProps) {
      console.log('old props:', prevProps);
      console.log('new props:', this.props);
    }

    render() {
      const {forwardedRef, ...rest} = this.props;
      // 将自定义的 prop 属性 “forwardedRef” 定义为 ref
      return <Component ref={forwardedRef} {...rest} />;    }
  }

  // 注意 React.forwardRef 回调的第二个参数 “ref”。
  // 我们可以将其作为常规 prop 属性传递给 LogProps，例如 “forwardedRef”
  // 然后它就可以被挂载到被 LogProps 包裹的子组件上。
  return React.forwardRef((props, ref) => {    return <LogProps
  {...props} forwardedRef={ref} />;  });
}
```

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*”：

```
const WrappedComponent = React.forwardRef((props, ref) => {
  return <LogProps {...props} forwardedRef={ref} />;
});
```

如果你命名了渲染函数，DevTools 也将包含其名称（例如 “*ForwardRef(myFunction)*”）：

```
const wrappedComponent = React.forwardRef(
  function myFunction(props, ref) {
    return <LogProps {...props} forwardedRef={ref} />;
  }
);
```

你甚至可以设置函数的 `displayName` 属性来包含被包裹组件的名称：

```
function LogProps(Component) {
  class LogProps extends React.Component {
    // ...
  }

  function forwardRef(props, ref) {
    return <LogProps {...props} forwardedRef={ref} />;
  }

  // 在 DevTools 中为该组件提供一个更有用的显示名。
  // 例如 “ForwardRef(logProps(MyComponent))”
  const name = Component.displayName || Component.name;
  forwardRef.displayName = `LogProps(${name})`;
  return React.forwardRef(forwardRef);
}
```

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>  
      </>    );  
  }  
}
```

你可以像使用任何其他元素一样使用 `<> </>`，除了它不支持 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。

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。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。容器组件担任分离将高层和低层关注的责任，由容器管理订阅和状态，并将 prop 传递给处理渲染 UI。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。

```
// 而不是这样...  
const EnhancedComponent = withRouter(connect(commentSelector)  
(WrappedComponent))  
  
// ... 你可以编写组合工具函数  
// 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 专门处理的。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。](#)

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所做的事情。

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 `ReactDOM.render()` 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, `ReactDOM.render()` 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>;
}

ReactDOM.render(
  <Button />,
  document.getElementById('container'),
  function() {
    $('#btn').click(function() {
      alert('Hello!');
    });
  }
);
```

从这起你可开始可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} />;

}

ReactDOM.render(
  <HelloButton />,
  document.getElementById('container')
);
```

[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.render()` 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.render()` 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.render\(\)`](#)：

```
function Paragraph(props) {  return <p>{props.text}</p>;}

const ParagraphView = Backbone.View.extend({  render() {
    const text = this.model.get('text');
    ReactDOM.render(<Paragraph text={text} />, this.el);    return
this;
},
remove() {
    ReactDOM.unmountComponentAtNode(this.el);
Backbone.View.prototype.remove.call(this);
}
});
```

[在 CodePen 上运行](#)

在 `remove` 方法中我们也需要调用 `ReactDOM.unmountComponentAtNode()` 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' });
ReactDOM.render(
  <Example model={model} />,
  document.getElementById('root')
);
```

[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，并选择性地，拷贝数据到本地 React state，来将 React 用于任何 model 库。

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" />  
<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 &amp; 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，`0` 仍然会被渲染：

```
<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>
```

