React 实验室: 我们都在研究什么 – 2022 六月

Every June 15, 2022 by Andrew Clark, Dan Abramov, Jan Kassens, Joseph Savona, Josh Story, Lauren Tan, Luna Ruan, Mengdi Chen, Rick Hanlon, Robert Zhang, Sathya Gunasekaran, Sebastian Markbåge, and Xuan Huang

React 18 着实酝酿了很多年，但它也为 React 团队带来了宝贵的经验。它的发布是多年的研究和众多路线探索的结果。其中一些路线是成功的；更多的则是在进入了死胡同后换来了一些新的洞见。我们学到的一个教训是，如果让社区干等新功能的发布，却无从知晓我们正在探索的事情的话，会比较令人沮丧。

我们通常在任何时候都有许多同时进行的项目，有更具实验性的，也有一些更加明确的。以后，我们将会定期与社区分享我们在进行的这些项目。

当然，大家要有一个心理预期，就是这些项目并没有一个明确的时间表。其中许多项目还处于高度研究的状况，很难确定具体的发布日期。有一些甚至可能最终都不会发布。即便如此，我们还是想与你分享我们都在思考哪些问题，以及我们迄今为止我们都有哪些进展。

服务端组件

我们于 2020 年 12 月宣布了 React 服务端组件 (RSC) 的实验性方案 。从那时起，我们就一直在尽力完成 React 18 中 RSC 所依赖的一些能力，并致力于根据实验情况做一些改进。

值得一提得是，我们放弃了必须使用 react-fetch 这样专用 I/O 库的路线，而是决定采用一个兼容 async/await 的路线。严格地说，这个改动并不会阻碍 RSC 的发布，目前你仍然可以继续用路由（routers）来获取数据。另一个变化是我们已经决定不用文件扩展名这种方式，而是转而考虑使用 标注来确定边界（annotating boundaries）。

我们正在与 Vercel 和 Shopify 合作，在 Webpack 与 Vite 中找到一种通用的语义来统一 bundler 上的的支持。在发布之前，我们希望确保 RSC 的语义在整个 React 生态系统中是相同的，这样才能达到稳定。

静态服务端渲染优化

静态站点生成 (SSG) 和增量静态重生成 (ISR) 都是通过让页面可以被缓存而提升性能的好方法，但我们觉得也有必要添加新的能力来提高动态服务器端渲染 (SSR) 的性能 —— 特别是当大多数内容都可以被缓存但有一部分仍然是动态的时候。我们正在探索如何利用编译期优化来优化服务器渲染。

React 优化编译器

我们在 React Conf 2021 上预告 了 React Forget 项目。它是一个可以自动生成等效于 useMemo 和 useCallback 代码的编译器，旨在保持 React 现有编程模型的前提下最小化重渲染的开销。

最近，我们完成了对编译器的重写，使其更加可靠和强大。新的编译器架构使我们能够分析与记忆化比如 局部可变数据（local mutations） 这样的复杂代码，并打开了许多新的编译期优化的可能，而不仅仅局限于 Hooks 能做的优化。

我们还在开发一个用于探索这个编译器的 Playground。虽然 Playground 的首要目的是使编译器开发本身更加容易，但因为它能够揭示编译器背后的一些原理，并且可以实时渲染编译器优化后的效果，所以我们觉得它也会方便大家上手体验直观的感受编译优化的效果。Playground 将会随着编译器一同发布。

离屏（Offscreen）

当前，如果你想隐藏和显示一个组件，你有两个选择。第一种是从 UI 树中完全添加或删除它，但这种方法的问题在于每次卸载（unmount）时 UI 的状态都会丢失，包括存储在 DOM 中的状态，例如滚动条的位置。

而另一种选择是在保持组件装载（mount）的情况下用 CSS 切换视觉上的外观。这么做可以保留 UI 的状态，但也伴随着性能代价，因为 React 仍然必须在收到新更新时不断渲染这个即便已经隐藏起来组件，以及其所有的子组件。

Offscreen 引入了第三种选择：在视觉上隐藏 UI，但降低其内容渲染的优先级。这个想法在本质上类似于 content-visibility 这个 CSS 属性：当内容被隐藏时，它不需要与 UI 的其余部分保持同步。React 可以推迟渲染这个组件，直到应用程序没有其余工作需要做闲置下来，或者直到内容需要再次可见时，才去渲染它。

Offscreen 只是一个底层能力，他的目的是解锁更高层次的功能。就像 startTransition 等 React 的其他并发特性一样，你在大多数情况下都可能不会直接与 Offscreen API 打交道，而是直接使用元框架们通过 Offscreen 实现的高阶模式就好，诸如：

即时过渡（Instant transitions）： 一些路由框架已经可以通过诸如在 hover 链接时预加载数据等方式来对后续的跳转提速。通过 Offscreen，他们将可以直接在后台就预渲染下一个屏幕。

可重用状态（Reusable state）： 同样，在路由或者标签之间跳转时，你可以用 Offscreen 来保留前一个屏幕的状态，使得你可以在切换回来后从中断处继续。

虚拟化列表渲染（Virtualized list rendering）： 显示大型列表时，一些虚拟化列表的框架通常都会预渲染可见范围外的内容，你可以使用 Offscreen 来让预渲染不可见内容具备比渲染可见内容有更低的优先级。

背景内容（Backgrounded content）：这是另一个我们在探索的功能，比如在展示模态弹窗（model）时，可以在不隐藏背景的情况下降低对背景内容渲染的优先级。

过渡跟踪（Transition Tracing）

目前，React 有两个 Profiler 性能分析工具。最早的 Profiler 显示了一次分析会话中所有提交的概览。对于每次提交，它还显示所有渲染的组件以及渲染它们所花费的时间。我们还有一个在 React 18 中引入的 时间线（Timeline） Profiler 的 beta 版本，它显示组件何时安排更新以及 React 何时处理这些更新。这两个分析器都可以帮助开发人员识别代码中的性能问题。

我们已经意识到，开发人员并没有发现，其实了解单个缓慢的提交或脱离上下文的组件是很有用的。了解导致缓慢提交的真正原因会更有用。并且开发人员希望能够跟踪特定的交互（例如按钮单击、初始加载或页面导航）以观察性能回归并了解交互缓慢的原因以及如何去修复它。

我们之前尝试通过创建 交互跟踪 API 来解决这个问题，但它存在一些降低跟踪交互缓慢原因准确性的基本设计缺陷，有时会导致交互永无止境。由于这些问题，我们最终 移除了这个 API 。

我们正在开发一个新版本的交互跟踪 API（暂时称为过渡跟踪（Transition Tracing），因为它是通过 startTransition 来触发的）来解决这些问题。

React 新文档

去年，我们宣布了新的 React 文档网站的 beta 版本，新的学习材料从 Hooks 开始，并有新的图表、插图以及许多交互式代码示例和习题挑战。之前我们因为要专注于 React 18 的开发而暂停了这项工作，但现在既然 React 18 已经发布，我们就开始积极努力地完成与发布新文档了。

我们目前正在写一个关于副作用（effects）的详细章节，因为我们这听说对于新的和有经验的 React 用户来说都是相对具有挑战性的主题之一。与 Effects 同步 是该系列中的第一篇文章，接下来几周我们还会发布更多内容。当我们第一次开始编写有关 effects 的详细部分时，我们已经意识到可以通过向 React 添加新的原始 API 来简化许多常见的 effects 模式。useEvent RFC 中分享了一些初步想法。它目前处于早期研究阶段，我们仍在迭代这个想法。我们感谢社区迄今为止对 RFC 的评论，以及对正在进行的文档重写的 反馈 和贡献。我们要特别感谢 Harish Kumar 提交并审查了对新网站实施的诸多改进。

React 18 发布计划

June 08, 2021 by Andrew Clark, Brian Vaughn, Christine Abernathy, Dan Abramov, Rachel Nabors, Rick Hanlon, Sebastian Markbåge, and Seth Webster

2021 年 11 月 15 日更新

React 18 已进入 Beta 阶段。有关该版本的更多信息，可以在 React 18 工作组相关的文章中找到.

React 团队非常激动地与你分享一些最新的工作进展：

我们已经开始了 React 18 版本的发布工作，这将是我们的下一个主要版本。

我们创建了工作组，为社区逐步采用 React 18 的新特性做准备。

我们发布了 React 18 Alpha 版本，便于库作者尝试它并为我们提出相应反馈。

目前这些更新主要面向第三方库的维护者。如果你正在学习、教学或使用 React 来构建面向用户的应用程序，你可以忽略这篇博客。但如果你出于好奇，我们同样欢迎你关注 React 18 工作组中的讨论！

React 18 带来了什么

当 React 18 发布时，它将包含开箱即用的改进（如 automatic batching），全新的 API（如 startTransition）以及内置支持了 React.lazy 的 全新 SSR 架构。

这些功能之所以能够实现，要归功于我们在 React 18 中新加入的可选的 “并发渲染（concurrent rendering）” 机制。它使得 React 可以同时准备多个版本的 UI。这个机制主要发生在幕后，但它为 React 解锁了非常多新的可能性，来帮助你提高你应用程序的实际与感知性能。

如果你一直在关注我们对 React 未来的研究（我们并不期待你这么做！），你可能已经听说过 “并发模式（concurrent mode）”，或许还听过它可能会搞坏你的应用程序。为了回应社区对此的反馈，我们重新设计了可渐进的升级策略。相比于之前要么不升要么全升的一刀切方式，只有由新特性触发的更新会启用并发渲染。在实践中，这意味着 你无需重写代码即可直接使用 React 18，且可以根据自己的节奏和需要来尝试新特性。

循序渐进的采用策略

由于 React 18 中的并发性是可选功能，所以并不会立刻对组件行为带来任何明显的破坏性变化。你几乎不需要对应用程序中的代码进行任何改动就可以直接升级到 React 18，并不会比以往的 React 版本升级要困难。根据我们自己将几个应用程序升级到 React 18 的经验来看，预计许多用户能在一个下午的时间内完成升级工作。

我们在 Facebook 成功地将并发功能交付给了数以万计的组件，根据我们的经验来看，我们发现大多数 React 组件无需任何改动就能 “正常工作”。我们致力于确保整个社区都能平滑的升级，所以今天我们宣布了 React 18 工作组的成立。

与社区合作

我们正在这个版本中尝试一些新的可能：我们邀请了来自整个 React 社区的专家、开发者、库作者和教育者参与我们的 React 18 工作组，以提供反馈，提出问题，并就发布工作进行合作。我们无法邀请所有我们想邀请的人来参加这个最初的小团体，但如果实验成功，我们希望将来会有更多的人参与！

React 18 工作组的目标是为生态做好准备，使现有的应用程序和库能够顺利、逐步地采用 React 18。该工作组托管在 GitHub Discussions，以供公众阅读。工作组成员可以留下反馈，提

出问题，并分享想法。核心团队也将使用 repo 的讨论区来分享我们的研究成果。随着稳定版的发布越来越近，任何重要的信息我们将在博客上发布。

欲了解关于升级到 React 18 的更多信息，或关于该版本的其他资源，请参阅 React 18 公告.

访问 React 18 工作组

大家可以在 React 18 工作组仓库 中阅读相关讨论的情况。

我们预计对工作组感兴趣的小伙伴会激增，所以只有被邀请的成员可以创建或评论主题。然而，这些过程对公众是完全可见的，所以每个人都可以获得相同的信息，我们相信这是一个很好的折衷方案，既能为工作组成员创造一个富有成效的环境，又能保持对广大社区的透明度。

其他依旧，你可以在我们的 issue 中提交错误报告、问题和反馈。

如何尝试 React 18 Alpha

新的 alpha 版本通过 @alpha 标签定期发布到 npm 中。这些版本是由仓库的主分支的最新提交构建而来。当一个特性或 bug 修复被合并时，它将在下一个工作日出现在 alpha 版本中。

在 alpha 版本之间可能会有重大的变更或 API 变化。请谨记，alpha 版本不建议用于面向用户的生产应用中。

预计 React 18 的发布时间

我们没有安排具体的发布时间，但我们预计需要几个月的反馈和迭代时间，React 18 才能做好准备，以应用于大多数生产项目。

库的 Alpha 版本：今天可用

公开的 Beta 版：至少几个月

RC 版本：至少在 Beta 版发布后的几周

正式版：至少在 RC 版本发布之后的几周

关于发布时间表的更多细节，可以关注工作组。当临近公开发布时，我们会在这个博客上发布更新。

React v17.0

October 20, 2020 by Dan Abramov and Rachel Nabors

今天，我们宣布 React 17 正式发布！在此之前，我们在 React 17 RC 的博文中已经介绍了 React 17 发布的意义以及包含的变化。此文是针对那篇文章的简单总结，如果你已阅读过那篇博文，此文可略过。

无新特性

React v17 的发布非比寻常，因为它没有增加任何面向开发者的新特性。

但是，**这个版本会使得 React 自身的升级变得更加容易**。

值得特别说明地是，React v17 作为后续版本的 ”基石“，它让不同版本的 React 相互嵌套变得更加容易。

除此之外，还会使 React 更容易嵌入到由其他技术构建的应用中。

渐进式升级

React v17 开启了 React 渐进式升级的新篇章。当你从 React 15 升级至 16 时（或者，从 16 升级到 17），你通常会一次性升级整个应用程序，这对大部分应用来说十分有效。但是，如果代码库编写于几年前，并且

没有及时的维护升级，这会使得升级成本越来越高。并且，在 React 17 之前，如果在同一个页面上使用不同的 React 版本（可以这么做，但是有风险），会导致事件问题的出现，会有一些未知的风险。

我们正在修复 React v17 中的许多问题。这意味着，**当 React 18 或未来版本来临时，你将有更多选择**。首选，当然还是一次性升级整个应用；但你还有个可选方案，渐进式升级你的应用。举个例子，你可能将大部分功能升级至 React v18，但保留部分懒加载的对话框或子路由在 React v17。

但这并**不意味着**你必须进行渐进式升级。**对于大多数应用来说，一次性升级仍是更好的选择**。加载两个版本的 React，仍然不是理想方案 —— 即使其中一个版本是按需加载的。但对于那些长期未维护的大型应用来说，这意义非凡，React v17 开始让这些应用不会被轻易淘汰。

我们准备了示例仓库，此示例演示了如何在必要时懒加载旧版本的 React。此示例由 Create React App 构建，使用其他工具也可以实现同样的效果。欢迎使用其他工具的小伙伴通过 PR 的形式提供 Demo。

注意

我们**将其他特性**推迟到了 React v17 之后。这个版本的目标就是实现渐进式升级。如果升级到 17 很困难，那就违背了此版本的目的。

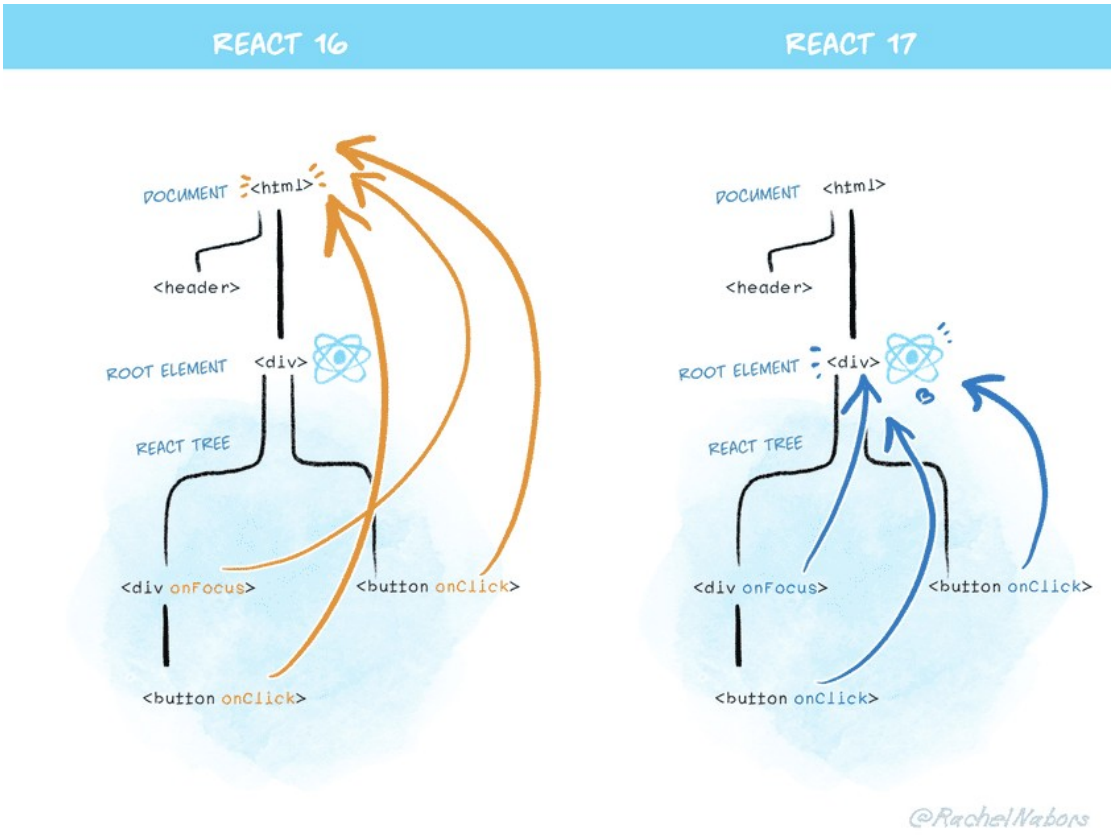
事件委托的变更

为了实现渐进式升级，我们需要对 React 的事件系统进行修改。React 17 是一个重要版本，因为这个版本的可能存在破坏性更改。关于版本的更多信息，请查阅版本的 FAQ，以了解我们对版本稳定性的承诺。

React v17 中，React 不会再将事件处理添加到 `document` 上，而是将事件处理添加到渲染 React 树的根 DOM 容器中：

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

在 React 16 及之前版本中，React 会对大多数事件进行 `document.addEventListener()` 操作。React v17 开始会通过调用 `rootNode.addEventListener()` 来代替。



经核实，多年来在 issue 追踪器 上报告的许多问题都已被新特性解决，其中大多与将 React 与非 React 代码集成有关。

如果你在升级时遇到了这方面的问题，可以看看这个常见的解决方案。

其他破坏性更改

React v17 的 RC 博文描述了关于 React v17 中其他的破坏性更改。

我们在升级 Facebook 项目代码中 10w+ 组件的过程中，只修改了不到 20 个组件，所以**我们猜测大多数应用在升级 v17 时，不会有太大的问题**。

如果你遇到任何问题，请告诉我们。

全新的 JSX 转换

React v17 支持了全新的 JSX 转换。我们还针对 React 16.14.0，React 15.7.0 和 0.14.0 版本做了兼容。请注意，此功能完全可选，并非必须使用。之前的 JSX 转换将会继续维护，并且没有停止支持它的计划。

React Native

React Native 会有一个单独的发布计划。目前，我们对 React v17 的支持已在 React Native 0.64 中落地。你可以在 React Native 社区的发布 issue tracker 上参与讨论。