Optimizing Performance

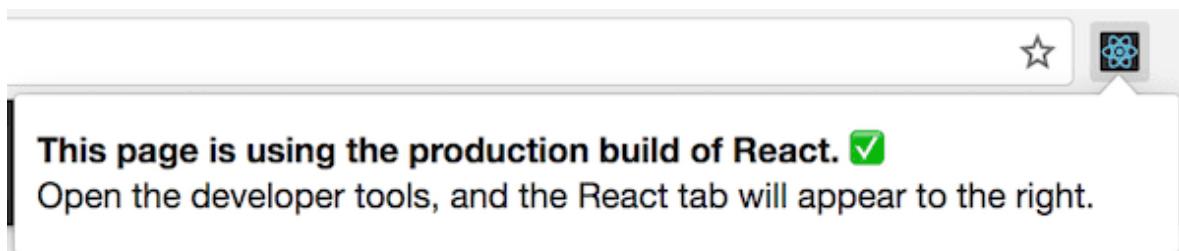
UI 更新需要昂贵的 DOM 操作，而 React 内部使用几种巧妙的技术以便最小化 DOM 操作次数。对于大部分应用而言，使用 React 时无需专门优化就已拥有高性能的用户界面。尽管如此，你仍然有办法来加速你的 React 应用。

使用生产版本

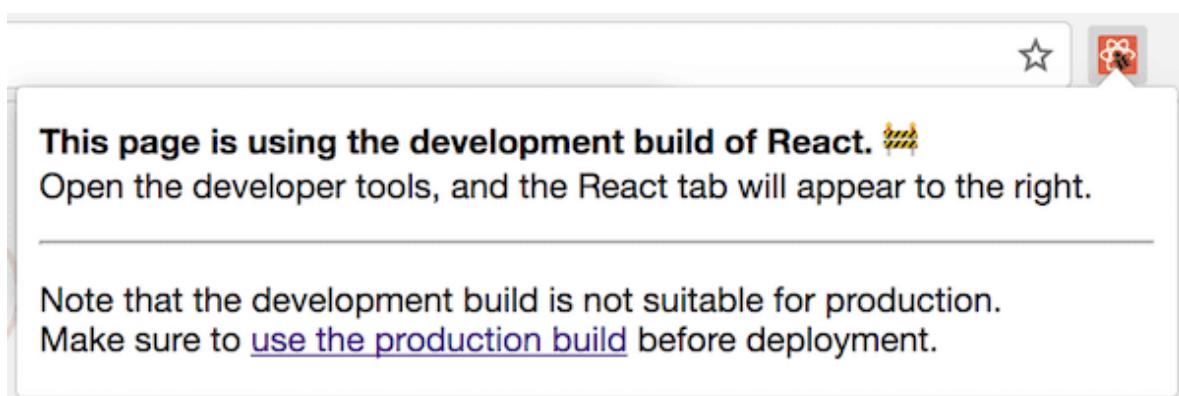
当你需要对你的 React 应用进行 benchmark，或者遇到了性能问题，请确保你正在使用压缩后的生产版本。

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@16/umd/react.production.min.js">
</script>
<script src="https://unpkg.com/react-dom@16/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');

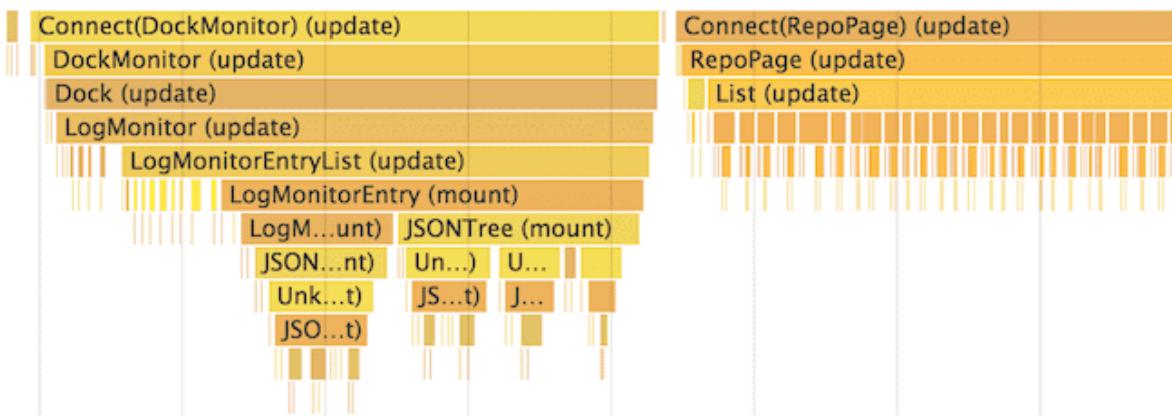
module.exports = {
  mode: 'production',
  optimization: {
    minimizer: [new TerserPlugin({ /* additional options here */ })],
  },
};
```

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 Chrome Performance 标签分析组件

在[开发](#)模式下，你可以通过支持的浏览器可视化地了解组件是如何挂载、更新以及卸载的。例如：



在 Chrome 中进行如下操作：

1. 临时禁用所有的 Chrome 扩展，尤其是 React 开发者工具。他们会严重干扰度量结果！
2. 确保你是在 React 的开发模式下运行应用。
3. 打开 Chrome 开发者工具的 [Performance](#) 标签并按下 **Record**。
4. 对你想分析的行为进行复现。尽量在 20 秒内完成以避免 Chrome 卡住。
5. 停止记录。
6. 在 [User Timing](#) 标签下会显示 React 归类好的事件。

你可以查阅 [Ben Schwarz 的文章](#)以获取更详尽的指导。

需要注意的是在生产环境中组件会相对渲染得更快些。当然了，这能帮助你查看是否有不相关的组件被错误地更新，以及 UI 更新的深度和频率。

目前只有 Chrome、Edge 和 IE 支持该功能，但是我们使用的是标准的[用户计时 API](#)。我们期待有更多浏览器能支持它。

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。

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 节点以及没有必要的节点访问，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：

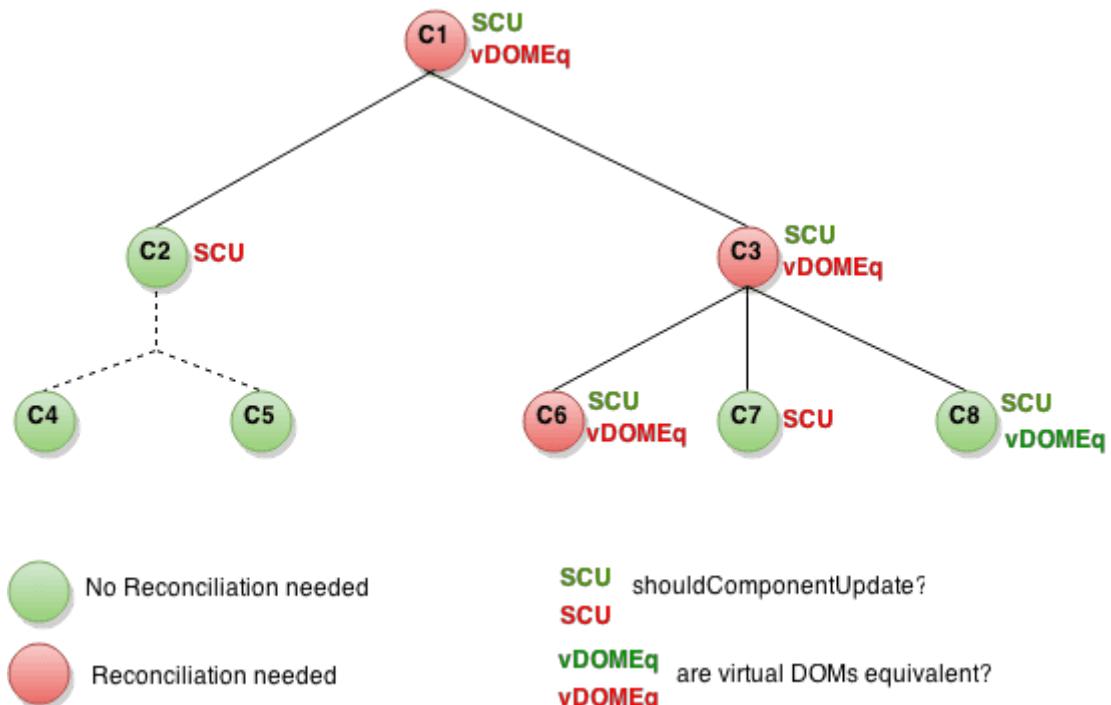
```
shouldComponentUpdate(nextProps, nextState) {
  return true;
}
```

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'};  
}
```