安装

使用 npm 安装 React v17：

```
npm install react@17.0.0 react-dom@17.0.0
```

使用 yarn 安装 React v17：

```
yarn add react@17.0.0 react-dom@17.0.0
```

我们还提供了由 UMD 构建的 CDN 版本：

```
<script crossorigin
src="https://unpkg.com/react@17.0.0/umd/react.production.min.js"></
script>
<script crossorigin src="https://unpkg.com/react-dom@17.0.0/umd/react-
dom.production.min.js"></script>
```

请参阅文档中的详细安装说明。

变更日志

React

- 为全新的 JSX 转换器添加 `react/jsx-runtime` 和 `react/jsx-dev-runtime`。（@lunaruan 提交于 #18299）
- 根据原生框架构建组件调用栈。（@sebmarkbage 提交于 #18561）
- 可以在 context 中设置 `displayName` 以改善调用栈信息。（@eps1lon 提交于 #18224）

- 防止 `'use strict'` 从 UMD 的 bundles 中泄露。（@koba04 提交于 #19614）
- 停止使用 `fb.me` 进行重定向。（@cylim 提交于 #19598）

React DOM

- 将事件委托从 `document` 切换为 root。（@trueadm 提交于 #18195 及其他）
- 在运行下一个副作用前，请清理所有副作用。（@bvaughn 提交于 #17947）
- 异步运行 `useEffect` 清理函数。（@bvaughn 提交于 #17925）
- 使用浏览器的 `focusin` 和 `focusout` 替换 `onFocus` 和 `onBlur` 的底层实现。（@trueadm 提交于 #19186）
- 将所有 `Capture` 事件都使用浏览器的捕获阶段实现。（@trueadm 提交于 #19221）
- 禁止在 `onScroll` 事件时冒泡。（@gaearon 提交于 #19464)
- 如果 `forwardRef` 或 `memo` 组件的返回值为 `undefined`，则抛出异常。（@gaearon 提交于 #19550）
- 移除事件池。（@trueadm 提交于 #18969）
- 移除 React Native Web 不需要的内部组件。（@necolas 提交于 #18483)
- 当挂载 root 时，附加所有已知的事件监听器。（@gaearon 提交于 #19659）
- 在 Dev 模式下，禁用第二次渲染过程中的 `console`。（@sebmarkbage 提交于 #18547）
- 弃用为记录且具有误导性的 `ReactTestUtils.SimulateNative` API。（@gaearon 提交于 #13407)
- 重命名内部使用的私有字段（@gaearon 提交于 #18377）
- 不在开发环境调用 User Timing API。（@gaearon 提交于 #18417）
- 在严格模式下重复渲染期间禁用 console。（@sebmarkbage 提交于 #18547）
- 在严格模式下，二次渲染组件也不使用 Hook。（@eps1lon 提交于 #18430）
- 允许在生命周期函数中调用 `ReactDOM.flushSync`（但会发出警告）。（@sebmarkbage 提交于 #18759）
- 将 `code` 属性添加到键盘事件对象中。（@bl00mber 提交于 #18287）
- 为 `video` 元素添加 `disableRemotePlayback` 属性。（@tombrowndev 提交于 #18619）
- 为 `input` 元素添加 `enterKeyHint` 属性。（@eps1lon 提交于 #18634）
- 当没有给 `<Context.Provider>` 提供任何值时，会发出警告。（@charlie1404 提交于 #19054）
- 如果 `forwardRef` 或 `memo` 组件的返回值为 `undefined`，则抛出警告。（@bvaughn 提交于 #19550）
- 为无效更新改进错误信息。（@JoviDeCroock 提交于 #18316）
- 从调用栈信息中忽略 forwardRef 和 memo。（@sebmarkbage 提交于 #18559）
- 在受控输入与非受控输入间切换时，改善错误消息。（@vcarl 提交于 #17070）
- 保持 `onTouchStart`、`onTouchMove` 和 `onWheel` 默认为 passive。（@gaearon 提交于 #19654）
- 修复在 development 模式下 iframe 关闭时，`setState` 挂起的问题。（@gaearon 提交于 #19220）
- 使用 `defaultProps` 修复拉架子组件在渲染时的问题。（@jddxf 提交于 #18539）

- 修复当 `dangerouslySetInnerHTML` 为 `undefined` 时，误报警告的问题。（@eps1lon 提交于 #18676）
- 使用非标准的 `require` 实现来修复 Test Utils。（@just-boris 提交于 #18632）
- 修复 `onBeforeInput` 报告错误的 `event.type`。（@eps1lon 提交于 #19561）
- 修复 Firefox 中 `event.relatedTarget` 输出为 `undefined` 的问题。（@claytercek 提交于 #19607）
- 修复 IE11 中 “unspecified error” 的问题。（@hemakshis 提交于 #19664）
- 修复 shadow root 中的渲染问题。（@Jack-Works 提交于 #15894）
- 使用事件捕获修复 `movementX/Y` polyfill 的问题。（@gaearon 提交于 #19672）
- 使用委托处理 `onSubmit` 和 `onReset` 事件。（@gaearon 提交于 #19333）
- 提高内存使用率。（@trueadm 提交于 #18970）

React DOM Server

- 使用服务端渲染的 `useCallback` 与 `useMemo` 一致。（@alexmckenley 提交于 #18783）
- 修复函数组件抛出异常时状态泄露的问题。（@pmaccart 提交于 #19212）

React Test Renderer

- 改善 `findByType` 错误信息。（@henryqdineen 提交于 #17439）

Concurrent Mode（实验阶段）

- 改进启发式更新算法。（@acdlite 提交于 #18796）
- 在实验性 API 前添加 `unstable_` 前缀。(@acdlite 提交于 #18825)
- 移除 `unstable_discreteUpdates` 和 `unstable_flushDiscreteUpdates`。（@trueadm 提交于 #18825）
- 移除了 `timeoutMs` 参数。（@acdlite 提交于 #19703）
- 禁用 `<div hidden />` 预渲染，以支持未来的 API。（@acdlite 提交于 #18917）
- 为 Suspense 添加了 `unstable_expectedLoadTime`，用于 CPU-bound 树。（@acdlite 提交于 #19936）
- 添加了一个实验性的 `unstable_useOpaqueIdentifier` Hook。（@lunaruan 提交于 #17322）
- 添加了一个实验性的 `unstable_startTransition` API. (@rickhanlonii 提交于 #19696)
- 在测试渲染器中使用 `act` 后，不在刷新 Suspense 的 fallback。（@acdlite 提交于 #18596）
- 将全局渲染的 timeout 用于 CPU Suspense。（@sebmarkbage 提交于 #19643）
- 挂载前，清除现有根目录的内容。（@bvaughn 提交于 #18730）
- 修复带有错误边界的 bug。（@acdlite 提交于 #18265）
- 修复了导致挂起树更新丢失的 bug。（@acdlite 提交于 #18384 以及 #18457）
- 修复导致渲染阶段更新丢失的 bug。（@acdlite 提交于 #18537）
- 修复 SuspenseList 的 bug。（@sebmarkbage 提交于 #18412）
- 修复导致 Suspense fallback 过早显示的 bug。（@acdlite 提交于 #18411）

- 修复 SuspenseList 中使用 class 组件异常的 bug。（@sebmarkbage 提交于 #18448）
- 修复输入内容可能被更新被丢弃的 bug。（@jddxf 提交于 #18515 以及 @acdlite 提交于 #18535）
- 修复暂挂 Suspense fallback 后卡住的错误。（@acdlite 提交于 #18663）
- 如果 hydrate 中，不要切断 SuspenseList 的尾部。（@sebmarkbage 提交于 #18854）
- 修复 `useMutableSource` 中的 bug，此 bug 可能在 `getSnapshot` 更改时出现。（@bvaughn 提交于 #18297）
- 修复 `useMutableSource` 令人恶心的 bug。（@bvaughn 提交于 #18912）
- 如果外部渲染且提交之前调用 `setState`，会发出警告。（@sebmarkbage 提交于 #18838）

介绍全新的 JSX 转换

September 22, 2020 by Luna Ruan

虽然 React 17 并未包含新特性，但它将提供一个全新版本的 JSX 转换。本文中，我们将为你描述它是什么以及如何使用。

何为 JSX 转换？

在浏览器中无法直接使用 JSX，所以大多数 React 开发者需依靠 Babel 或 TypeScript 来**将 JSX 代码转换为 JavaScript**。许多包含预配置的工具，例如 Create React App 或 Next.js，在其内部也引入了 JSX 转换。React 17 发布在即，尽管我们想对 JSX 的转换进行改进，但我们不想打破现有的配置。于是我们选择与 Babel 合作 ，为想要升级的开发者**提供了一个全新的，重构过的 JSX 转换的版本**。

升级至全新的转换完全是可选的，但升级它会为你带来一些好处：

- 使用全新的转换，你可以**单独使用 JSX 而无需引入 React**。
- 根据你的配置，JSX 的编译输出可能会**略微改善 bundle 的大小**。
- 它将**减少你需要学习 React 概念的数量**，以备未来之需。

此次升级不会改变 JSX 语法，也并非必须。旧的 JSX 转换将继续工作，没有计划取消对它的支持。

React 17 的 RC 版本 已经引入了对新转换的支持，所以你可以尝试一下！为了让大家更容易使用，在 React 17 正式发布后，我们还将此支持移植到了 React 16.14.0，React 15.7.0 以及 React 0.14.10。你可以在下方找到不同工具的升级说明。

接下来，我们来仔细对比新旧转换的区别。

新的转换有何不同？

当你使用 JSX 时，编译器会将其转换为浏览器可以理解的 React 函数调用。**旧的 JSX 转换**会把 JSX 转换为 `React.createElement(...)` 调用。

例如，假设源代码如下：

```
import React from 'react';

function App() {
  return <h1>Hello World</h1>;
}
```

旧的 JSX 转换会将上述代码变成普通的 JavaScript 代码：

```
import React from 'react';

function App() {
  return React.createElement('h1', null, 'Hello world');
}
```

注意

无需改变源码。我们将介绍 JSX 转换如何将你的 JSX 源码变成浏览器可以理解的 JavaScript 代码。

然而，这并不完美：

- 如果使用 JSX，则需在 `React` 的环境下，因为 JSX 将被编译成 `React.createElement`。

- 有一些 `React.createElement` 无法做到的性能优化和简化。

为了解决这些问题，React 17 在 React 的 package 中引入了两个新入口，这些入口只会被 Babel 和 TypeScript 等编译器使用。新的 JSX 转换**不会将 JSX 转换为 `React.createElement`**，而是自动从 React 的 package 中引入新的入口函数并调用。

假设你的源代码如下：

```
function App() {
  return <h1>Hello World</h1>;
}
```

下方是新 JSX 被转换编译后的结果：

```
// 由编译器引入（禁止自己引入！）
import {jsx as _jsx} from 'react/jsx-runtime';

function App() {
  return _jsx('h1', { children: 'Hello world' });
}
```

注意，此时源代码**无需引入 React** 即可使用 JSX 了！（但仍需引入 React，以便使用 React 提供的 Hook 或其他导出。）

此变化与所有现有 JSX 代码兼容，所以你无需修改组件。如果你对此感兴趣，你可以查看 RFC 了解全新转换工作的具体细节。

注意

`react/jsx-runtime` 和 `react/jsx-dev-runtime` 中的函数只能由编译器转换使用。如果你需要在代码中手动创建元素，你可以继续使用 `React.createElement`。它将继续工作，不会消失。

如何升级至新的 JSX 转换

如果你还没准备好升级为全新的 JSX 转换，或者你正在为其他库使用 JSX，请不要担心，旧的转换不会被移除，并将继续支持。

如果你想升级，你需要准备两件事：

- **支持新转换的 React 版本**（React 17 的 RC 版本 及更高版本支持它，但是我们也发布了 React 16.14.0，React 15.7.0 以及 0.14.10 等主要版本，以供还在使用旧版本的开发者使用它们）
- **一个兼容新转换的编译器**（请看下面关于不同工具的说明）。

由于新的 JSX 转换不依赖 React 环境，我们准备了一个自动脚本，用于移除你代码中不必要的引入。

Create React App

Create React App 4.0.0+ 使用了兼容 React 版本的 JSX 转换。

Next.js

Next.js 的 v9.5.3+ 会使用新的转换来兼容 React 版本。

Gatsby

Gatsby 的 v2.24.5+ 会使用新的转换来兼容 React 版本。

注意

如果你在 Gatsby 中遇到 error，请升级至 React 17 的 RC 版本，运行 `npm update` 解决此问题。

手动设置 Babel

Babel 的 v7.9.0 及以上版本可支持全新的 JSX 转换。

首先，你需要更新至最新版本的 Babel 和 transform 插件。

如果你使用的是 `@babel/plugin-transform-react-jsx`：

```
# npm 用户
npm update @babel/core @babel/plugin-transform-react-jsx
# yarn 用户
yarn upgrade @babel/core @babel/plugin-transform-react-jsx
```

如果你使用的是 `@babel/preset-react`：

```
# npm 用户
npm update @babel/core @babel/preset-react
# yarn 用户
yarn upgrade @babel/core @babel/preset-react
```

目前，旧的转换的默认选项为 `{"runtime": "classic"}`。如需启用新的转换，你可以使用 `{"runtime": "automatic"}` 作为 `@babel/plugin-transform-react-jsx` 或 `@babel/preset-react` 的选项：

```
// 如果你使用的是 @babel/preset-react
{
  "presets": [
    ["@babel/preset-react", {
      "runtime": "automatic"
    }]
  ]
}
// 如果你使用的是 @babel/plugin-transform-react-jsx
{
  "plugins": [
    ["@babel/plugin-transform-react-jsx", {
      "runtime": "automatic"
    }]
  ]
}
```

从 Babel 8 开始，`"automatic"` 会将两个插件默认集成在 rumtime 中。欲了解更多信息，请查阅 Babel 文档中的 @babel/plugin-transform-react-jsx 以及 @babel/preset-react。

注意

如果你在使用 JSX 时，使用 React 以外的库，你可以使用 `importSource` 选项从该库中引入 — 前提是它提供了必要的入口。或者你可以继续使用经典的转换，它会继续被支持。

如果你是库的作者并且需要为你的库实现 `/jsx-runtime` 的入口，需注意一种情况，在此情况下，为了向下兼容，即使使用了新的 jsx 转换，也必须考虑 `createElement`。在上述情况中，将直接从 `importSource` 的根入口中自动引入 `createElement`。

ESLint

如果你正在使用 eslint-plugin-react，其中的 `react/jsx-uses-react` 和 `react/react-in-jsx-scope` 规则将不再需要，可以关闭它们或者删除。

```
{
  // ...
  "rules": {
    // ...
    "react/jsx-uses-react": "off",
    "react/react-in-jsx-scope": "off"
  }
}
```

TypeScript

TypeScript 将在 v4.1 及以上版本中支持新的 JSX 转换。

Flow

Flow 将在 v0.126.0 中支持新的 JSX 转换。

移除未使用的 React 引入

因为新的 JSX 转换会自动引入必要的 `react/jsx-runtime` 函数，因此当你使用 JSX 时，将无需再引入 React。将可能会导致你代码中有未使用到的 React 引入。保留它们也无伤大雅，但如果你想删除它们，我们建议运行 “codemod” 脚本来自动删除它们：

```
cd your_project
npx react-codemod update-react-imports
```

注意：

如果你在运行 codemod 时出现错误，请尝试使用 `npx react-codemod update-react-imports` 选择不同的 JavaScript 环境。尤其是选择“JavaScript with Flow”时，即使你未使用 Flow，也可以选择它，因为它比 JavaScript 支持更新的语法。如果遇到问题，请告知我们。

请注意，codemod 的输出可能与你的代码风格并不匹配，因此你可能需要再 codemod 完成后运行 Prettier 以保证格式一致。

运行 codemod 会执行如下操作：

- 升级到新的 JSX 转换，删除所有未使用的 React 引入。
- 所有 React 的默认引入将被改为解构命名引入（例如，`import React from "react"` 会变成 `import { useState } from "react"`），这将成为未来开发的首选风格。codemod **不会** 影响现有的命名空间引入方式（即 `import * as React from "react"`），这也是一种有效的风格。默认的引入将在 React 17 中继续工作，但从长远来看，我们建议尽量不使用它们。

示例：

```
import React from 'react';

function App() {
  return <h1>Hello World</h1>;
}
```

将被替换为

```
function App() {
  return <h1>Hello World</h1>;
}
```

如果你使用了 React 的其他导出 — 比如 Hook，那么 codemod 将把它们转换为具名导入。

示例：

```
import React from 'react';
```

```
function App() {
  const [text, setText] = React.useState('Hello World');
  return <h1>{text}</h1>;
}
```

会被替换为

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

function App() {
  const [text, setText] = useState('Hello World');
  return <h1>{text}</h1>;
}
```

除了清理未使用的引入外，此工具还可帮你为未来 React 主要版本（不是 React 17 版本）做铺垫，该版本将支持 ES 模块，并且没有默认导出。

鸣谢

我们要感谢 Babel，TypeScript，Create React App，Next.js，Gatsby，ESLint 以及 Flow 的主要维护者为新 JSX 转换提供的实现和整合。我们还要感谢 React 社区对相关 RFC 提供的反馈和讨论。

React v17.0 RC 版本发布：没有新特性

August 10, 2020 by Dan Abramov and Rachel Nabors

今天，我们发布了 React 17 的第一个 RC 版本。自 React 上一个主要版本至今已经两年半了，按照我们的标准，时间跨度有些长了！在

此篇博客中，我们将讲解此次主要版本的职责，对你的影响以及如何试用此版本。

无新特性

React 17 的版本是非比寻常的，因为它没有添加任何面向开发人员的新功能。而主要侧重于**升级简化 React 本身**。

我们正在积极开发 React 的新功能，但它们并不属于此版本。React 17 是我们进行深度推广战略的关键所在。

此版本之所以特殊，你可以认为**React 17 是 “垫脚石” 版本**，它会使得由一个 React 版本管理的 tree 嵌入到另一个 React 版本管理的 tree 中时会更加安全。

逐步升级

在过去 7 年里，React 一直遵循 “all-or-nothing” 的升级策略。你可以继续使用旧版本，也可以将整个应用程序升级至新版本。但没有介于两者之间的情况。

此方式持续至今，但是我们遇到了 “all-or-nothing” 升级策略的局限性。

许多 API 的变更，例如，弃用旧版 context API 时，并不能以自动化的方式来完成。至今可能大多数应用程序从未使用过它们，但我们仍然选择在 React 中支持它们。我们必须在无限期支持过时的 API 或针对某些应用仍使用旧版本 React 间进行选择。但这两个方案都不合适。

因此，我们想提供另一种方案。

React 17 开始支持逐步升级 React 版本。当从 React 15 升级至 16 时（或者从 React 16 升级至 17 时），通常会一次升级整个应用程序。这适用于大部分应用程序。但是，如果代码库是在几年前编写的，并且并

没有得到很好的维护，那么升级它会变得越来越有挑战性。尽管可以在页面上使用两个版本的 React，但是直到 React 17 依旧有事件问题出现。

我们使用 React 17 解决了许多诸如此类的问题。这将意味着**当 React 18 或未来版本问世时，你将有更多选择**。首选还是像以前一样，一次升级整个应用程序。但你也可以选择逐步升级你的应用程序。例如，你可能会将大部分应用程序迁移至 React 18，但在 React 17 上保留一些延迟加载的对话框或子路由。

但这不意味着你*必须*逐步升级。对于大部分应用程序来说，一次全量升级仍是最好的解决方案。加载两个 React 版本，即使其中一个是按需延迟加载的，仍然不太理想。但是，对于没有积极维护的大型应用来说，可以考虑此种方案，并且 React 17 开始可以保证这些应用程序不落伍。

为了实现逐步升级，我们需要对 React 事件系统进行一些更改。而这些更改可能会对代码产生影响，这也是 React 17 成为主要版本的原因。实际上，10 万个以上的组件中受影响的组件不超过 20 个，因此，**我们希望大多数应用程序都可以升级到 React 17，而不会产生太多影响**。如果你遇到问题，请联系我们。

逐步升级 Demo

我们准备了一个示例 repo，展示了如何在必要时延迟加载旧版本的 React。此 demo 使用了 Create React App 进行构建，但对其他工具采用类似的方法应该也适用。我们欢迎使用其他工具的开发者编写 demo 并提交 PR。

注意

我们已将**其他更改推迟**到 React 17 之后。此版本的目标是实现逐步升级。如果升级 React 17 太困难，则此目标会无法实现。

更改事件委托

从技术上讲，始终可以在应用程序中嵌套不同版本的 React。但是，由于 React 事件系统的工作原理，这很难实现。

在 React 组件中，通常会内联编写事件处理：