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() {    // 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>  
  );  
}  
  
ReactDOM.render(<Parent />, appRoot);
```

[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。

Profiler API

`Profiler` 测量渲染一个 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>
      <Profiler id="PreviewPane" onRender={callback}>
        <PreviewPane {...props} />
      </Profiler>
    </Panel>
  </Profiler>
)>;

```

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
  actualDuration, // 本次更新 committed 花费的渲染时间
  baseDuration, // 估计不使用 memoization 的情况下渲染整颗子树需要的时间
  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:

- `id: string` - 发生提交的 `Profiler` 树的 `id`。如果有多个 profiler，它能用来分辨树的哪一部分发生了“提交”。
- `phase: "mount" | "update"` - 判断是组件树的第一次装载引起的重渲染，还是由 `props`、`state` 或是 `hooks` 改变引起的重渲染。
- `actualDuration: number` - 本次更新在渲染 `Profiler` 和它的子代上花费的时间。这个数值表明使用 `memoization` 之后能表现得多好。（例如 [React.memo](#)，

`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

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 `get defaultProps()` 函数：

```
var Greeting = createReactClass({  
  getDefaultProps: 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。

如果你觉得上述写法很繁琐，那么可以尝试使用**目前还处于试验性阶段**的 Babel 插件 [Class Properties](#)。

```
class SayHello extends React.Component {
  constructor(props) {
    super(props);
    this.state = {message: 'Hello!'};
  }
  // 警告：这种语法还处于试验性阶段！
  // 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>
    );
  }
}
```

请注意，上面这种语法**目前还处于试验性阶段**，这意味着语法随时都可能改变，也存在最终不被列入框架标准的可能。

为了保险起见，以下三种做法都是可以的：

- 在 constructor 中绑定方法。
- 使用箭头函数，比如：`onClick={(e) => this.handleClick(e)}`。
- 继续使用 `createClass`。

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 (
      <div>
        <h1>{this.state.seconds}</h1>
      </div>
    );
  }
});
```

```
<p>
  React has been running for {this.state.seconds} seconds.
</p>
);
}

);

ReactDOM.render(
  <TickTock />,
  document.getElementById('example')
);
```

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>;
  }
}

ReactDOM.render(
  <Hello towhat="world" />,
  document.getElementById('root')
);
```

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}`);
  }
}

ReactDOM.render(
  React.createElement(Hello, {towhat: 'World'}, null),
  document.getElementById('root')
);
```

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;

ReactDOM.render(
  e('div', null, 'Hello world'),
  document.getElementById('root')
);
```

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 的底层实现没有具体了解的情况下编写应用。在开发者编写应用时虽然保持相对简单的心智，但开发者无法了解内部的实现机制。本文描述了在实现 React 的“diffing”算法中我们做出的设计决策以保证组件满足更新具有可预测性，以及在繁杂业务下依然保持应用的高性能性。

设计动力

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有效率的更新 UI 以保证当前 UI 与最新的树保持同步。

这个算法问题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数。然而，即使在[最前沿的算法中](#)，该算法的复杂程度为 $O(n^3)$ ，其中 n 是树中元素的数量。

如果在 React 中使用了该算法，那么展示 1000 个元素所需要执行的计算量将在十亿的量级范围。这个开销实在是太过高昂。于是 React 在以下两个假设的基础之上提出了一套 $O(n)$ 的启发式算法：

1. 两个不同类型的元素会产生出不同的树；
2. 开发者可以通过 `key` prop 来暗示哪些子元素在不同的渲染下能保持稳定；

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

当对比两颗树时，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。

比对不同类型的元素

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。

当拆卸一棵树时，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 `componentWillMount()` 方法，紧接着 `componentDidMount()` 方法。所有跟之前的树所关联的 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。

比对同一类型的元素

当比对两个相同类型的 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" />
```

通过比对这两个元素，React 知道只需要修改 DOM 元素上的 `className` 属性。

当更新 `style` 属性时，React 仅更新有所变更的属性。比如：

```
<div style={{color: 'red', fontWeight: 'bold'}} />

<div style={{color: 'green', fontWeight: 'bold'}} />
```