```
<button onClick={handleClick}>
```

与此代码等效的原生 DOM 操作如下：

```
myButton.addEventListener('click', handleClick);
```

但是，对大多数事件来说，React 实际上并不会将它们附加到 DOM 节点上。相反，React 会直接在 `document` 节点上为每种事件类型附加一个处理器。这被称为事件委托。除了在大型应用程序上具有性能优势外，它还使添加类似于 replaying events 这样的新特性变得更加容易。

自从其发布以来，React 一直自动进行事件委托。当 document 上触发 DOM 事件时，React 会找出调用的组件，然后 React 事件会在组件中向上 “冒泡”。但实际上，原生事件已经冒泡出了 `document` 级别，React 在其中安装了事件处理器。

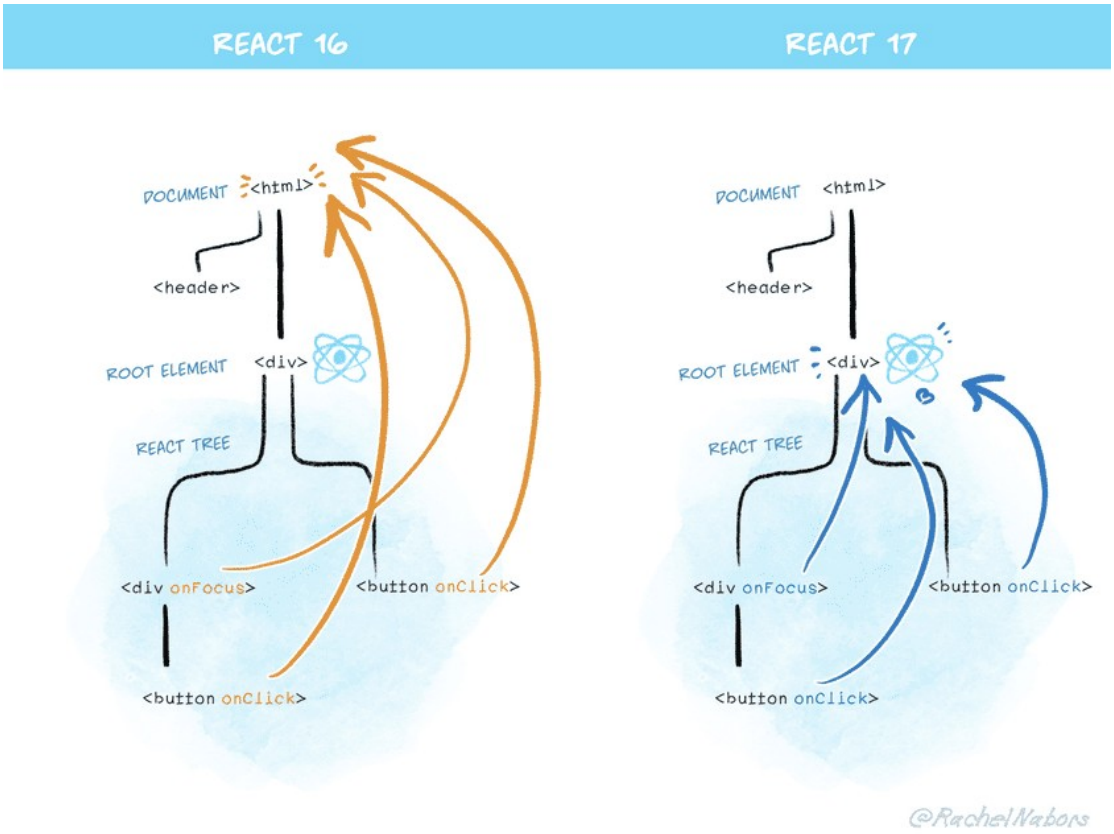
但是，这就是逐步升级的困难所在。
如果页面上有多个 React 版本，他们都将在顶层注册事件处理器。这会破坏 `e.stopPropagation()`：如果嵌套树结构中阻止了事件冒泡，但外部树依然能接收到它。这会使不同版本 React 嵌套变得困难重重。这种担忧并不是没有根据的 —— 例如，四年前 Atom 编辑器就遇到了相同的问题。

这也是我们为什么要改变 React 底层附加事件方式的原因。

在 **React 17** 中，**React** 将不再向 `document` 附加事件处理器。而会将事件处理器附加到渲染 **React** 树的根 **DOM** 容器中：

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

在 React 16 或更早版本中，React 会对大多数事件执行 `document.addEventListener()`。React 17 将会在底层调用 `rootNode.addEventListener()`。



由于此更改，**现在可以更加安全地进行新旧版本 React 树的嵌套**。请注意，要使其正常工作，两个版本都必须为 17 或更高版本，这就是为什么强烈建议升级到 React 17 的根本原因。从某种意义上讲，React 17 是一个 “垫脚石” 版本，使逐步升级成为可能。

此更改还使得**将 React 嵌入使用其他技术构建的应用程序变得更加容易**。例如，如果应用程序的“外壳”是用 jQuery 编写的，但其中较新的代码

是用 React 编写的，则 React 代码中的 `e.stopPropagation()` 会阻止它影响 jQuery 的代码 —— 这符合预期。换个角度来说，如果你不再喜欢 React 并想重写应用程序（比如，用 jQuery），则可以从外壳开始将 React 转换为 jQuery，而不会破坏事件冒泡。

经核实，多年来在 issue 追踪器 上报告的许多问题都已被新特性解决，这些问题大多都与将 React 与非 React 代码集成有关。

注意

你可能想知道这是否会破坏根 DOM 容器之外的 Portals。答案是 React 还会监听 portals 容器上的事件，所以这不是问题。

解决隐患

与其他重大更改一样，可能需要对代码进行调整。在 Facebook，我们在成千上万个模块中，大约调整了 10 个模块以适应此更改。

例如，如果模块中使用 `document.addEventListener(...)` 手动添加了 DOM 监听，你可能希望能捕获到所有 React 事件。在 React 16 或更早版本中，即使你在 React 事件处理器中调用 `e.stopPropagation()`，你创建的 DOM 监听仍会触发，这是因为原生事件已经处于 document 级别。使用 React 17 冒泡将被阻止（按需），因此你的 `document` 级别的事件监听不会触发：

```
document.addEventListener('click', function() {
  // This custom handler will no longer receive clicks
  // from React components that called e.stopPropagation()
});
```

你可以将监听转换为使用捕获来修复此类代码。为此，你可以将 `{ capture: true }` 作为 `document.addEventListener` 的第三个参数传递：

```
document.addEventListener('click', function() {
  // Now this event handler uses the capture phase,
  // so it receives *all* click events below!
}, { capture: true });
```

请注意，此策略在全局上具有更好的适应性。例如，它可能会修复代码中现有的错误，这些错误在 React 事件处理器外部调用 `e.stopPropagation()` 发生。换句话说，**React 17 的事件冒泡更接近常规 DOM**。

其他重大更改

我们将 React 17 中的重大更改保持在最低水平。例如，它不会删除以前版本中弃用的任务方法。但是，它的确包含一些其他重大更改，根据经验，这些更改会相对安全。总体而言，由于这些因素的存在，在 10 万个以上的组件中受影响的组件不超过 20 个。

对标浏览器

我们对事件系统进行了一些较小的更改：

- `onScroll` 事件**不再冒泡**，以防止出现常见的混淆。
- React 的 `onFocus` 和 `onBlur` 事件已在底层切换为原生的 `focusin` 和 `focusout` 事件。它们更接近 React 现有行为，有时还会提供额外的信息。
- 捕获事件（例如，`onClickCapture`）现在使用的是实际浏览器中的捕获监听器。

这些更改会使 React 与浏览器行为更接近，并提高了互操作性。

注意：

尽管 React 17 **底层**已将 `onFocus` 事件从 `focus` 切换为 `focusin`，但请注意，这并未影响冒泡行为。在 React 中，`onFocus` 事件总是冒泡的，在 React 17 中会继续保持，因为通常它是一个更有用的默认值。请参阅 sandbox，以了解为不同特定用例添加不同检查。

去除事件池

React 17 中移除了 “event pooling（事件池）“。它并不会提高现代浏览器的性能，甚至还会使经验丰富的开发者一头雾水：

```
function handleChange(e) {
  setData(data => ({
    ...data,
    // This crashes in React 16 and earlier:
    text: e.target.value
  }));
}
```

这是因为 React 在旧浏览器中重用了不同事件的事件对象，以提高性能，并将所有事件字段在它们之前设置为 `null`。在 React 16 及更早版本中，使用者必须调用 `e.persist()` 才能正确的使用该事件，或者正确读取需要的属性。

在 React 17 中，此代码可以按照预期效果执行。旧的事件池优化操作已被完成删除，因此，使用者可以在需要时读取事件字段。

这改变了行为，因此我们将其标记为重大更改，但在实践中我们没有看到它在 Facebook 上造成影响。（甚至还修复了一些错误！）请注意，`e.persist()` 在 React 事件对象中仍然可用，只是无效果罢了。

副作用清理时间

我们将使 `useEffect` 和清理函数的时间保持一致。

```
useEffect(() => {
  // This is the effect itself.
  return () => {    // This is its cleanup.  };});
```

大多数副作用（effect）不需要延迟屏幕更新，因此 React 在屏幕上反映出更新后立即异步执行它们。（在极少数情况下，你需要一种副作用来阻止绘制，例如，如果需要获取尺寸和位置，请使用 `useLayoutEffect`。）

然而，当组件被卸载时，副作用*清理*函数（类似于在 class 组件中同步调用 `componentWillUnmount`）同步运行。我们发现，对于大型应用程序来说，这不是理想选择，因为同步会减缓屏幕的过渡（例如，切换标签）。

在 React 17 中，副作用清理函数总会异步执行 —— 如果要卸载组件，则清理会在屏幕更新*后*运行。
这反映了副作用本身如何更紧密地运行。在极少数情况下，你可能希望依靠同步执行，可以改用 `useLayoutEffect`。

注意

你可能想知道这是否意味着你现在将无法修复有关未挂载组件上的 `setState` 的警告。不必担心，React 专门处理了这种情况，并且不会在卸载和清理之间短暂间隔内发出 `setState` 的警告。**因此，取消代码的请求或间隔几乎总是可以保存不变的。**

此外，React 17 将在运行任何新副作用之前执行所有副作用的清理函数（针对所有组件）。React 16 只对组件内的 effect 保证这种顺序。

隐患

可复用的库可能需要对此情况进行深度测试，但我们只遇到了几个组件会因为这次改变出现问题。有问题的代码的其中一个示例如下所示：

```
useEffect(() => {
  someRef.current.someSetupMethod();
  return () => {
    someRef.current.someCleanupMethod();
  };
});
```

问题在于 `someRef.current` 是可变的，因此在运行清除函数时，它可能已经设置为 `null`。解决方案是在副作用**内部**存储会发生变化的值：

```
useEffect(() => {
  const instance = someRef.current;
  instance.someSetupMethod();
```

```
  return () => {
    instance.someCleanupMethod();
  };
});
```

我们不希望此问题对大家造成影响，我们提供了 eslint-plugin-react-hooks/exhaustive-deps 的 lint 规则（请确保在项目中使用它）会对此情况发出警告。

返回一致的 undefined 错误

在 React 16 及更早版本中，返回 undefined 始终是一个错误：

```
function Button() {
  return; // Error: Nothing was returned from render
}
```

部分原因是这很容易无意间返回 undefined：

```
function Button() {
  // We forgot to write return, so this component returns undefined.
  // React surfaces this as an error instead of ignoring it.
  <button />;
}
```

以前，React 只对 class 和函数组件执行此操作，但并不会检查 forwardRef 和 memo 组件的返回值。这是由于编码错误导致。

在 React 17 中，forwardRef 和 memo 组件的行为会与常规函数组件和 class 组件保持一致。在返回 undefined 时会报错

```
let Button = forwardRef(() => {
  // We forgot to write return, so this component returns undefined.
  // React 17 surfaces this as an error instead of ignoring it.
  <button />;
});

let Button = memo(() => {
  // We forgot to write return, so this component returns undefined.
  // React 17 surfaces this as an error instead of ignoring it.
  <button />;
});
```

对于不想进行任何渲染的情况，请返回 null。

原生组件栈

当你在浏览器中遇到错误时，浏览器会为你提供带有 JavaScript 函数的名称及位置的调用栈信息。然而，JavaScript 调用栈通常不足以诊断问题，因为 React 树的层次结构可能同样重要。你不仅要知道哪个 `Button` 抛出了错误，而且还想知道 `Button` 在 React 树中的哪个位置。

为了解决这个问题，当你遇到错误时，从 React 16 开始会打印“组件栈”信息。尽管如此，它们仍然不如原生的 JavaScript 调用栈。特别是，它们在控制台中不可点击，因为 React 不知道函数在源代码中的声明位置。此外，它们在生产中几乎无用。不同于常规压缩后的 JavaScript 调用栈，它们可以通过 sourcemap 的形式自动恢复到原始函数的位置，而使用 React 组件栈，在生产环境下必须在调用栈信息和 bundle 大小间进行选择。

在 React 17 中，使用了不同的机制生成组件调用栈，该机制会将它们与常规的原生 JavaScript 调用栈缝合在一起。这使得你可以在生产环境中获得完全符号化的 React 组件调用栈信息。

React 实现这一点的方式有点非常规。目前，浏览器无法提供获取函数调用栈框架（源文件和位置）的方法。因此，当 React 捕获到错误时，将通过组件上述组件内部抛出的临时错误（并捕获）来重建其组件调用栈信息。这会增加崩溃时的性能损失，但每个组件类型只会发生一次。

如果你对此感兴趣，可以在这个 PR 中阅读更多详细信息，但是在大多数情况下，这种机制不会影响你的代码。从使用者的角度来看，新功能就是可以单击组件调用栈（因为它们依赖于本机浏览器调用栈框架），并且可以像常规 JavaScript 错误那样在生产中进行解码。

构成重大变化的部分是，要使此功能正常工作，React 将在捕获错误后在调用栈中重新执行上面某些函数和某些 class 构造函数的部分。由于渲染函数和 class 构造函数不应具有副作用（这对于 SSR 也很重要），因此这不会造成任何实际问题。

移除私有导出

最后，值得注意的重大变化时我们删除了一些以前暴露给其他项目的 React 内部组件。特别是，React Native for Web 过去常常依赖于事件系统的某些内部组件，但这种依赖关系很脆弱且经常被破坏。

在 React 17 中，这些私有导出已被移除。据我们所知，React Native for Web 是唯一使用它们的项目，它们已经完成了向不依赖那些私有导出函数的其他方法迁移。

这意味着旧版本的 React Native for Web 不会与 React 17 兼容，但是新版本可以使用它。实际上，并没有太大的变化，因为 React Native for Web 必须发布新版本以适应其内部 React 的变化。

另外，我们删除了 `ReactTestUtils.SimulateNative` 的 helper 方法。他们从未被记录，没有按照他们名字所暗示的那样去做，也没有处理我们对事件系统所做的更改。如果你想要一种简便的方式来触发测试中原生浏览器的事件，请改用 React Testing Library。

安装

我们鼓励你尽快尝试 React 17.0 RC 版本，在迁移过程中遇到任何问题都可以向我们提出。**请注意，候选版本没有稳定版本稳定，因此请不要将其部署到生产环境。**

通过 npm 安装 React 17 RC 版，请执行：

```
npm install react@17.0.0-rc.3 react-dom@17.0.0-rc.3
```

通过 yarn 安装 React 17 RC 版，请执行：

```
yarn add react@17.0.0-rc.3 react-dom@17.0.0-rc.3
```

我们还通过 CDN 提供了 React RC 的 UMD 构建版本：

```
<script crossorigin
sr
c="https://unpkg.com/react@17.0.0-rc.3/umd/react.production.min.js"></
script>
<script crossorigin
src="https://unpkg.com/react-dom@17.0.0-rc.3/umd/react-
dom.production.min.js"></script>
```

有关详细安装说明，请参阅文档。

Changelog

React

- 为全新的 JSX 转换器添加 `react/jsx-runtime` 和 `react/jsx-dev-runtime`。（@lunaruan 提交于 #18299）
- 根据原生框架构建组件调用栈。（@sebmarkbage 提交于 #18561）
- 可以在 context 中设置 `displayName` 以改善调用栈信息。（@eps1lon 提交于 #18224）
- 防止 `'use strict'` 从 UMD 的 bundles 中泄露。（@koba04 提交于 #19614）
- 停止使用 `fb.me` 进行重定向。（@cylim 提交于 #19598）

React DOM

- 将事件委托从 `document` 切换为 root。（@trueadm 提交于 #18195 及其他）
- 在运行下一个副作用前，请清理所有副作用。（@bvaughn 提交于 #17947）
- 异步运行 `useEffect` 清理函数。（@bvaughn 提交于 #17925）
- 使用浏览器的 `focusin` 和 `focusout` 替换 `onFocus` 和 `onBlur` 的底层实现。（@trueadm 提交于 #19186）
- 将所有 `Capture` 事件都使用浏览器的捕获阶段实现。（@trueadm 提交于 #19221）
- 禁止在 `onScroll` 事件时冒泡。（@gaearon 提交于 #19464）
- 如果 `forwardRef` 或 `memo` 组件的返回值为 `undefined`，则抛出异常。（@gaearon 提交于 #19550）
- 移除事件池。（@trueadm 提交于 #18969）
- 移除 React Native Web 不需要的内部组件。（@necolas 提交于 #18483）
- 当挂载 root 时，附加所有已知的事件监听器。（@gaearon 提交于 #19659）
- 在 Dev 模式下，禁用第二次渲染过程中的 `console`。（@sebmarkbage 提交于 #18547）

- 弃用为记录且具有误导性的 `ReactTestUtils.SimulateNative` API。（@gaearon 提交于 #13407）
- 重命名内部使用的私有字段（@gaearon 提交于 #18377）
- 不在开发环境调用 User Timing API。（@gaearon 提交于 #18417）
- 在严格模式下重复渲染期间禁用 console。（@sebmarkbage 提交于 #18547）
- 在严格模式下，二次渲染组件也不使用 Hook。（@eps1lon 提交于 #18430）
- 允许在生命周期函数中调用 `ReactDOM.flushSync`（但会发出警告）。（@sebmarkbage 提交于 #18759）
- 将 `code` 属性添加到键盘事件对象中。（@bl00mber 提交于 #18287）
- 为 `video` 元素添加 `disableRemotePlayback` 属性。（@tombrowndev 提交于 #18619）
- 为 `input` 元素添加 `enterKeyHint` 属性。（@eps1lon 提交于 #18634）
- 当没有给 `<Context.Provider>` 提供任何值时，会发出警告。（@charlie1404 提交于 #19054）
- 如果 `forwardRef` 或 `memo` 组件的返回值为 `undefined`，则抛出警告。（@bvaughn 提交于 #19550）
- 为无效更新改进错误信息。（@JoviDeCroock 提交于 #18316）
- 从调用栈信息中忽略 forwardRef 和 memo。（@sebmarkbage 提交于 #18559）
- 在受控输入与非受控输入间切换时，改善错误消息。（@vcarl 提交于 #17070）
- 保持 `onTouchStart`、`onTouchMove` 和 `onWheel` 默认为 passive。（@gaearon 提交于 #19654）
- 修复在 development 模式下 iframe 关闭时，`setState` 挂起的问题。（@gaearon 提交于 #19220）
- 使用 `defaultProps` 修复拉架子组件在渲染时的问题。（@jddxf 提交于 #18539）
- 修复当 `dangerouslySetInnerHTML` 为 `undefined` 时，误报警告的问题。（@eps1lon 提交于 #18676）
- 使用非标准的 `require` 实现来修复 Test Utils。（@just-boris 提交于 #18632）
- 修复 `onBeforeInput` 报告错误的 `event.type`。（@eps1lon 提交于 #19561）
- 修复 Firefox 中 `event.relatedTarget` 输出为 `undefined` 的问题。（@claytercek 提交于 #19607）
- 修复 IE11 中 “unspecified error” 的问题。（@hemakshis 提交于 #19664）
- 修复 shadow root 中的渲染问题。（@Jack-Works 提交于 #15894）
- 使用事件捕获修复 `movementX/Y` polyfill 的问题。（@gaearon 提交于 #19672）
- 使用委托处理 `onSubmit` 和 `onReset` 事件。（@gaearon 提交于 #19333）
- 提高内存使用率。（@trueadm 提交于 #18970）

React DOM Server

- 使用服务端渲染的 `useCallback` 与 `useMemo` 一致。（@alexmckenley 提交于 #18783）
- 修复函数组件抛出异常时状态泄露的问题。（@pmaccart 提交于 #19212）

React Test Renderer

- 改善 `findByType` 错误信息。（@henryqdineen 提交于 #17439）

Concurrent Mode（实验阶段）

- 改进启发式更新算法。（@acdlite 提交于 #18796）
- 在实现性 API 前添加 `unstable_` 前缀。(@acdlite 提交于 #18825)
- 移除 `unstable_discreteUpdates` 和 `unstable_flushDiscreteUpdates`。（@trueadm 提交于 #18825）
- 移除了 `timeoutMs` 参数。（@acdlite 提交于 #19703）
- 禁用 `<div hidden />` 预渲染，以支持未来的 API。（@acdlite 提交于 #18917）
- 为 Suspense 添加了 `unstable_expectedLoadTime`，用于 CPU-bound 树。（@acdlite 提交于 #19936）
- 添加了一个实现性的 `unstable_useOpaqueIdentifier` Hook。（@lunaruan 提交于 #17322）
- 添加了一个实验性的 `unstable_startTransition` API. (@rickhanlonii 提交于 #19696)
- 在测试渲染器中使用 `act` 后，不在刷新 Suspense 的 fallback。（@acdlite 提交于 #18596）
- 将全局渲染的 timeout 用于 CPU Suspense。（@sebmarkbage 提交于 #19643）
- 挂载前，清除现有根目录的内容。（@bvaughn 提交于 #18730）
- 修复带有错误边界的 bug。（@acdlite 提交于 #18265）
- 修复了导致挂起树更新丢失的 bug。（@acdlite 提交于 #18384 以及 #18457）
- 修复导致渲染阶段更新丢失的 bug。（@acdlite 提交于 #18537）
- 修复 SuspenseList 的 bug。（@sebmarkbage 提交于 #18412）
- 修复导致 Suspense fallback 过早显示的 bug。（@acdlite 提交于 #18411）
- 修复 SuspenseList 中使用 class 组件异常的 bug。（@sebmarkbage 提交于 #18448）
- 修复输入内容可能被更新被丢弃的 bug。（@jddxf 提交于 #18515 以及 @acdlite 提交于 #18535）
- 修复暂挂 Suspense fallback 后卡住的错误。（@acdlite 提交于 #18663）
- 如果 hydrate 中，不要切断 SuspenseList 的尾部。（@sebmarkbage 提交于 #18854）
- 修复 `useMutableSource` 中的 bug，此 bug 可能在 `getSnapshot` 更改时出现。（@bvaughn 提交于 #18297）
- 修复 `useMutableSource` 令人恶心的 bug。（@bvaughn 提交于 #18912）
- 如果外部渲染且提交之前调用 `setState`，会发出警告。（@sebmarkbage 提交于 #18838）

React v16.13.0

February 26, 2020 by Sunil Pai

今天我们发布了 React 16.13.0。此版本修复了部分 bug 并添加了新的弃用警告，以助力接下来的主要版本。

新的警告

Render 期间更新的警告

React 组件不应在 render 期间对其他组件产生副作用。

在 render 期间调用 `setState` 是被支持的，但是 *仅仅适用于同一个组件*。 如果你在另一个组件 render 期间调用 `setState`，现在你将会看到一条警告。

```
Warning: Cannot update a component from inside the function body of a
different component.
```

这些警告将会帮助你找到应用中由意外的状态改变引起的 bug。 在极少数情况下，由于渲染，你有意要更改另一个组件的状态，你可以将 `setState` 调用包装为 `useEffect`。

冲突的样式规则警告

当动态地应用包含了全写和简写的 `style` 版本的 CSS 属性时，特定的更新组合可能会导致样式不一致。例如：

```
<div style={toggle ?
  { background: 'blue', backgroundColor: 'red' } :
  { backgroundColor: 'red' }
}>
  ...
</div>
```

你可能期待这个 `<div>` 总是拥有红色背景，不论 `toggle` 的值是什么。然而，在 `true` 和 `false` 之间交替使用 `toggle` 时，背景色开始是 `re`

d，然后在 transparent 和 blue 之间交替， 正如你能在这个 demo 中看到的。

React 现在检测到冲突的样式规则并打印出警告。要解决此问题，请不要在 style 属性中混合使用同一 CSS 属性的简写和全写版本。

某些废弃字符串 ref 的警告

字符串 ref 是过时的 API 这是不可取的，将来将被弃用：

```
<Button ref="myRef" />
```

(不要将字符串 ref 与一般的 ref 混淆，后者**仍然完全受支持**。)

在将来，我们将提供一个自动脚本（“codemod”），以从字符串 ref 中迁移。然而，一些罕见的案例不能自动迁移。此版本在弃用之前添加了一个新的警告**仅适用于那些情况**。

例如，如果将字符串 ref 与 Render Prop 模式一起使用，则它将触发：

```
class ClassWithRenderProp extends React.Component {
  componentDidMount() {
    doSomething(this.refs.myRef);
  }
  render() {
    return this.props.children();
  }
}

class ClassParent extends React.Component {
  render() {
    return (
      <ClassWithRenderProp>
        {() => <Button ref="myRef" />}
      </ClassWithRenderProp>
    );
  }
}
```

这样的代码通常暗含 bug。（你可能希望 ref 在 ClassParent 上可用，但是它被放在 ClassWithRenderProp 上）。

你很可能没有这样的代码。如果是有意为之，请将其转换为 `React.createRef()`：

```
class ClassWithRenderProp extends React.Component {
  myRef = React.createRef();
  componentDidMount() {
    doSomething(this.myRef.current);
  }
  render() {
    return this.props.children(this.myRef);
  }
}

class ClassParent extends React.Component {
  render() {
    return (
      <ClassWithRenderProp>
        {myRef => <Button ref={myRef} />}
      </ClassWithRenderProp>
    );
  }
}
```

Note

要查看此警告，你需要在你的 Babel plugins 中安装 babel-plugin-transform-react-jsx-self。它必须 *仅仅* 在开发模式下启用。

如果你使用 Create React App 或者用 babel 7+ 的 React preset，那么默认情况下已经安装了这个插件。

弃用 `React.createFactory`

`React.createFactory` 为 React 创建一个帮助器元素。此版本向该方法添加了一个弃用警告。它将在未来的主要版本中删除。

把 `React.createFactory` 替换为普通的 JSX 。或者可以复制并粘贴此单行辅助对象或将其发布为库：

```
let createFactory = type => React.createElement.bind(null, type);
```

它的作用完全相同。

弃用 ReactDOM.unstable_createPortal 推荐 ReactDOM.createPortal

当 React 16 发布时，createPortal 成为一个官方支持的 API。

但是，我们保留了 unstable_createPortal 作为受支持的别名，以使采用它的少数库正常工作。我们现在反对使用 unstable 别名。直接使用 createPortal 而不是 unstable_createPortal。它有完全相同的签名。

其他改进

Hydration 过程中组件堆栈的警告

React 将组件堆栈添加到其开发警告中，使开发人员能够隔离 bug 并调试他们的程序。此版本将组件堆栈添加到许多以前没有的开发警告中。举个例子，考虑一下以前版本中的 hydration 警告：

虽然它指出了代码中的一个错误，但不清楚错误在哪里存在，以及下一步该怎么做。此版本向此警告添加了一个组件堆栈，使其看起来如下所示：

这样可以清楚地看出问题所在，并让你更快地找到并修复错误。

值得注意的错误修复

此版本包含其他一些值得注意的改进：

- 在严格的开发模式下，React 调用生命周期方法两次，以清除任何可能不需要的副作用。此版本将此行为添加到 shouldComponentUpdate 中。这不会影响大多

数代码，除非在 `shouldComponentUpdate` 中有副作用。要解决此问题，请将具有副作用的代码移到 `componentDidUpdate` 中。

- 在严格开发模式下，使用遗留上下文 API 的警告不包括触发警告的组件堆栈。此版本将丢失的堆栈添加到警告中。
- `onMouseEnter` 现在在被禁用的 `<button>` 对象上不能被触发。
- 自从我们发布 v16 以来，ReactDOM 缺少一个 `version` 导出。这个版本又添加了它。我们不建议在应用程序逻辑中使用它，但是在调试同一页面上的 ReactDOM 的不匹配/多个版本时，它非常有用。

我们感谢所有帮助揭示和解决这些和其他问题的贡献者。你可以找到完整的变更日志 如下。

安装

React

React v16.13.0 现在在 npm 库中已经可用。

用 Yarn 安装 React 16，运行：

```
yarn add react@^16.13.0 react-dom@^16.13.0
```

用 npm 安装 React 16，运行：

```
npm install --save react@^16.13.0 react-dom@^16.13.0
```

我们也通过 CDN 提供了 React 的 UMD 版本：

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

参考这篇文档 详细安装说明。

变更日志

React

- 当字符串 ref 的使用方式不符合将来的代码模式时发出警告 (@lunaruan in #17864)
- 弃用 `React.createFactory()` (@trueadm in #17878)

React DOM

- `style` 中的更改可能导致意外冲突时发出警告 (@sophiebits 在 #14181，#18002)

- 在另一个组件的 render 阶段更新 function 组件时发出警告 (@acdlite 在 #17099)
- 弃用 `unstable_createPortal` (@trueadm 在 #17880)
- 修复 `onMouseEnter` 在被禁用的按钮上被触发 (@AlfredoGJ 在 #17675)
- 在 `StrictMode` 下调用 `shouldComponentUpdate` 两次 (@bvaughn 在 #17942)
- 增加 ReactDOM `version` 属性 (@ealush 在 #15780)
- 不要调用 `dangerouslySetInnerHTML` 的 `toString()` 方法 (@sebmarkbage 在 #17773)
- 在组件堆栈中展示更多警告 (@gaearon 在 #17922，#17586)

并发模式（实验）

- 警告有问题的用法 `ReactDOM.createRoot()` (@trueadm 在 #17937)
- 移除 `ReactDOM.createRoot()` 回调传参并且在用法上增加了警告 (@bvaughn 在 #17916)
- 不要将空闲/屏幕外的工作与其他工作分组 (@sebmarkbage 在 #17456)
- 调整 `SuspenseList` CPU 边界启发式 (@sebmarkbage 在 #17455)
- 增加丢失事件插件属性 (@trueadm 在 #17914)
- 修复 `isPending` 仅当从输入事件内部转换时为 true (@acdlite 在 #17382)
- 修复 `React.memo` 组件在被更高优先级的更新中断时删除更新 (@acdlite 在 #18091)
- 以错误的优先级暂停时不发出警告 (@gaearon 在 #17971)
- 用重定基更新修复一个 bug (@acdlite 和 @sebmarkbage 在 #17560，#17510，#17483，#17480)

React v18.0

March 29, 2022 by The React Team

React 18 is now available on npm!

In our last post, we shared step-by-step instructions for upgrading your app to React 18. In this post, we'll give an overview of what's new in React 18, and what it means for the future.

Our latest major version includes out-of-the-box improvements like automatic batching, new APIs like startTransition, and streaming server-side rendering with support for Suspense.

Many of the features in React 18 are built on top of our new concurrent renderer, a behind-the-scenes change that unlocks powerful new capabilities. Concurrent React is opt-in — it's only enabled when you use a concurrent feature — but we think it will have a big impact on the way people build applications.

We've spent years researching and developing support for concurrency in React, and we've taken extra care to provide a gradual adoption path for existing users. Last summer, we formed the React 18 Working Group to gather feedback from experts in the community and ensure a smooth upgrade experience for the entire React ecosystem.

In case you missed it, we shared a lot of this vision at React Conf 2021:

- In the keynote, we explain how React 18 fits into our mission to make it easy for developers to build great user experiences
- Shruti Kapoor demonstrated how to use the new features in React 18
- Shaundai Person gave us an overview of streaming server rendering with Suspense

Below is a full overview of what to expect in this release, starting with Concurrent Rendering.

Note for React Native users: React 18 will ship in React Native with the New React Native Architecture. For more information, see the React Conf keynote here.

What is Concurrent React?

The most important addition in React 18 is something we hope you never have to think about: concurrency. We think this is largely true for application developers, though the story may be a bit more complicated for library maintainers.

Concurrency is not a feature, per se. It's a new behind-the-scenes mechanism that enables React to prepare multiple versions of your UI at the same time. You can think of concurrency as an implementation detail — it's valuable because of the features that it unlocks. React uses sophisticated techniques in its internal implementation, like priority queues and multiple buffering. But you won't see those concepts anywhere in our public APIs.

When we design APIs, we try to hide implementation details from developers. As a React developer, you focus on *what* you want the user experience to look like, and React handles *how* to deliver that experience. So we don't expect React developers to know how concurrency works under the hood.

However, Concurrent React is more important than a typical implementation detail — it's a foundational update to React's core rendering model. So while it's not super important to know how concurrency works, it may be worth knowing what it is at a high level.

A key property of Concurrent React is that rendering is interruptible. When you first upgrade to React 18, before adding any concurrent features, updates are rendered the same as in previous versions of React — in a single, uninterrupted, synchronous transaction. With synchronous rendering, once an update starts rendering, nothing can interrupt it until the user can see the result on screen.

In a concurrent render, this is not always the case. React may start rendering an update, pause in the middle, then continue later. It may even abandon an in-progress render altogether. React guarantees that the UI will appear consistent even if a render is interrupted. To do this, it waits to perform DOM mutations until the end, once the entire tree has been evaluated. With this capability, React can prepare new screens in the background without blocking the main thread. This means the UI can respond immediately to user input even if it's in the middle of a large rendering task, creating a fluid user experience.

Another example is reusable state. Concurrent React can remove sections of the UI from the screen, then add them back later while reusing the previous state. For example, when a user tabs away from a screen and back, React should be able to restore the previous screen in the same state it was in before. In an upcoming minor, we're planning to add a new component called `<Offscreen>` that implements this pattern. Similarly, you'll be able to use Offscreen to prepare new UI in the background so that it's ready before the user reveals it.

Concurrent rendering is a powerful new tool in React and most of our new features are built to take advantage of it, including Suspense, transitions, and streaming server rendering. But React 18 is just the beginning of what we aim to build on this new foundation.

Gradually Adopting Concurrent Features

Technically, concurrent rendering is a breaking change. Because concurrent rendering is interruptible, components behave slightly differently when it is enabled.

In our testing, we've upgraded thousands of components to React 18. What we've found is that nearly all existing components "just work" with concurrent rendering, without any changes. However, some of them may require some additional migration effort. Although the changes are

usually small, you'll still have the ability to make them at your own pace. The new rendering behavior in React 18 is **only enabled in the parts of your app that use new features.**

The overall upgrade strategy is to get your application running on React 18 without breaking existing code. Then you can gradually start adding concurrent features at your own pace. You can use `<StrictMode>` to help surface concurrency-related bugs during development. Strict Mode doesn't affect production behavior, but during development it will log extra warnings and double-invoke functions that are expected to be idempotent. It won't catch everything, but it's effective at preventing the most common types of mistakes.

After you upgrade to React 18, you'll be able to start using concurrent features immediately. For example, you can use startTransition to navigate between screens without blocking user input. Or useDeferredValue to throttle expensive re-renders.

However, long term, we expect the main way you'll add concurrency to your app is by using a concurrent-enabled library or framework. In most cases, you won't interact with concurrent APIs directly. For example, instead of developers calling startTransition whenever they navigate to a

new screen, router libraries will automatically wrap navigations in startTransition.

It may take some time for libraries to upgrade to be concurrent compatible. We've provided new APIs to make it easier for libraries to take advantage of concurrent features. In the meantime, please be patient with maintainers as we work to gradually migrate the React ecosystem.

For more info, see our previous post: How to upgrade to React 18.

Suspense in Data Frameworks

In React 18, you can start using Suspense for data fetching in opinionated frameworks like Relay, Next.js, Hydrogen, or Remix. Ad hoc data fetching with Suspense is technically possible, but still not recommended as a general strategy.

In the future, we may expose additional primitives that could make it easier to access your data with Suspense, perhaps without the use of an opinionated framework. However, Suspense works best when it's deeply integrated into your application's architecture: your router, your data layer, and your server rendering environment. So even long term, we

expect that libraries and frameworks will play a crucial role in the React ecosystem.

As in previous versions of React, you can also use Suspense for code splitting on the client with React.lazy. But our vision for Suspense has always been about much more than loading code — the goal is to extend support for Suspense so that eventually, the same declarative Suspense fallback can handle any asynchronous operation (loading code, data, images, etc).

Server Components is Still in Development

Server Components is an upcoming feature that allows developers to build apps that span the server and client, combining the rich interactivity of client-side apps with the improved performance of traditional server rendering. Server Components is not inherently coupled to Concurrent React, but it's designed to work best with concurrent features like Suspense and streaming server rendering.

Server Components is still experimental, but we expect to release an initial version in a minor 18.x release. In the meantime, we're working with frameworks like Next.js, Hydrogen, and Remix to advance the proposal and get it ready for broad adoption.

What's New in React 18

New Feature: Automatic Batching

Batching is when React groups multiple state updates into a single re-render for better performance. Without automatic batching, we only batched updates inside React event handlers. Updates inside of promises, setTimeout, native event handlers, or any other event were not batched in React by default. With automatic batching, these updates will be batched automatically:

```
// Before: only React events were batched.
setTimeout(() => {
  setCount(c => c + 1);
  setFlag(f => !f);
  // React will render twice, once for each state update (no batching)
}, 1000);

// After: updates inside of timeouts, promises,
// native event handlers or any other event are batched.
setTimeout(() => {
  setCount(c => c + 1);
  setFlag(f => !f);
  // React will only re-render once at the end (that's batching!)
}, 1000);
```

For more info, see this post for Automatic batching for fewer renders in React 18.

New Feature: Transitions

A transition is a new concept in React to distinguish between urgent and non-urgent updates.

- **Urgent updates** reflect direct interaction, like typing, clicking, pressing, and so on.
- **Transition updates** transition the UI from one view to another.

Urgent updates like typing, clicking, or pressing, need immediate response to match our intuitions about how physical objects behave. Otherwise they feel “wrong”. However, transitions are different because the user doesn’t expect to see every intermediate value on screen.

For example, when you select a filter in a dropdown, you expect the filter button itself to respond immediately when you click. However, the actual results may transition separately. A small delay would be imperceptible and often expected. And if you change the filter again before the results are done rendering, you only care to see the latest results.

Typically, for the best user experience, a single user input should result in both an urgent update and a non-urgent one. You can use startTransition API inside an input event to inform React which updates are urgent and which are “transitions”:

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

// Urgent: Show what was typed
setInputValue(input);

// Mark any state updates inside as transitions
startTransition(() => {
  // Transition: Show the results
  setSearchQuery(input);
});
```

Updates wrapped in startTransition are handled as non-urgent and will be interrupted if more urgent updates like clicks or key presses come in. If a transition gets interrupted by the user (for example, by typing multiple characters in a row), React will throw out the stale rendering work that wasn't finished and render only the latest update.