通过比对这两个元素，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。

比对同类型的组件元素

当一个组件更新时，组件实例保持不变，这样 state 在跨越不同的渲染时保持一致。React 将更新该组件实例的 props 以跟最新的元素保持一致，并且调用该实例的 `componentWillReceiveProps()` 和 `componentWillUpdate()` 方法。

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

对子节点进行递归

在默认条件下，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，变更开销比较小。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。

如果简单实现的话，那么在列表头部插入会很影响性能，那么变更开销会比较大。比如：

```
<ul>
  <li>Duke</li>
  <li>villanova</li>
</ul>

<ul>
  <li>Connecticut</li>
  <li>Duke</li>
  <li>villanova</li>
</ul>
```

React 会针对每个子元素 mutate 而不是保持相同的 `Duke` 和

`villanova` 子树完成。这种情况下低效可能会带来性能问题。

Keys

为了解决以上问题，React 支持 `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。以下例子在新增 `key` 之后使得之前的低效转换变得高效：

```
<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。

现实场景中，产生一个 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，但一旦有顺序修改，diff 就会变得慢。

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）可能相互篡改导致无法预期的变动。

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。

我们定期探索优化算法，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 依赖探索的算法，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。如果你发现你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。

Refs and the DOM

Refs 提提供了一种方式，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。

Render Props

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

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

```
<DataProvider render={data => (
  <h1>Hello {data.target}</h1>
)}>
```

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，但如何分享一个组件封装到其他需要相同 state 组件的状态或行为并不总是很容易。

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鼠标的猫的图片。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}>

    {/* 
      我们可以在这里换掉 <p> 的 <Cat> ..... 
      但是接着我们需要创建一个单独的 <MouseWithSomethingElse>
      每次我们需要使用它时，<MouseWithCat> 是不是真的可以重复使用.
    */}
    <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：我们可以提供一个带有函数 prop 的 `<Mouse>` 组件，它能够动态决定什么需要渲染的，而不是将 `<Cat>` 硬编码到 `<Mouse>` 组件里，并有效地改变它的渲染结果。

```

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}>

/*
Instead of providing a static representation of what
<Mouse> renders,
use the `render` prop to dynamically determine what to
render.
*/
{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`

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！
          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，因为你需要关闭组件的 props 和/或 state），则 `<Mouse>` 应该扩展 `React.Component`。

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：基本类型](#)
- [TypeScript 文档：JavaScript 迁移](#)
- [TypeScript 文档：React 与 Webpack](#)

Reason

[**Reason**](#) 不是一种新的语言；它是一种新的语法和工具链，底层使用的是经过实战验证的 [OCaml](#) 语言。Reason 在 OCaml 之上提供了 JavaScript 程序员的熟悉语法，而且集成了现有的 NPM/Yarn 工作流。

Reason 是由 Facebook 开发，并且运用在一些现有产品中比如 Messenger。虽然它有一定的实验性质，但它拥有由 Facebook 维护的[专门的 React 绑定](#)和一个[活跃的社区](#)。

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 [E#/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：

```
import React from 'react';

function ExampleApplication() {
  return (
    <div>
      <Header />
      <React.StrictMode>          <div>
        <ComponentOne />
        <ComponentTwo />
      </div>
      </React.StrictMode>          <Footer />
    </div>
  );
}
```

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](#)

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：

```
class MyComponent extends React.Component {  
  constructor(props) {  
    super(props);  
  
    this.inputRef = React.createRef();  }  
  
  render() {  
    return <input type="text" ref={this.inputRef} />;  }  
  
  componentDidMount() {  
    this.inputRef.current.focus();  }  
}
```

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：

```
class TopLevelRoute extends React.Component {  
  constructor(props) {  
    super(props);  
  
    SharedApplicationState.recordEvent('ExampleComponent');  
  }  
}
```

这段代码看起来似乎没有问题。但是如果 `SharedApplicationState.recordEvent` 不是 [幂等的](#)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 可能在开发过程中可能不会表现出来，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。

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。

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
};
```

`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 `onOfType` 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

您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":
ReactDOM.render(
  <Greeting />,
  document.getElementById('example')
);
```

如果你正在使用像 [transform-class-properties](#) 的 Babel 转换工具，你也可以在 React 组件类中声明 `defaultProps` 作为静态属性。此语法提案还没有最终确定，需要进行编译后才能在浏览器中运行。要了解更多信息，请查阅 [class fields proposal](#)。

```
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`。

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} />
          <input type="submit" value="Submit" />
        </label>
      </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`。

```
render() {
  return (
    <form onSubmit={this.handleSubmit}>
      <label>
        Name:
        <input type="text" defaultValue="John" />
      </label>
    </form>
  );
}
```

```
<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。

```
class FileInput extends React.Component {  
  constructor(props) {  
    super(props);  
    this.handleSubmit = this.handleSubmit.bind(this);  
    this.fileInput = React.createRef();  
  }  
  handleSubmit(event) {  
    event.preventDefault();  
    alert(`Selected file - ${this.fileInput.current.files[0].name}`);  
  }  
  
  render() {  
    return (  
      <form onSubmit={this.handleSubmit}>  
        <label>  
          Upload file:  
          <input type="file" ref={this.fileInput} />  
        </label>  
        <br />  
        <button type="submit">Submit</button>  
    );  
  }  
}
```

```
        </form>
    );
}

ReactDOM.render(
  <FileInput />,
  document.getElementById('root')
);
```

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);
    ReactDOM.render(<a href={url}>{name}</a>, mountPoint);
  }
}
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。

第三章 Hook

Hook 简介

Hook 是 React 16.8 的新增特性。它可以在你不编写 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，并且可以学习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，这往往会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。没有稳定的[语法提案](#)，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 to 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。**

Hook 概览

[开始学习 React 进阶教程，一线大厂前端必备技能立即领取](#)

Hook 是 React 16.8 的新增特性。它可以在你不编写 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.state` 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>
  );
}
```

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。

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 FriendListItems(props) {
  const isOnline = useFriendStatus(props.friend.id);
  return (
    <li style={{ color: isOnline ? 'green' : 'black' }}>
      {props.friend.name}
    </li>
  );
}
```

这两个组件的 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。

使用 State Hook

[开始学习 React 进阶教程，一线大厂前端必备技能立即领取](#)

Hook 是 React 16.8 的新增特性。它可以在不编写 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它转化为 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。

使用 Effect Hook

[开始学习 React 进阶教程，一线大厂前端必备技能立即领取](#)

Hook 是 React 16.8 的新增特性。它可以在不编写 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);
  }
  handleStatusChange(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：

```

// Mount with { friend: { id: 100 } } props
ChatAPI.subscribeToFriendStatus(100, handleStatusChange); // 运行
第一个 effect

// Update with { friend: { id: 200 } } props
ChatAPI.unsubscribeFromFriendStatus(100, handleStatusChange); // 清除
上一个 effect

```

```
ChatAPI.subscribeToFriendStatus(200, handleStatusChange); // 运行  
下一个 effect  
  
// Update with { friend: { id: 300 } } props  
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。**

Hook 规则

[开始学习 React 进阶教程，一线大厂前端必备技能立即领取](#)

Hook 是 React 16.8 的新增特性。它可以在你不编写 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 函数的最顶层调用它们。 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？

```

// ● 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) // ● 此 Hook 被忽略!
useState('Poppins')      // ● 2 (之前为 3)。读取变量名为 surname 的
state 失败
useEffect(updateTitle)   // ● 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，对应得是 `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。

自定义 Hook

[开始学习 React 进阶教程，一线大厂前端必备技能立即领取](#)

Hook 是 React 16.8 的新增特性。它可以在你不编写 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 = [
  { id: 1, name: 'Phoebe' },
  { id: 2, name: 'Rachel' },
  { id: 3, name: 'Ross' },
];

function ChatRecipientPicker() {
  const [recipientID, setRecipientID] = useState(1);
  const isRecipientOnline = useFriendStatus(recipientID);
  return (
    <>
      <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()