- `useTransition`: a hook to start transitions, including a value to track the pending state.
- `startTransition`: a method to start transitions when the hook cannot be used.

Transitions will opt in to concurrent rendering, which allows the update to be interrupted. If the content re-suspends, transitions also tell React to continue showing the current content while rendering the transition content in the background (see the Suspense RFC for more info).

See docs for transitions here.

New Suspense Features

Suspense lets you declaratively specify the loading state for a part of the component tree if it's not yet ready to be displayed:

```
<Suspense fallback={<Spinner />}>
  <Comments />
</Suspense>
```

Suspense makes the "UI loading state" a first-class declarative concept in the React programming model. This lets us build higher-level features on top of it.

We introduced a limited version of Suspense several years ago. However, the only supported use case was code splitting with React.lazy, and it wasn't supported at all when rendering on the server.

In React 18, we've added support for Suspense on the server and expanded its capabilities using concurrent rendering features.

Suspense in React 18 works best when combined with the transition API. If you suspend during a transition, React will prevent already-visible content from being replaced by a fallback. Instead, React will delay the render until enough data has loaded to prevent a bad loading state. For more, see the RFC for Suspense in React 18.

New Client and Server Rendering APIs

In this release we took the opportunity to redesign the APIs we expose for rendering on the client and server. These changes allow users to continue using the old APIs in React 17 mode while they upgrade to the new APIs in React 18.

React DOM Client

These new APIs are now exported from `react-dom/client`:

- `createRoot`: New method to create a root to `render` or `unmount`. Use it instead of `ReactDOM.render`. New features in React 18 don't work without it.
- `hydrateRoot`: New method to hydrate a server rendered application. Use it instead of `ReactDOM.hydrate` in conjunction with the new React DOM Server APIs. New features in React 18 don't work without it.

Both `createRoot` and `hydrateRoot` accept a new option called `onRecoverableError` in case you want to be notified when React recovers from errors during rendering or hydration for logging. By default, React will use `reportError`, or `console.error` in the older browsers.

See docs for React DOM Client here.

React DOM Server

These new APIs are now exported from `react-dom/server` and have full support for streaming Suspense on the server:

- `renderToPipeableStream`: for streaming in Node environments.
- `renderToReadableStream`: for modern edge runtime environments, such as Deno and Cloudflare workers.

The existing `renderToString` method keeps working but is discouraged.

See docs for React DOM Server here.

New Strict Mode Behaviors

In the future, we'd like to add a feature that allows React to add and remove sections of the UI while preserving state. For example, when a user tabs away from a screen and back, React should be able to immediately show the previous screen. To do this, React would unmount and remount trees using the same component state as before.

This feature will give React apps better performance out-of-the-box, but requires components to be resilient to effects being mounted and destroyed multiple times. Most effects will work without any changes, but some effects assume they are only mounted or destroyed once.

To help surface these issues, React 18 introduces a new development-only check to Strict Mode. This new check will automatically unmount and remount every component, whenever a component mounts for the first time, restoring the previous state on the second mount.

Before this change, React would mount the component and create the effects:

```
* React mounts the component.
  * Layout effects are created.
  * Effects are created.
```

With Strict Mode in React 18, React will simulate unmounting and remounting the component in development mode:

```
* React mounts the component.
  * Layout effects are created.
  * Effects are created.
* React simulates unmounting the component.
  * Layout effects are destroyed.
  * Effects are destroyed.
* React simulates mounting the component with the previous state.
  * Layout effects are created.
  * Effects are created.
```

See docs for ensuring reusable state here.

New Hooks

useId

`useId` is a new hook for generating unique IDs on both the client and server, while avoiding hydration mismatches. It is primarily useful for component libraries integrating with accessibility APIs that require unique IDs. This solves an issue that already exists in React 17 and below, but it's even more important in React 18 because of how the new streaming server renderer delivers HTML out-of-order. See docs here.

Note

`useId` is **not** for generating keys in a list. Keys should be generated from your data.

useTransition

`useTransition` and `startTransition` let you mark some state updates as not urgent. Other state updates are considered urgent by default. React will allow urgent state updates (for example, updating a text input) to interrupt non-urgent state updates (for example, rendering a list of search results). See docs here

useDeferredValue

`useDeferredValue` lets you defer re-rendering a non-urgent part of the tree. It is similar to debouncing, but has a few advantages compared to

it. There is no fixed time delay, so React will attempt the deferred render right after the first render is reflected on the screen. The deferred render is interruptible and doesn't block user input. See docs here.

useSyncExternalStore

`useSyncExternalStore` is a new hook that allows external stores to support concurrent reads by forcing updates to the store to be synchronous. It removes the need for useEffect when implementing subscriptions to external data sources, and is recommended for any library that integrates with state external to React. See docs here.

Note

`useSyncExternalStore` is intended to be used by libraries, not application code.

useInsertionEffect

`useInsertionEffect` is a new hook that allows CSS-in-JS libraries to address performance issues of injecting styles in render. Unless you've already built a CSS-in-JS library we don't expect you to ever use this. This hook will run after the DOM is mutated, but before layout effects read the new layout. This solves an issue that already exists in React 17 and below, but is even more important in React 18 because React yields to the browser during concurrent rendering, giving it a chance to recalculate layout. See docs here.

Note

`useInsertionEffect` is intended to be used by libraries, not application code.

How to Upgrade

See How to Upgrade to React 18 for step-by-step instructions and a full list of breaking and notable changes.

Changelog

React

- Add `useTransition` and `useDeferredValue` to separate urgent updates from transitions.
  (#10426, #10715, #15593, #15272, #15578, #15769, #17058, #18796, #19121, #19703, #19719, #19724, #20672, #20976 by @acdlite, @lunaruan, @rickhanlonii, and @sebmarkbage)
- Add `useId` for generating unique IDs.
  (#17322, #18576, #22644, #22672, #21260 by @acdlite, @lunaruan, and @sebmarkbage)
- Add `useSyncExternalStore` to help external store libraries integrate with React.
  (#15022, #18000, #18771, #22211, #22292, #22239, #22347, #23150 by @acdlite, @bvaughn, and @drarmstr)
- Add `startTransition` as a version of `useTransition` without pending feedback. (#19696 by @rickhanlonii)
- Add `useInsertionEffect` for CSS-in-JS libraries. (#21913 by @rickhanlonii)
- Make Suspense remount layout effects when content reappears.
  (#19322, #19374, #19523, #20625, #21079 by @acdlite, @bvaughn, and @lunaruan)
- Make `<StrictMode>` re-run effects to check for restorable state.
  (#19523 , #21418 by @bvaughn and @lunaruan)
- Assume Symbols are always available. (#23348 by @sebmarkbage)
- Remove `object-assign` polyfill. (#23351 by @sebmarkbage)
- Remove unsupported `unstable_changedBits` API. (#20953 by @acdlite)
- Allow components to render undefined. (#21869 by @rickhanlonii)
- Flush `useEffect` resulting from discrete events like clicks synchronously. (#21150 by @acdlite)
- Suspense `fallback={undefined}` now behaves the same as `null` and isn't ignored. (#21854 by @rickhanlonii)

- Consider all `lazy()` resolving to the same component equivalent. (#20357 by @sebmarkbage)
- Don't patch console during first render. (#22308 by @lunaruan)
- Improve memory usage. (#21039 by @bgirard)
- Improve messages if string coercion throws (Temporal.*, Symbol, etc.) (#22064 by @justingrant)
- Use `setImmediate` when available over `MessageChannel`. (#20834 by @gaearon)
- Fix context failing to propagate inside suspended trees. (#23095 by @gaearon)
- Fix `useReducer` observing incorrect props by removing the eager bailout mechanism. (#22445 by @josephsavona)
- Fix `setState` being ignored in Safari when appending iframes. (#23111 by @gaearon)
- Fix a crash when rendering `ZonedDateTime` in the tree. (#20617 by @dimaqq)
- Fix a crash when document is set to `null` in tests. (#22695 by @SimenB)
- Fix `onLoad` not triggering when concurrent features are on. (#23316 by @gnoff)
- Fix a warning when a selector returns `NaN`. (#23333 by @hachibeeDI)
- Fix a crash when document is set to `null` in tests. (#22695 by @SimenB)
- Fix the generated license header. (#23004 by @vitaliemiron)
- Add `package.json` as one of the entry points. (#22954 by @Jack)
- Allow suspending outside a Suspense boundary. (#23267 by @acdlite)
- Log a recoverable error whenever hydration fails. (#23319 by @acdlite)

React DOM

- Add `createRoot` and `hydrateRoot`. (#10239, #11225, #12117, #13732, #15502, #15532, #17035, #17165, #20669, #20748, #20888, #21072, #21417, #21652, #21687, #23207, #23385 by @acdlite, @bvaughn, @gaearon, @lunaruan, @rickhanlonii, @trueadm, and @sebmarkbage)
- Add selective hydration. (#14717, #14884, #16725, #16880, #17004, #22416, #22629, #22448, #22856, #23176 by @acdlite, @gaearon, @salazarm, and @sebmarkbage)
- Add `aria-description` to the list of known ARIA attributes. (#22142 by @mahyareb)
- Add `onResize` event to video elements. (#21973 by @rileyjshaw)
- Add `imageSizes` and `imageSrcSet` to known props. (#22550 by @eps1lon)
- Allow non-string `<option>` children if `value` is provided. (#21431 by @sebmarkbage)
- Fix `aspectRatio` style not being applied. (#21100 by @gaearon)
- Warn if `renderSubtreeIntoContainer` is called. (#23355 by @acdlite)

React DOM Server

- Add the new streaming renderer. (#14144, #20970, #21056, #21255, #21200, #21257, #21276, #22443, #22450, #23247, #24025, #24030 by @sebmarkbage)

- Fix context providers in SSR when handling multiple requests. (#23171 by @frandiox)
- Revert to client render on text mismatch. (#23354 by @acdlite)
- Deprecate `renderToNodeStream`. (#23359 by @sebmarkbage)
- Fix a spurious error log in the new server renderer. (#24043 by @eps1lon)
- Fix a bug in the new server renderer. (#22617 by @shuding)
- Ignore function and symbol values inside custom elements on the server. (#21157 by @sebmarkbage)

React DOM Test Utils

- Throw when `act` is used in production. (#21686 by @acdlite)
- Support disabling spurious act warnings with `global.IS_REACT_ACT_ENVIRONMENT`. (#22561 by @acdlite)
- Expand act warning to cover all APIs that might schedule React work. (#22607 by @acdlite)
- Make `act` batch updates. (#21797 by @acdlite)
- Remove warning for dangling passive effects. (#22609 by @acdlite)

React Refresh

- Track late-mounted roots in Fast Refresh. (#22740 by @anc95)
- Add `exports` field to `package.json`. (#23087 by @otakustay)

Server Components (Experimental)

- Add Server Context support. (#23244 by @salazarm)
- Add `lazy` support. (#24068 by @gnoff)
- Update webpack plugin for webpack 5 (#22739 by @michenly)
- Fix a mistake in the Node loader. (#22537 by @btea)
- Use `globalThis` instead of `window` for edge environments. (#22777 by @huozhi)

介绍全新的 JSX 转换

September 22, 2020 by Luna Ruan

虽然 React 17 并未包含新特性，但它将提供一个全新版本的 JSX 转换。本文中，我们将为你描述它是什么以及如何使用。

何为 JSX 转换？

在浏览器中无法直接使用 JSX，所以大多数 React 开发者需依靠 Babel 或 TypeScript 来**将 JSX 代码转换为 JavaScript**。许多包含预配置的工具，例如 Create React App 或 Next.js，在其内部也引入了 JSX 转换。React 17 发布在即，尽管我们想对 JSX 的转换进行改进，但我们不想打破现有的配置。于是我们选择与 Babel 合作 ，为想要升级的开发者**提供了一个全新的，重构过的 JSX 转换的版本**。

升级至全新的转换完全是可选的，但升级它会为你带来一些好处：

- 使用全新的转换，你可以**单独使用 JSX 而无需引入 React**。
- 根据你的配置，JSX 的编译输出可能会**略微改善 bundle 的大小**。
- 它将**减少你需要学习 React 概念的数量**，以备未来之需。

此次升级不会改变 JSX 语法，也并非必须。旧的 JSX 转换将继续工作，没有计划取消对它的支持。

React 17 的 RC 版本 已经引入了对新转换的支持，所以你可以尝试一下！为了让大家更容易使用，在 React 17 正式发布后，我们还将此支持移植到了 React 16.14.0，React 15.7.0 以及 React 0.14.10。你可以在下方找到不同工具的升级说明。

接下来，我们来仔细对比新旧转换的区别。

新的转换有何不同？

当你使用 JSX 时，编译器会将其转换为浏览器可以理解的 React 函数调用。**旧的 JSX 转换**会把 JSX 转换为 `React.createElement(...)` 调用。

例如，假设源代码如下：

```
import React from 'react';

function App() {
  return <h1>Hello World</h1>;
}
```

旧的 JSX 转换会将上述代码变成普通的 JavaScript 代码：

```
import React from 'react';

function App() {
  return React.createElement('h1', null, 'Hello world');
}
```

注意

无需改变源码。我们将介绍 JSX 转换如何将你的 JSX 源码变成浏览器可以理解的 JavaScript 代码。

然而，这并不完美：

- 如果使用 JSX，则需在 `React` 的环境下，因为 JSX 将被编译成 `React.createElement`。
- 有一些 `React.createElement` 无法做到的性能优化和简化。

为了解决这些问题，React 17 在 React 的 package 中引入了两个新入口，这些入口只会被 Babel 和 TypeScript 等编译器使用。新的 JSX 转换**不会将 JSX 转换为 `React.createElement`**，而是自动从 React 的 package 中引入新的入口函数并调用。

假设你的源代码如下：

```
function App() {
  return <h1>Hello World</h1>;
}
```

下方是新 JSX 被转换编译后的结果：

```
// 由编译器引入（禁止自己引入！）
import {jsx as _jsx} from 'react/jsx-runtime';

function App() {
  return _jsx('h1', { children: 'Hello world' });
}
```

注意，此时源代码**无需引入 React** 即可使用 JSX 了！（但仍需引入 React，以便使用 React 提供的 Hook 或其他导出。）

此变化与所有现有 JSX 代码兼容，所以你无需修改组件。如果你对此感兴趣，你可以查看 RFC 了解全新转换工作的具体细节。

注意

`react/jsx-runtime` 和 `react/jsx-dev-runtime` 中的函数只能由编译器转换使用。如果你需要在代码中手动创建元素，你可以继续使用 `React.createElement`。它将继续工作，不会消失。

如何升级至新的 JSX 转换

如果你还没准备好升级为全新的 JSX 转换，或者你正在为其他库使用 JSX，请不要担心，旧的转换不会被移除，并将继续支持。

如果你想升级，你需要准备两件事：

- **支持新转换的 React 版本**（React 17 的 RC 版本 及更高版本支持它，但是我们也发布了 React 16.14.0，React 15.7.0 以及 0.14.10 等主要版本，以供还在使用旧版本的开发者使用它们）
- **一个兼容新转换的编译器**（请看下面关于不同工具的说明）。

由于新的 JSX 转换不依赖 React 环境，我们准备了一个自动脚本，用于移除你代码中不必要的引入。

Create React App

Create React App 4.0.0+ 使用了兼容 React 版本的 JSX 转换。

Next.js

Next.js 的 v9.5.3+ 会使用新的转换来兼容 React 版本。

Gatsby

Gatsby 的 v2.24.5+ 会使用新的转换来兼容 React 版本。

注意

如果你在 Gatsby 中遇到 error，请升级至 React 17 的 RC 版本，运行 `npm update` 解决此问题。

手动设置 Babel

Babel 的 v7.9.0 及以上版本可支持全新的 JSX 转换。

首先，你需要更新至最新版本的 Babel 和 transform 插件。

如果你使用的是 `@babel/plugin-transform-react-jsx`：

```
# npm 用户
npm update @babel/core @babel/plugin-transform-react-jsx
# yarn 用户
yarn upgrade @babel/core @babel/plugin-transform-react-jsx
```

如果你使用的是 `@babel/preset-react`：

```
# npm 用户
npm update @babel/core @babel/preset-react
# yarn 用户
yarn upgrade @babel/core @babel/preset-react
```

目前，旧的转换的默认选项为 `{"runtime": "classic"}`。如需启用新的转换，你可以使用 `{"runtime": "automatic"}` 作为 `@babel/plugin-transform-react-jsx` 或 `@babel/preset-react` 的选项：

```
// 如果你使用的是 @babel/preset-react
{
  "presets": [
    ["@babel/preset-react", {
      "runtime": "automatic"
    }]
  ]
}
// 如果你使用的是 @babel/plugin-transform-react-jsx
{
  "plugins": [
    ["@babel/plugin-transform-react-jsx", {
```

```
      "runtime": "automatic"
    }]
  ]
}
```

从 Babel 8 开始，`"automatic"` 会将两个插件默认集成在 rumtime 中。

欲了解更多信息，请查阅 Babel 文档中的 @babel/plugin-transform-react-jsx 以及 @babel/preset-react。

注意

如果你在使用 JSX 时，使用 React 以外的库，你可以使用 `importSource` 选项从该库中引入 — 前提是它提供了必要的入口。或者你可以继续使用经典的转换，它会继续被支持。

如果你是库的作者并且需要为你的库实现 `/jsx-runtime` 的入口，需注意一种情况，在此情况下，为了向下兼容，即使使用了新的 jsx 转换，也必须考虑 `createElement`。在上述情况中，将直接从 `importSource` 的根入口中自动引入 `createElement`。

ESLint

如果你正在使用 eslint-plugin-react，其中的 `react/jsx-uses-react` 和 `react/react-in-jsx-scope` 规则将不再需要，可以关闭它们或者删除。

```
{
  // ...
  "rules": {
    // ...
    "react/jsx-uses-react": "off",
    "react/react-in-jsx-scope": "off"
  }
}
```

TypeScript

TypeScript 将在 v4.1 及以上版本中支持新的 JSX 转换。

Flow

Flow 将在 v0.126.0 中支持新的 JSX 转换。

移除未使用的 React 引入

因为新的 JSX 转换会自动引入必要的 `react/jsx-runtime` 函数，因此当你使用 JSX 时，将无需再引入 React。将可能会导致你代码中有未使用到的 React 引入。保留它们也无伤大雅，但如果你想删除它们，我们建议运行 “codemod” 脚本来自动删除它们：

```
cd your_project
npx react-codemod update-react-imports
```

注意：

如果你在运行 codemod 时出现错误，请尝试使用 `npx react-codemod update-react-imports` 选择不同的 JavaScript 环境。尤其是选择 “JavaScript with Flow” 时，即使你未使用 Flow，也可以选择它，因为它比 JavaScript 支持更新的语法。如果遇到问题，请告知我们。

请注意，codemod 的输出可能与你的代码风格并不匹配，因此你可能需要再 codemod 完成后运行 Prettier 以保证格式一致。

运行 codemod 会执行如下操作：

- 升级到新的 JSX 转换，删除所有未使用的 React 引入。
- 所有 React 的默认引入将被改为解构命名引入（例如，`import React from "react"` 会变成 `import { useState } from "react"`），这将成为未来开发的首选风格。codemod **不会** 影响现有的命名空间引入方式（即 `import * as React from "react"`），这也是一种有效的风格。默认的引入将在 React 17 中继续工作，但从长远来看，我们建议尽量不使用它们。

示例：

```
import React from 'react';
```

```
function App() {
  return <h1>Hello World</h1>;
}
```

将被替换为

```
function App() {
  return <h1>Hello World</h1>;
}
```

如果你使用了 React 的其他导出 — 比如 Hook，那么 codemod 将把它们转换为具名导入。

示例：

```
import React from 'react';

function App() {
  const [text, setText] = React.useState('Hello World');
  return <h1>{text}</h1>;
}
```

会被替换为

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

function App() {
  const [text, setText] = useState('Hello World');
  return <h1>{text}</h1>;
}
```

除了清理未使用的引入外，此工具还可帮你为未来 React 主要版本（不是 React 17 版本）做铺垫，该版本将支持 ES 模块，并且没有默认导出。

鸣谢

我们要感谢 Babel，TypeScript，Create React App，Next.js，Gatsby，ESLint 以及 Flow 的主要维护者为新 JSX 转换提

供的实现和整合。我们还要感谢 React 社区对相关 RFC 提供的反馈和讨论。

How to Upgrade to React 18

March 08, 2022 by Rick Hanlon

As we shared in the release post, React 18 introduces features powered by our new concurrent renderer, with a gradual adoption strategy for existing applications. In this post, we will guide you through the steps for upgrading to React 18.

Please report any issues you encounter while upgrading to React 18.

Note for React Native users: React 18 will ship in a future version of React Native. This is because React 18 relies on the New React Native Architecture to benefit from the new capabilities presented in this blogpost. For more information, see the React Conf keynote here.

Installing

To install the latest version of React:

```
npm install react react-dom
```

Or if you're using yarn:

```
yarn add react react-dom
```

Updates to Client Rendering APIs

When you first install React 18, you will see a warning in the console:

ReactDOM.render is no longer supported in React 18. Use createRoot instead. Until you switch to the new API, your app will behave as if it's running React 17. Learn more: https://reactjs.org/link/switch-to-createroot

React 18 introduces a new root API which provides better ergonomics for managing roots. The new root API also enables the new concurrent renderer, which allows you to opt-into concurrent features.

```
// Before
import { render } from 'react-dom';
const container = document.getElementById('app');
render(<App tab="home" />, container);

// After
import { createRoot } from 'react-dom/client';
const container = document.getElementById('app');
const root = createRoot(container); // createRoot(container!) if you use TypeScript
root.render(<App tab="home" />);
```

We've also changed `unmountComponentAtNode` to `root.unmount`:

```
// Before
unmountComponentAtNode(container);

// After
root.unmount();
```

We've also removed the callback from render, since it usually does not have the expected result when using Suspense:

```
// Before
const container = document.getElementById('app');
render(<App tab="home" />, container, () => {
  console.log('rendered');
});

// After
function AppWithCallbackAfterRender() {
  useEffect(() => {
    console.log('rendered');
  });

  return <App tab="home" />
}

const container = document.getElementById('app');
const root = createRoot(container);
root.render(<AppWithCallbackAfterRender />);
```

Note:

There is no one-to-one replacement for the old render callback API — it depends on your use case. See the working group post for Replacing render with createRoot for more information.

Finally, if your app uses server-side rendering with hydration, upgrade `hydrate` to `hydrateRoot`:

```
// Before
import { hydrate } from 'react-dom';
const container = document.getElementById('app');
hydrate(<App tab="home" />, container);

// After
import { hydrateRoot } from 'react-dom/client';
const container = document.getElementById('app');
const root = hydrateRoot(container, <App tab="home" />);
```

```
// Unlike with createRoot, you don't need a separate root.render() call
here.
```

For more information, see the working group discussion here.

Note

If your app doesn't work after upgrading, check whether it's wrapped in `<StrictMode>`. Strict Mode has gotten stricter in React 18, and not all your components may be resilient to the new checks it adds in development mode. If removing Strict Mode fixes your app, you can remove it during the upgrade, and then add it back (either at the top or for a part of the tree) after you fix the issues that it's pointing out.

Updates to Server Rendering APIs

In this release, we're revamping our `react-dom/server` APIs to fully support Suspense on the server and Streaming SSR. As part of these changes, we're deprecating the old Node streaming API, which does not support incremental Suspense streaming on the server.

Using this API will now warn:

- `renderToNodeStream`: **Deprecated** ⛔

Instead, for streaming in Node environments, use:

- `renderToPipeableStream`: **New** ✨

We're also introducing a new API to support streaming SSR with Suspense for modern edge runtime environments, such as Deno and Cloudflare workers:

- `renderToReadableStream`: **New** ✨

The following APIs will continue working, but with limited support for Suspense:

- `renderToString`: **Limited** ⚠
- `renderToStaticMarkup`: **Limited** ⚠

Finally, this API will continue to work for rendering e-mails:

- `renderToStaticNodeStream`

For more information on the changes to server rendering APIs, see the working group post on Upgrading to React 18 on the server, a deep dive on the new Suspense SSR Architecture, and Shaundai Person's talk on Streaming Server Rendering with Suspense at React Conf 2021.

Updates to TypeScript definitions

If your project uses TypeScript, you will need to update your `@types/react` and `@types/react-dom` dependencies to the latest versions. The new types are safer and catch issues that used to be ignored by the type checker. The most notable change is that the `children` prop now needs to be listed explicitly when defining props, for example:

```
interface MyButtonProps {
  color: string;
  children?: React.ReactNode;}
```

See the React 18 typings pull request for a full list of type-only changes. It links to example fixes in library types so you can see how to adjust your code. You can use the automated migration script to help port your application code to the new and safer typings faster.

If you find a bug in the typings, please file an issue in the DefinitelyTyped repo.

Automatic Batching

React 18 adds out-of-the-box performance improvements by doing more batching by default. Batching is when React groups multiple state updates into a single re-render for better performance. Before React 18, we only batched updates inside React event handlers. Updates inside of promises, setTimeout, native event handlers, or any other event were not batched in React by default:

```
// Before React 18 only React events were batched

function handleClick() {
  setCount(c => c + 1);
  setFlag(f => !f);
  // React will only re-render once at the end (that's batching!)
}

setTimeout(() => {
  setCount(c => c + 1);
  setFlag(f => !f);
  // React will render twice, once for each state update (no batching)
}, 1000);
```

Starting in React 18 with `createRoot`, all updates will be automatically batched, no matter where they originate from. This means that updates inside of timeouts, promises, native event handlers or any other event will batch the same way as updates inside of React events:

```
// After React 18 updates inside of timeouts, promises,
// native event handlers or any other event are batched.

function handleClick() {
  setCount(c => c + 1);
  setFlag(f => !f);
  // React will only re-render once at the end (that's batching!)
}

setTimeout(() => {
  setCount(c => c + 1);
  setFlag(f => !f);
  // React will only re-render once at the end (that's batching!)
}, 1000);
```

This is a breaking change, but we expect this to result in less work rendering, and therefore better performance in your applications. To opt-out of automatic batching, you can use `flushSync`:

```
import { flushSync } from 'react-dom';

function handleClick() {
  flushSync(() => {
    setCounter(c => c + 1);
  });
  // React has updated the DOM by now
  flushSync(() => {
    setFlag(f => !f);
  });
  // React has updated the DOM by now
}
```

For more information, see the Automatic batching deep dive.

New APIs for Libraries

In the React 18 Working Group we worked with library maintainers to create new APIs needed to support concurrent rendering for use cases specific to their use case in areas like styles, and external stores. To support React 18, some libraries may need to switch to one of the following APIs:

- `useSyncExternalStore` is a new hook that allows external stores to support concurrent reads by forcing updates to the store to be synchronous. This new API is recommended for any library that integrates with state external to React. For more information, see the useSyncExternalStore overview post and useSyncExternalStore API details.
- `useInsertionEffect` is a new hook that allows CSS-in-JS libraries to address performance issues of injecting styles in render. Unless you've already built a CSS-in-JS library we don't expect you to ever use this. This hook will run after the DOM is mutated, but before layout effects read the new layout. This solves an issue that already exists in React 17 and below, but is even more important in React 18 because React yields to the browser during concurrent rendering, giving it a chance to recalculate layout. For more information, see the Library Upgrade Guide for `<style>`.

React 18 also introduces new APIs for concurrent rendering such as `startTransition`, `useDeferredValue` and `useId`, which we share more about in the release post.

Updates to Strict Mode

In the future, we'd like to add a feature that allows React to add and remove sections of the UI while preserving state. For example, when a user tabs away from a screen and back, React should be able to immediately show the previous screen. To do this, React would unmount and remount trees using the same component state as before.

This feature will give React better performance out-of-the-box, but requires components to be resilient to effects being mounted and destroyed multiple times. Most effects will work without any changes, but some effects assume they are only mounted or destroyed once.

To help surface these issues, React 18 introduces a new development-only check to Strict Mode. This new check will automatically unmount and remount every component, whenever a component mounts for the first time, restoring the previous state on the second mount.

Before this change, React would mount the component and create the effects:

```
* React mounts the component.
    * Layout effects are created.
    * Effect effects are created.
```

With Strict Mode in React 18, React will simulate unmounting and remounting the component in development mode:

```
* React mounts the component.
    * Layout effects are created.
    * Effect effects are created.
* React simulates unmounting the component.
    * Layout effects are destroyed.
    * Effects are destroyed.
* React simulates mounting the component with the previous state.
    * Layout effect setup code runs
    * Effect setup code runs
```

For more information, see the Working Group posts for Adding Reusable State to StrictMode and How to support Reusable State in Effects.

Configuring Your Testing Environment

When you first update your tests to use `createRoot`, you may see this warning in your test console:

The current testing environment is not configured to support act(...)

To fix this, set `globalThis.IS_REACT_ACT_ENVIRONMENT` to `true` before running your test:

```
// In your test setup file
globalThis.IS_REACT_ACT_ENVIRONMENT = true;
```

The purpose of the flag is to tell React that it's running in a unit test-like environment. React will log helpful warnings if you forget to wrap an update with `act`.

You can also set the flag to `false` to tell React that `act` isn't needed. This can be useful for end-to-end tests that simulate a full browser environment.

Eventually, we expect testing libraries will configure this for you automatically. For example, the next version of React Testing Library has built-in support for React 18 without any additional configuration.

More background on the the `act` testing API and related changes is available in the working group.

Dropping Support for Internet Explorer

In this release, React is dropping support for Internet Explorer, which is going out of support on June 15, 2022. We're making this change now because new features introduced in React 18 are built using modern browser features such as microtasks which cannot be adequately polyfilled in IE.

If you need to support Internet Explorer we recommend you stay with React 17.

Deprecations

- `react-dom`: `ReactDOM.render` has been deprecated. Using it will warn and run your app in React 17 mode.
- `react-dom`: `ReactDOM.hydrate` has been deprecated. Using it will warn and run your app in React 17 mode.
- `react-dom`: `ReactDOM.unmountComponentAtNode` has been deprecated.
- `react-dom`: `ReactDOM.renderSubtreeIntoContainer` has been deprecated.
- `react-dom/server`: `ReactDOMServer.renderToNodeStream` has been deprecated.

Other Breaking Changes

- **Consistent useEffect timing**: React now always synchronously flushes effect functions if the update was triggered during a discrete user input event such as a click or a keydown event. Previously, the behavior wasn't always predictable or consistent.
- **Stricter hydration errors**: Hydration mismatches due to missing or extra text content are now treated like errors instead of warnings. React will no longer attempt to "patch up" individual nodes by inserting or deleting a node on the client in an attempt to match the server markup, and will revert to client rendering up to the closest `<Suspense>` boundary in the tree. This ensures the hydrated tree is consistent and avoids potential privacy and security holes that can be caused by hydration mismatches.
- **Suspense trees are always consistent:** If a component suspends before it's fully added to the tree, React will not add it to the tree in an incomplete state or fire its

effects. Instead, React will throw away the new tree completely, wait for the asynchronous operation to finish, and then retry rendering again from scratch. React will render the retry attempt concurrently, and without blocking the browser.

- **Layout Effects with Suspense**: When a tree re-suspends and reverts to a fallback, React will now clean up layout effects, and then re-create them when the content inside the boundary is shown again. This fixes an issue which prevented component libraries from correctly measuring layout when used with Suspense.
- **New JS Environment Requirements**: React now depends on modern browsers features including `Promise`, `Symbol`, and `Object.assign`. If you support older browsers and devices such as Internet Explorer which do not provide modern browser features natively or have non-compliant implementations, consider including a global polyfill in your bundled application.

Other Notable Changes

React

- **Components can now render `undefined`:** React no longer warns if you return `undefined` from a component. This makes the allowed component return values consistent with values that are allowed in the middle of a component tree. We suggest to use a linter to prevent mistakes like forgetting a `return` statement before JSX.
- **In tests, `act` warnings are now opt-in:** If you're running end-to-end tests, the `act` warnings are unnecessary. We've introduced an opt-in mechanism so you can enable them only for unit tests where they are useful and beneficial.
- **No warning about `setState` on unmounted components:** Previously, React warned about memory leaks when you call `setState` on an unmounted component. This warning was added for subscriptions, but people primarily run into it in scenarios where setting state is fine, and workarounds make the code worse. We've removed this warning.
- **No suppression of console logs:** When you use Strict Mode, React renders each component twice to help you find unexpected side effects. In React 17, we've suppressed console logs for one of the two renders to make the logs easier to read. In response to community feedback about this being confusing, we've removed the suppression. Instead, if you have React DevTools installed, the second log's renders will be displayed in grey, and there will be an option (off by default) to suppress them completely.
- **Improved memory usage:** React now cleans up more internal fields on unmount, making the impact from unfixed memory leaks that may exist in your application code less severe.

React DOM Server

- **`renderToString`:** Will no longer error when suspending on the server. Instead, it will emit the fallback HTML for the closest `<Suspense>` boundary and then

retry rendering the same content on the client. It is still recommended that you switch to a streaming API
like `renderToPipeableStream` or `renderToReadableStream` instead.

- **`renderToStaticMarkup`:** Will no longer error when suspending on the server. Instead, it will emit the fallback HTML for the closest `<Suspense>` boundary.

How to Upgrade to React 18

March 08, 2022 by Rick Hanlon

As we shared in the release post, React 18 introduces features powered by our new concurrent renderer, with a gradual adoption strategy for existing applications. In this post, we will guide you through the steps for upgrading to React 18.

Please report any issues you encounter while upgrading to React 18.

Note for React Native users: React 18 will ship in a future version of React Native. This is because React 18 relies on the New React Native Architecture to benefit from the new capabilities presented in this blogpost. For more information, see the React Conf keynote here.

Installing

To install the latest version of React:

```
npm install react react-dom
```

Or if you're using yarn:

```
yarn add react react-dom
```

Updates to Client Rendering APIs

When you first install React 18, you will see a warning in the console:

ReactDOM.render is no longer supported in React 18. Use createRoot instead. Until you switch to the new API, your app will behave as if it's running React 17. Learn more: https://reactjs.org/link/switch-to-createroot

React 18 introduces a new root API which provides better ergonomics for managing roots. The new root API also enables the new concurrent renderer, which allows you to opt-into concurrent features.