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。

Hook API 索引

[开始学习 React 进阶教程，一线大厂前端必备技能立即领取](#)

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](#)

基础 Hook

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 `useState` 结合展开运算符来达到合并更新对象的效果。

```
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

调用 State Hook 的更新函数并传入当前的 state 时，React 将跳过子组件的渲染及 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。

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，在浏览器完成布局与绘制之后，传给 `useEffect` 的函数会延迟调用。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，因此不应在函数中执行阻塞浏览器更新屏幕的操作。

然而，并非所有 effect 都可以被延迟执行。例如，在浏览器执行下一次绘制前，用户可见的 DOM 变更就必须同步执行，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。

虽然 `useEffect` 会在浏览器绘制后延迟执行，但会保证在任何新的渲染前执行。React 将在组件更新前刷新上一轮渲染的 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

```
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

```
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](#)。

以下是用 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

```
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

```
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与渲染无关的操作，诸如副作用这类的操作属于 `useEffect` 的适用范畴，而不是 `useMemo`。

如果没有提供依赖项数组，`useMemo` 在每次渲染时都会计算新的值。

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

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

```
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保持不变。

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(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

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再显示（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(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 `toLocaleString` 函数调用：

```
useDebugValue(date, date => date.toLocaleString());
```

Hooks FAQ

[开始学习 React 进阶教程，一线大厂前端必备技能立即领取](#)

Hook 是 React 16.8 的新增特性。它可以在你不编写 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些什么。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。

目前 Hook 还处于早期阶段，一些第三方的库可能还暂时无法兼容 Hook。

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';
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.render(<Counter />, container);  });  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，甚至存在一些伪真理，但如果没有任何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 和避免伪真理之前取得平衡，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。

如果你错过自动合并，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，你可以为之创建一个自定义 Hook。）

如何获取上一轮的 props 或 state？

目前，你可以 [通过 ref](#) 来手动实现：

```
function Counter() {
  const [count, setCount] = useState(0);

  const prevCountRef = useRef();
  useEffect(() => {
    prevCountRef.current = count;
  });
  const prevCount = prevCountRef.current;
  return <h1>Now: {count}, before: {prevCount}</h1>;
}
```

这或许有一点错综复杂，但你可以把它抽取成一个自定义 Hook:

```
function Counter() {
  const [count, setCount] = useState(0);
  const prevCount = usePrevious(count);  return <h1>Now: {count},
before: {prevCount}</h1>;
}

function usePrevious(value) {  const ref = useRef();
useEffect(() => {
  ref.current = value;
});
return ref.current;
}
```

注意看这是如何作用于 props, state, 或任何其他计算出来的值的。

```
function Counter() {
  const [count, setCount] = useState(0);

  const calculation = count + 100;
  const prevCalculation = usePrevious(calculation);  // ...
```

考虑到这是一个相对常见的使用场景，很可能在未来 React 会自带一个 `usePrevious` Hook。

参见 [derived state 推荐模式](#).

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。

In this example, the callback ref will be called only when the component mounts and unmounts, since the rendered `<h1>` component stays present throughout any rerenders. If you want to be notified any time a component resizes, you may want to use [ResizeObserver](#) or a third-party Hook built on it.

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的方法。

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`、`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}`);
    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。

如果处于某些原因你 *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);
  }, []);
  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。然后你就可以对它进行读写了。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);
}, []);
// 这个 effect 从不会重新执行}

```

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，你依然可以选择是要把应用的 `state` 作为 `props` 向下传递（更显明确）还是作为作为 `context`（对很深的更新而言更加方便）。如果你也使用 `context` 来向下传递 `state`，请使用两种不同的 `context` 类型——`dispatch` `context` 永远不会变，因此组件通过读取它就不需要重新渲染了，除非它们还需要应用的 `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。

同时也请注意这种模式在 [并行模式](#) 下可能会导致一些问题。我们计划在未来提供一个更加合理的替代方案，但当下最安全的解决方案是，如果回调所依赖的值变化了，总是让回调失效。

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 保持对当先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 团队的其它成员的提炼。