```
// Before
import { render } from 'react-dom';
const container = document.getElementById('app');
render(<App tab="home" />, container);

// After
import { createRoot } from 'react-dom/client';
const container = document.getElementById('app');
const root = createRoot(container); // createRoot(container!) if you use TypeScript
root.render(<App tab="home" />);
```

We've also changed `unmountComponentAtNode` to `root.unmount`:

```
// Before
unmountComponentAtNode(container);

// After
root.unmount();
```

We've also removed the callback from render, since it usually does not have the expected result when using Suspense:

```
// Before
const container = document.getElementById('app');
render(<App tab="home" />, container, () => {
  console.log('rendered');
});

// After
function AppWithCallbackAfterRender() {
  useEffect(() => {
    console.log('rendered');
  });

  return <App tab="home" />
}

const container = document.getElementById('app');
const root = createRoot(container);
root.render(<AppWithCallbackAfterRender />);
```

Note:

There is no one-to-one replacement for the old render callback API — it depends on your use case. See the working group post for Replacing render with createRoot for more information.

Finally, if your app uses server-side rendering with hydration, upgrade `hydrate` to `hydrateRoot`:

```
// Before
import { hydrate } from 'react-dom';
const container = document.getElementById('app');
hydrate(<App tab="home" />, container);

// After
import { hydrateRoot } from 'react-dom/client';
const container = document.getElementById('app');
const root = hydrateRoot(container, <App tab="home" />);
```

```
// Unlike with createRoot, you don't need a separate root.render() call
here.
```

For more information, see the working group discussion here.

Note

If your app doesn't work after upgrading, check whether it's wrapped in `<StrictMode>`. Strict Mode has gotten stricter in React 18, and not all your components may be resilient to the new checks it adds in development mode. If removing Strict Mode fixes your app, you can remove it during the upgrade, and then add it back (either at the top or for a part of the tree) after you fix the issues that it's pointing out.

Updates to Server Rendering APIs

In this release, we're revamping our `react-dom/server` APIs to fully support Suspense on the server and Streaming SSR. As part of these changes, we're deprecating the old Node streaming API, which does not support incremental Suspense streaming on the server.

Using this API will now warn:

- `renderToNodeStream`: **Deprecated** ⛔

Instead, for streaming in Node environments, use:

- `renderToPipeableStream`: **New** ✨

We're also introducing a new API to support streaming SSR with Suspense for modern edge runtime environments, such as Deno and Cloudflare workers:

- `renderToReadableStream`: **New** ✨

The following APIs will continue working, but with limited support for Suspense:

- `renderToString`: **Limited** ⚠
- `renderToStaticMarkup`: **Limited** ⚠

Finally, this API will continue to work for rendering e-mails:

- `renderToStaticNodeStream`

For more information on the changes to server rendering APIs, see the working group post on Upgrading to React 18 on the server, a deep dive on the new Suspense SSR Architecture, and Shaundai Person's talk on Streaming Server Rendering with Suspense at React Conf 2021.

Updates to TypeScript definitions

If your project uses TypeScript, you will need to update your `@types/react` and `@types/react-dom` dependencies to the latest versions. The new types are safer and catch issues that used to be ignored by the type checker. The most notable change is that the `children` prop now needs to be listed explicitly when defining props, for example:

```
interface MyButtonProps {
  color: string;
  children?: React.ReactNode;}
```

See the React 18 typings pull request for a full list of type-only changes. It links to example fixes in library types so you can see how to adjust your code. You can use the automated migration script to help port your application code to the new and safer typings faster.

If you find a bug in the typings, please file an issue in the DefinitelyTyped repo.

Automatic Batching

React 18 adds out-of-the-box performance improvements by doing more batching by default. Batching is when React groups multiple state updates into a single re-render for better performance. Before React 18, we only batched updates inside React event handlers. Updates inside of promises, setTimeout, native event handlers, or any other event were not batched in React by default:

```
// Before React 18 only React events were batched

function handleClick() {
  setCount(c => c + 1);
  setFlag(f => !f);
  // React will only re-render once at the end (that's batching!)
}

setTimeout(() => {
  setCount(c => c + 1);
  setFlag(f => !f);
  // React will render twice, once for each state update (no batching)
}, 1000);
```

Starting in React 18 with `createRoot`, all updates will be automatically batched, no matter where they originate from. This means that updates inside of timeouts, promises, native event handlers or any other event will batch the same way as updates inside of React events:

```
// After React 18 updates inside of timeouts, promises,
// native event handlers or any other event are batched.

function handleClick() {
  setCount(c => c + 1);
  setFlag(f => !f);
  // React will only re-render once at the end (that's batching!)
}

setTimeout(() => {
  setCount(c => c + 1);
  setFlag(f => !f);
  // React will only re-render once at the end (that's batching!)
}, 1000);
```

This is a breaking change, but we expect this to result in less work rendering, and therefore better performance in your applications. To opt-out of automatic batching, you can use `flushSync`:

```
import { flushSync } from 'react-dom';

function handleClick() {
  flushSync(() => {
    setCounter(c => c + 1);
  });
  // React has updated the DOM by now
  flushSync(() => {
    setFlag(f => !f);
  });
  // React has updated the DOM by now
}
```

For more information, see the Automatic batching deep dive.

New APIs for Libraries

In the React 18 Working Group we worked with library maintainers to create new APIs needed to support concurrent rendering for use cases specific to their use case in areas like styles, and external stores. To support React 18, some libraries may need to switch to one of the following APIs:

- `useSyncExternalStore` is a new hook that allows external stores to support concurrent reads by forcing updates to the store to be synchronous. This new API is recommended for any library that integrates with state external to React. For more information, see the useSyncExternalStore overview post and useSyncExternalStore API details.
- `useInsertionEffect` is a new hook that allows CSS-in-JS libraries to address performance issues of injecting styles in render. Unless you've already built a CSS-in-JS library we don't expect you to ever use this. This hook will run after the DOM is mutated, but before layout effects read the new layout. This solves an issue that already exists in React 17 and below, but is even more important in React 18 because React yields to the browser during concurrent rendering, giving it a chance to recalculate layout. For more information, see the Library Upgrade Guide for `<style>`.

React 18 also introduces new APIs for concurrent rendering such as `startTransition`, `useDeferredValue` and `useId`, which we share more about in the release post.

Updates to Strict Mode

In the future, we'd like to add a feature that allows React to add and remove sections of the UI while preserving state. For example, when a user tabs away from a screen and back, React should be able to immediately show the previous screen. To do this, React would unmount and remount trees using the same component state as before.

This feature will give React better performance out-of-the-box, but requires components to be resilient to effects being mounted and destroyed multiple times. Most effects will work without any changes, but some effects assume they are only mounted or destroyed once.

To help surface these issues, React 18 introduces a new development-only check to Strict Mode. This new check will automatically unmount and remount every component, whenever a component mounts for the first time, restoring the previous state on the second mount.

Before this change, React would mount the component and create the effects:

```
* React mounts the component.
    * Layout effects are created.
    * Effect effects are created.
```

With Strict Mode in React 18, React will simulate unmounting and remounting the component in development mode:

```
* React mounts the component.
    * Layout effects are created.
    * Effect effects are created.
* React simulates unmounting the component.
    * Layout effects are destroyed.
    * Effects are destroyed.
* React simulates mounting the component with the previous state.
    * Layout effect setup code runs
    * Effect setup code runs
```

For more information, see the Working Group posts for Adding Reusable State to StrictMode and How to support Reusable State in Effects.

Configuring Your Testing Environment

When you first update your tests to use `createRoot`, you may see this warning in your test console:

The current testing environment is not configured to support act(...)

To fix this, set `globalThis.IS_REACT_ACT_ENVIRONMENT` to `true` before running your test:

```
// In your test setup file
globalThis.IS_REACT_ACT_ENVIRONMENT = true;
```

The purpose of the flag is to tell React that it's running in a unit test-like environment. React will log helpful warnings if you forget to wrap an update with `act`.

You can also set the flag to `false` to tell React that `act` isn't needed. This can be useful for end-to-end tests that simulate a full browser environment.

Eventually, we expect testing libraries will configure this for you automatically. For example, the next version of React Testing Library has built-in support for React 18 without any additional configuration.

More background on the the `act` testing API and related changes is available in the working group.

Dropping Support for Internet Explorer

In this release, React is dropping support for Internet Explorer, which is going out of support on June 15, 2022. We're making this change now because new features introduced in React 18 are built using modern browser features such as microtasks which cannot be adequately polyfilled in IE.

If you need to support Internet Explorer we recommend you stay with React 17.

Deprecations

- `react-dom`: `ReactDOM.render` has been deprecated. Using it will warn and run your app in React 17 mode.
- `react-dom`: `ReactDOM.hydrate` has been deprecated. Using it will warn and run your app in React 17 mode.
- `react-dom`: `ReactDOM.unmountComponentAtNode` has been deprecated.
- `react-dom`: `ReactDOM.renderSubtreeIntoContainer` has been deprecated.
- `react-dom/server`: `ReactDOMServer.renderToNodeStream` has been deprecated.

Other Breaking Changes

- **Consistent useEffect timing**: React now always synchronously flushes effect functions if the update was triggered during a discrete user input event such as a click or a keydown event. Previously, the behavior wasn't always predictable or consistent.
- **Stricter hydration errors**: Hydration mismatches due to missing or extra text content are now treated like errors instead of warnings. React will no longer attempt to "patch up" individual nodes by inserting or deleting a node on the client in an attempt to match the server markup, and will revert to client rendering up to the closest `<Suspense>` boundary in the tree. This ensures the hydrated tree is consistent and avoids potential privacy and security holes that can be caused by hydration mismatches.
- **Suspense trees are always consistent:** If a component suspends before it's fully added to the tree, React will not add it to the tree in an incomplete state or fire its

effects. Instead, React will throw away the new tree completely, wait for the asynchronous operation to finish, and then retry rendering again from scratch. React will render the retry attempt concurrently, and without blocking the browser.

- **Layout Effects with Suspense**: When a tree re-suspends and reverts to a fallback, React will now clean up layout effects, and then re-create them when the content inside the boundary is shown again. This fixes an issue which prevented component libraries from correctly measuring layout when used with Suspense.
- **New JS Environment Requirements**: React now depends on modern browsers features including `Promise`, `Symbol`, and `Object.assign`. If you support older browsers and devices such as Internet Explorer which do not provide modern browser features natively or have non-compliant implementations, consider including a global polyfill in your bundled application.

Other Notable Changes

React

- **Components can now render `undefined`:** React no longer warns if you return `undefined` from a component. This makes the allowed component return values consistent with values that are allowed in the middle of a component tree. We suggest to use a linter to prevent mistakes like forgetting a `return` statement before JSX.
- **In tests, `act` warnings are now opt-in:** If you're running end-to-end tests, the `act` warnings are unnecessary. We've introduced an opt-in mechanism so you can enable them only for unit tests where they are useful and beneficial.
- **No warning about `setState` on unmounted components:** Previously, React warned about memory leaks when you call `setState` on an unmounted component. This warning was added for subscriptions, but people primarily run into it in scenarios where setting state is fine, and workarounds make the code worse. We've removed this warning.
- **No suppression of console logs:** When you use Strict Mode, React renders each component twice to help you find unexpected side effects. In React 17, we've suppressed console logs for one of the two renders to make the logs easier to read. In response to community feedback about this being confusing, we've removed the suppression. Instead, if you have React DevTools installed, the second log's renders will be displayed in grey, and there will be an option (off by default) to suppress them completely.
- **Improved memory usage:** React now cleans up more internal fields on unmount, making the impact from unfixed memory leaks that may exist in your application code less severe.

React DOM Server

- **`renderToString`:** Will no longer error when suspending on the server. Instead, it will emit the fallback HTML for the closest `<Suspense>` boundary and then

retry rendering the same content on the client. It is still recommended that you switch to a streaming API
like `renderToPipeableStream` or `renderToReadableStream` instead.

- **`renderToStaticMarkup`:** Will no longer error when suspending on the server. Instead, it will emit the fallback HTML for the closest `<Suspense>` boundary.

How to Upgrade to React 18

March 08, 2022 by Rick Hanlon

As we shared in the release post, React 18 introduces features powered by our new concurrent renderer, with a gradual adoption strategy for existing applications. In this post, we will guide you through the steps for upgrading to React 18.

Please report any issues you encounter while upgrading to React 18.

Note for React Native users: React 18 will ship in a future version of React Native. This is because React 18 relies on the New React Native Architecture to benefit from the new capabilities presented in this blogpost. For more information, see the React Conf keynote here.

Installing

To install the latest version of React:

```
npm install react react-dom
```

Or if you're using yarn:

```
yarn add react react-dom
```

Updates to Client Rendering APIs

When you first install React 18, you will see a warning in the console:

ReactDOM.render is no longer supported in React 18. Use createRoot instead. Until you switch to the new API, your app will behave as if it's running React 17. Learn more: https://reactjs.org/link/switch-to-createroot

React 18 introduces a new root API which provides better ergonomics for managing roots. The new root API also enables the new concurrent renderer, which allows you to opt-into concurrent features.

```
// Before
import { render } from 'react-dom';
const container = document.getElementById('app');
render(<App tab="home" />, container);

// After
import { createRoot } from 'react-dom/client';
const container = document.getElementById('app');
const root = createRoot(container); // createRoot(container!) if you use TypeScript
root.render(<App tab="home" />);
```

We've also changed `unmountComponentAtNode` to `root.unmount`:

```
// Before
unmountComponentAtNode(container);

// After
root.unmount();
```

We've also removed the callback from render, since it usually does not have the expected result when using Suspense:

```
// Before
const container = document.getElementById('app');
render(<App tab="home" />, container, () => {
  console.log('rendered');
});

// After
function AppWithCallbackAfterRender() {
  useEffect(() => {
    console.log('rendered');
  });

  return <App tab="home" />
}

const container = document.getElementById('app');
const root = createRoot(container);
root.render(<AppWithCallbackAfterRender />);
```

Note:

There is no one-to-one replacement for the old render callback API — it depends on your use case. See the working group post for Replacing render with createRoot for more information.

Finally, if your app uses server-side rendering with hydration, upgrade `hydrate` to `hydrateRoot`:

```
// Before
import { hydrate } from 'react-dom';
const container = document.getElementById('app');
hydrate(<App tab="home" />, container);

// After
import { hydrateRoot } from 'react-dom/client';
const container = document.getElementById('app');
const root = hydrateRoot(container, <App tab="home" />);
```

```
// Unlike with createRoot, you don't need a separate root.render() call
here.
```

For more information, see the working group discussion here.

Note

If your app doesn't work after upgrading, check whether it's wrapped in `<StrictMode>`. Strict Mode has gotten stricter in React 18, and not all your components may be resilient to the new checks it adds in development mode. If removing Strict Mode fixes your app, you can remove it during the upgrade, and then add it back (either at the top or for a part of the tree) after you fix the issues that it's pointing out.

Updates to Server Rendering APIs

In this release, we're revamping our `react-dom/server` APIs to fully support Suspense on the server and Streaming SSR. As part of these changes, we're deprecating the old Node streaming API, which does not support incremental Suspense streaming on the server.

Using this API will now warn:

- `renderToNodeStream`: **Deprecated** ⛔

Instead, for streaming in Node environments, use:

- `renderToPipeableStream`: **New** ✨

We're also introducing a new API to support streaming SSR with Suspense for modern edge runtime environments, such as Deno and Cloudflare workers:

- `renderToReadableStream`: **New** ✨

The following APIs will continue working, but with limited support for Suspense:

- `renderToString`: **Limited** ⚠
- `renderToStaticMarkup`: **Limited** ⚠

Finally, this API will continue to work for rendering e-mails:

- `renderToStaticNodeStream`

For more information on the changes to server rendering APIs, see the working group post on Upgrading to React 18 on the server, a deep dive on the new Suspense SSR Architecture, and Shaundai Person's talk on Streaming Server Rendering with Suspense at React Conf 2021.

Updates to TypeScript definitions

If your project uses TypeScript, you will need to update your `@types/react` and `@types/react-dom` dependencies to the latest versions. The new types are safer and catch issues that used to be ignored by the type checker. The most notable change is that the `children` prop now needs to be listed explicitly when defining props, for example:

```
interface MyButtonProps {
  color: string;
  children?: React.ReactNode;}
```

See the React 18 typings pull request for a full list of type-only changes. It links to example fixes in library types so you can see how to adjust your code. You can use the automated migration script to help port your application code to the new and safer typings faster.

If you find a bug in the typings, please file an issue in the DefinitelyTyped repo.

Automatic Batching

React 18 adds out-of-the-box performance improvements by doing more batching by default. Batching is when React groups multiple state updates into a single re-render for better performance. Before React 18, we only batched updates inside React event handlers. Updates inside of promises, setTimeout, native event handlers, or any other event were not batched in React by default:

```
// Before React 18 only React events were batched

function handleClick() {
  setCount(c => c + 1);
  setFlag(f => !f);
  // React will only re-render once at the end (that's batching!)
}

setTimeout(() => {
  setCount(c => c + 1);
  setFlag(f => !f);
  // React will render twice, once for each state update (no batching)
}, 1000);
```

Starting in React 18 with `createRoot`, all updates will be automatically batched, no matter where they originate from. This means that updates inside of timeouts, promises, native event handlers or any other event will batch the same way as updates inside of React events:

```
// After React 18 updates inside of timeouts, promises,
// native event handlers or any other event are batched.

function handleClick() {
  setCount(c => c + 1);
  setFlag(f => !f);
  // React will only re-render once at the end (that's batching!)
}

setTimeout(() => {
  setCount(c => c + 1);
  setFlag(f => !f);
  // React will only re-render once at the end (that's batching!)
}, 1000);
```

This is a breaking change, but we expect this to result in less work rendering, and therefore better performance in your applications. To opt-out of automatic batching, you can use `flushSync`:

```
import { flushSync } from 'react-dom';

function handleClick() {
  flushSync(() => {
    setCounter(c => c + 1);
  });
  // React has updated the DOM by now
  flushSync(() => {
    setFlag(f => !f);
  });
  // React has updated the DOM by now
}
```

For more information, see the Automatic batching deep dive.

New APIs for Libraries

In the React 18 Working Group we worked with library maintainers to create new APIs needed to support concurrent rendering for use cases specific to their use case in areas like styles, and external stores. To support React 18, some libraries may need to switch to one of the following APIs:

- `useSyncExternalStore` is a new hook that allows external stores to support concurrent reads by forcing updates to the store to be synchronous. This new API is recommended for any library that integrates with state external to React. For more information, see the useSyncExternalStore overview post and useSyncExternalStore API details.
- `useInsertionEffect` is a new hook that allows CSS-in-JS libraries to address performance issues of injecting styles in render. Unless you've already built a CSS-in-JS library we don't expect you to ever use this. This hook will run after the DOM is mutated, but before layout effects read the new layout. This solves an issue that already exists in React 17 and below, but is even more important in React 18 because React yields to the browser during concurrent rendering, giving it a chance to recalculate layout. For more information, see the Library Upgrade Guide for `<style>`.

React 18 also introduces new APIs for concurrent rendering such as `startTransition`, `useDeferredValue` and `useId`, which we share more about in the release post.

Updates to Strict Mode

In the future, we'd like to add a feature that allows React to add and remove sections of the UI while preserving state. For example, when a user tabs away from a screen and back, React should be able to immediately show the previous screen. To do this, React would unmount and remount trees using the same component state as before.

This feature will give React better performance out-of-the-box, but requires components to be resilient to effects being mounted and destroyed multiple times. Most effects will work without any changes, but some effects assume they are only mounted or destroyed once.

To help surface these issues, React 18 introduces a new development-only check to Strict Mode. This new check will automatically unmount and remount every component, whenever a component mounts for the first time, restoring the previous state on the second mount.

Before this change, React would mount the component and create the effects:

```
* React mounts the component.
    * Layout effects are created.
    * Effect effects are created.
```

With Strict Mode in React 18, React will simulate unmounting and remounting the component in development mode:

```
* React mounts the component.
    * Layout effects are created.
    * Effect effects are created.
* React simulates unmounting the component.
    * Layout effects are destroyed.
    * Effects are destroyed.
* React simulates mounting the component with the previous state.
    * Layout effect setup code runs
    * Effect setup code runs
```

For more information, see the Working Group posts for Adding Reusable State to StrictMode and How to support Reusable State in Effects.

Configuring Your Testing Environment

When you first update your tests to use `createRoot`, you may see this warning in your test console:

The current testing environment is not configured to support act(...)

To fix this, set `globalThis.IS_REACT_ACT_ENVIRONMENT` to `true` before running your test:

```
// In your test setup file
globalThis.IS_REACT_ACT_ENVIRONMENT = true;
```

The purpose of the flag is to tell React that it's running in a unit test-like environment. React will log helpful warnings if you forget to wrap an update with `act`.

You can also set the flag to `false` to tell React that `act` isn't needed. This can be useful for end-to-end tests that simulate a full browser environment.

Eventually, we expect testing libraries will configure this for you automatically. For example, the next version of React Testing Library has built-in support for React 18 without any additional configuration.

More background on the the `act` testing API and related changes is available in the working group.

Dropping Support for Internet Explorer

In this release, React is dropping support for Internet Explorer, which is going out of support on June 15, 2022. We're making this change now because new features introduced in React 18 are built using modern browser features such as microtasks which cannot be adequately polyfilled in IE.

If you need to support Internet Explorer we recommend you stay with React 17.

Deprecations

- `react-dom`: `ReactDOM.render` has been deprecated. Using it will warn and run your app in React 17 mode.
- `react-dom`: `ReactDOM.hydrate` has been deprecated. Using it will warn and run your app in React 17 mode.
- `react-dom`: `ReactDOM.unmountComponentAtNode` has been deprecated.
- `react-dom`: `ReactDOM.renderSubtreeIntoContainer` has been deprecated.
- `react-dom/server`: `ReactDOMServer.renderToNodeStream` has been deprecated.

Other Breaking Changes

- **Consistent useEffect timing**: React now always synchronously flushes effect functions if the update was triggered during a discrete user input event such as a click or a keydown event. Previously, the behavior wasn't always predictable or consistent.
- **Stricter hydration errors**: Hydration mismatches due to missing or extra text content are now treated like errors instead of warnings. React will no longer attempt to "patch up" individual nodes by inserting or deleting a node on the client in an attempt to match the server markup, and will revert to client rendering up to the closest `<Suspense>` boundary in the tree. This ensures the hydrated tree is consistent and avoids potential privacy and security holes that can be caused by hydration mismatches.
- **Suspense trees are always consistent:** If a component suspends before it's fully added to the tree, React will not add it to the tree in an incomplete state or fire its

effects. Instead, React will throw away the new tree completely, wait for the asynchronous operation to finish, and then retry rendering again from scratch. React will render the retry attempt concurrently, and without blocking the browser.

- **Layout Effects with Suspense**: When a tree re-suspends and reverts to a fallback, React will now clean up layout effects, and then re-create them when the content inside the boundary is shown again. This fixes an issue which prevented component libraries from correctly measuring layout when used with Suspense.
- **New JS Environment Requirements**: React now depends on modern browsers features including `Promise`, `Symbol`, and `Object.assign`. If you support older browsers and devices such as Internet Explorer which do not provide modern browser features natively or have non-compliant implementations, consider including a global polyfill in your bundled application.

Other Notable Changes

React

- **Components can now render `undefined`:** React no longer warns if you return `undefined` from a component. This makes the allowed component return values consistent with values that are allowed in the middle of a component tree. We suggest to use a linter to prevent mistakes like forgetting a `return` statement before JSX.
- **In tests, `act` warnings are now opt-in:** If you're running end-to-end tests, the `act` warnings are unnecessary. We've introduced an opt-in mechanism so you can enable them only for unit tests where they are useful and beneficial.
- **No warning about `setState` on unmounted components:** Previously, React warned about memory leaks when you call `setState` on an unmounted component. This warning was added for subscriptions, but people primarily run into it in scenarios where setting state is fine, and workarounds make the code worse. We've removed this warning.
- **No suppression of console logs:** When you use Strict Mode, React renders each component twice to help you find unexpected side effects. In React 17, we've suppressed console logs for one of the two renders to make the logs easier to read. In response to community feedback about this being confusing, we've removed the suppression. Instead, if you have React DevTools installed, the second log's renders will be displayed in grey, and there will be an option (off by default) to suppress them completely.
- **Improved memory usage:** React now cleans up more internal fields on unmount, making the impact from unfixed memory leaks that may exist in your application code less severe.

React DOM Server

- **`renderToString`:** Will no longer error when suspending on the server. Instead, it will emit the fallback HTML for the closest `<Suspense>` boundary and then

retry rendering the same content on the client. It is still recommended that you switch to a streaming API
like `renderToPipeableStream` or `renderToReadableStream` instead.

- **`renderToStaticMarkup`:** Will no longer error when suspending on the server. Instead, it will emit the fallback HTML for the closest `<Suspense>` boundary.

How to Upgrade to React 18

March 08, 2022 by Rick Hanlon

As we shared in the release post, React 18 introduces features powered by our new concurrent renderer, with a gradual adoption strategy for existing applications. In this post, we will guide you through the steps for upgrading to React 18.

Please report any issues you encounter while upgrading to React 18.

Note for React Native users: React 18 will ship in a future version of React Native. This is because React 18 relies on the New React Native Architecture to benefit from the new capabilities presented in this blogpost. For more information, see the React Conf keynote here.

Installing

To install the latest version of React:

```
npm install react react-dom
```

Or if you're using yarn:

```
yarn add react react-dom
```

Updates to Client Rendering APIs

When you first install React 18, you will see a warning in the console:

ReactDOM.render is no longer supported in React 18. Use createRoot instead. Until you switch to the new API, your app will behave as if it's running React 17. Learn more: https://reactjs.org/link/switch-to-createroot

React 18 introduces a new root API which provides better ergonomics for managing roots. The new root API also enables the new concurrent renderer, which allows you to opt-into concurrent features.

```
// Before
import { render } from 'react-dom';
const container = document.getElementById('app');
render(<App tab="home" />, container);

// After
import { createRoot } from 'react-dom/client';
const container = document.getElementById('app');
const root = createRoot(container); // createRoot(container!) if you use TypeScript
root.render(<App tab="home" />);
```

We've also changed `unmountComponentAtNode` to `root.unmount`:

```
// Before
unmountComponentAtNode(container);

// After
root.unmount();
```

We've also removed the callback from render, since it usually does not have the expected result when using Suspense:

```
// Before
const container = document.getElementById('app');
render(<App tab="home" />, container, () => {
  console.log('rendered');
});

// After
function AppWithCallbackAfterRender() {
  useEffect(() => {
    console.log('rendered');
  });

  return <App tab="home" />
}

const container = document.getElementById('app');
const root = createRoot(container);
root.render(<AppWithCallbackAfterRender />);
```

Note:

There is no one-to-one replacement for the old render callback API — it depends on your use case. See the working group post for Replacing render with createRoot for more information.

Finally, if your app uses server-side rendering with hydration, upgrade `hydrate` to `hydrateRoot`:

```
// Before
import { hydrate } from 'react-dom';
const container = document.getElementById('app');
hydrate(<App tab="home" />, container);

// After
import { hydrateRoot } from 'react-dom/client';
const container = document.getElementById('app');
const root = hydrateRoot(container, <App tab="home" />);
```

```
// Unlike with createRoot, you don't need a separate root.render() call
here.
```

For more information, see the working group discussion here.

Note

If your app doesn't work after upgrading, check whether it's wrapped in `<StrictMode>`. Strict Mode has gotten stricter in React 18, and not all your components may be resilient to the new checks it adds in development mode. If removing Strict Mode fixes your app, you can remove it during the upgrade, and then add it back (either at the top or for a part of the tree) after you fix the issues that it's pointing out.

Updates to Server Rendering APIs

In this release, we're revamping our `react-dom/server` APIs to fully support Suspense on the server and Streaming SSR. As part of these changes, we're deprecating the old Node streaming API, which does not support incremental Suspense streaming on the server.

Using this API will now warn:

- `renderToNodeStream`: **Deprecated** ⛔

Instead, for streaming in Node environments, use:

- `renderToPipeableStream`: **New** ✨

We're also introducing a new API to support streaming SSR with Suspense for modern edge runtime environments, such as Deno and Cloudflare workers:

- `renderToReadableStream`: **New** ✨

The following APIs will continue working, but with limited support for Suspense:

- `renderToString`: **Limited** ⚠
- `renderToStaticMarkup`: **Limited** ⚠

Finally, this API will continue to work for rendering e-mails:

- `renderToStaticNodeStream`

For more information on the changes to server rendering APIs, see the working group post on Upgrading to React 18 on the server, a deep dive on the new Suspense SSR Architecture, and Shaundai Person's talk on Streaming Server Rendering with Suspense at React Conf 2021.

Updates to TypeScript definitions

If your project uses TypeScript, you will need to update your `@types/react` and `@types/react-dom` dependencies to the latest versions. The new types are safer and catch issues that used to be ignored by the type checker. The most notable change is that the `children` prop now needs to be listed explicitly when defining props, for example:

```
interface MyButtonProps {
  color: string;
  children?: React.ReactNode;}
```

See the React 18 typings pull request for a full list of type-only changes. It links to example fixes in library types so you can see how to adjust your code. You can use the automated migration script to help port your application code to the new and safer typings faster.

If you find a bug in the typings, please file an issue in the DefinitelyTyped repo.

Automatic Batching

React 18 adds out-of-the-box performance improvements by doing more batching by default. Batching is when React groups multiple state updates into a single re-render for better performance. Before React 18, we only batched updates inside React event handlers. Updates inside of promises, setTimeout, native event handlers, or any other event were not batched in React by default:

```
// Before React 18 only React events were batched

function handleClick() {
  setCount(c => c + 1);
  setFlag(f => !f);
  // React will only re-render once at the end (that's batching!)
}

setTimeout(() => {
  setCount(c => c + 1);
  setFlag(f => !f);
  // React will render twice, once for each state update (no batching)
}, 1000);
```

Starting in React 18 with `createRoot`, all updates will be automatically batched, no matter where they originate from. This means that updates inside of timeouts, promises, native event handlers or any other event will batch the same way as updates inside of React events:

```
// After React 18 updates inside of timeouts, promises,
// native event handlers or any other event are batched.

function handleClick() {
  setCount(c => c + 1);
  setFlag(f => !f);
  // React will only re-render once at the end (that's batching!)
}

setTimeout(() => {
  setCount(c => c + 1);
  setFlag(f => !f);
  // React will only re-render once at the end (that's batching!)
}, 1000);
```

This is a breaking change, but we expect this to result in less work rendering, and therefore better performance in your applications. To opt-out of automatic batching, you can use `flushSync`:

```
import { flushSync } from 'react-dom';

function handleClick() {
  flushSync(() => {
    setCounter(c => c + 1);
  });
  // React has updated the DOM by now
  flushSync(() => {
    setFlag(f => !f);
  });
  // React has updated the DOM by now
}
```

For more information, see the Automatic batching deep dive.

New APIs for Libraries

In the React 18 Working Group we worked with library maintainers to create new APIs needed to support concurrent rendering for use cases specific to their use case in areas like styles, and external stores. To support React 18, some libraries may need to switch to one of the following APIs:

- `useSyncExternalStore` is a new hook that allows external stores to support concurrent reads by forcing updates to the store to be synchronous. This new API is recommended for any library that integrates with state external to React. For more information, see the useSyncExternalStore overview post and useSyncExternalStore API details.
- `useInsertionEffect` is a new hook that allows CSS-in-JS libraries to address performance issues of injecting styles in render. Unless you've already built a CSS-in-JS library we don't expect you to ever use this. This hook will run after the DOM is mutated, but before layout effects read the new layout. This solves an issue that already exists in React 17 and below, but is even more important in React 18 because React yields to the browser during concurrent rendering, giving it a chance to recalculate layout. For more information, see the Library Upgrade Guide for `<style>`.

React 18 also introduces new APIs for concurrent rendering such as `startTransition`, `useDeferredValue` and `useId`, which we share more about in the release post.

Updates to Strict Mode

In the future, we'd like to add a feature that allows React to add and remove sections of the UI while preserving state. For example, when a user tabs away from a screen and back, React should be able to immediately show the previous screen. To do this, React would unmount and remount trees using the same component state as before.

This feature will give React better performance out-of-the-box, but requires components to be resilient to effects being mounted and destroyed multiple times. Most effects will work without any changes, but some effects assume they are only mounted or destroyed once.

To help surface these issues, React 18 introduces a new development-only check to Strict Mode. This new check will automatically unmount and remount every component, whenever a component mounts for the first time, restoring the previous state on the second mount.

Before this change, React would mount the component and create the effects:

```
* React mounts the component.
    * Layout effects are created.
    * Effect effects are created.
```

With Strict Mode in React 18, React will simulate unmounting and remounting the component in development mode:

```
* React mounts the component.
    * Layout effects are created.
    * Effect effects are created.
* React simulates unmounting the component.
    * Layout effects are destroyed.
    * Effects are destroyed.
* React simulates mounting the component with the previous state.
    * Layout effect setup code runs
    * Effect setup code runs
```

For more information, see the Working Group posts for Adding Reusable State to StrictMode and How to support Reusable State in Effects.

Configuring Your Testing Environment

When you first update your tests to use `createRoot`, you may see this warning in your test console:

The current testing environment is not configured to support act(...)

To fix this, set `globalThis.IS_REACT_ACT_ENVIRONMENT` to `true` before running your test:

```
// In your test setup file
globalThis.IS_REACT_ACT_ENVIRONMENT = true;
```

The purpose of the flag is to tell React that it's running in a unit test-like environment. React will log helpful warnings if you forget to wrap an update with `act`.

You can also set the flag to `false` to tell React that `act` isn't needed. This can be useful for end-to-end tests that simulate a full browser environment.

Eventually, we expect testing libraries will configure this for you automatically. For example, the next version of React Testing Library has built-in support for React 18 without any additional configuration.

More background on the the `act` testing API and related changes is available in the working group.

Dropping Support for Internet Explorer

In this release, React is dropping support for Internet Explorer, which is going out of support on June 15, 2022. We're making this change now because new features introduced in React 18 are built using modern browser features such as microtasks which cannot be adequately polyfilled in IE.

If you need to support Internet Explorer we recommend you stay with React 17.

Deprecations

- `react-dom`: `ReactDOM.render` has been deprecated. Using it will warn and run your app in React 17 mode.
- `react-dom`: `ReactDOM.hydrate` has been deprecated. Using it will warn and run your app in React 17 mode.
- `react-dom`: `ReactDOM.unmountComponentAtNode` has been deprecated.
- `react-dom`: `ReactDOM.renderSubtreeIntoContainer` has been deprecated.
- `react-dom/server`: `ReactDOMServer.renderToNodeStream` has been deprecated.

Other Breaking Changes

- **Consistent useEffect timing**: React now always synchronously flushes effect functions if the update was triggered during a discrete user input event such as a click or a keydown event. Previously, the behavior wasn't always predictable or consistent.
- **Stricter hydration errors**: Hydration mismatches due to missing or extra text content are now treated like errors instead of warnings. React will no longer attempt to "patch up" individual nodes by inserting or deleting a node on the client in an attempt to match the server markup, and will revert to client rendering up to the closest `<Suspense>` boundary in the tree. This ensures the hydrated tree is consistent and avoids potential privacy and security holes that can be caused by hydration mismatches.
- **Suspense trees are always consistent:** If a component suspends before it's fully added to the tree, React will not add it to the tree in an incomplete state or fire its

effects. Instead, React will throw away the new tree completely, wait for the asynchronous operation to finish, and then retry rendering again from scratch. React will render the retry attempt concurrently, and without blocking the browser.

- **Layout Effects with Suspense**: When a tree re-suspends and reverts to a fallback, React will now clean up layout effects, and then re-create them when the content inside the boundary is shown again. This fixes an issue which prevented component libraries from correctly measuring layout when used with Suspense.
- **New JS Environment Requirements**: React now depends on modern browsers features including `Promise`, `Symbol`, and `Object.assign`. If you support older browsers and devices such as Internet Explorer which do not provide modern browser features natively or have non-compliant implementations, consider including a global polyfill in your bundled application.

Other Notable Changes

React

- **Components can now render `undefined`:** React no longer warns if you return `undefined` from a component. This makes the allowed component return values consistent with values that are allowed in the middle of a component tree. We suggest to use a linter to prevent mistakes like forgetting a `return` statement before JSX.
- **In tests, `act` warnings are now opt-in:** If you're running end-to-end tests, the `act` warnings are unnecessary. We've introduced an opt-in mechanism so you can enable them only for unit tests where they are useful and beneficial.
- **No warning about `setState` on unmounted components:** Previously, React warned about memory leaks when you call `setState` on an unmounted component. This warning was added for subscriptions, but people primarily run into it in scenarios where setting state is fine, and workarounds make the code worse. We've removed this warning.
- **No suppression of console logs:** When you use Strict Mode, React renders each component twice to help you find unexpected side effects. In React 17, we've suppressed console logs for one of the two renders to make the logs easier to read. In response to community feedback about this being confusing, we've removed the suppression. Instead, if you have React DevTools installed, the second log's renders will be displayed in grey, and there will be an option (off by default) to suppress them completely.
- **Improved memory usage:** React now cleans up more internal fields on unmount, making the impact from unfixed memory leaks that may exist in your application code less severe.

React DOM Server

- **`renderToString`:** Will no longer error when suspending on the server. Instead, it will emit the fallback HTML for the closest `<Suspense>` boundary and then

retry rendering the same content on the client. It is still recommended that you switch to a streaming API
like `renderToPipeableStream` or `renderToReadableStream` instead.

- **`renderToStaticMarkup`:** Will no longer error when suspending on the server. Instead, it will emit the fallback HTML for the closest `<Suspense>` boundary.

How to Upgrade to React 18

March 08, 2022 by Rick Hanlon

As we shared in the release post, React 18 introduces features powered by our new concurrent renderer, with a gradual adoption strategy for existing applications. In this post, we will guide you through the steps for upgrading to React 18.

Please report any issues you encounter while upgrading to React 18.

Note for React Native users: React 18 will ship in a future version of React Native. This is because React 18 relies on the New React Native Architecture to benefit from the new capabilities presented in this blogpost. For more information, see the React Conf keynote here.

Installing

To install the latest version of React:

```
npm install react react-dom
```

Or if you're using yarn:

```
yarn add react react-dom
```

Updates to Client Rendering APIs

When you first install React 18, you will see a warning in the console:

ReactDOM.render is no longer supported in React 18. Use createRoot instead. Until you switch to the new API, your app will behave as if it's running React 17. Learn more: https://reactjs.org/link/switch-to-createroot

React 18 introduces a new root API which provides better ergonomics for managing roots. The new root API also enables the new concurrent renderer, which allows you to opt-into concurrent features.

```
// Before
import { render } from 'react-dom';
const container = document.getElementById('app');
render(<App tab="home" />, container);

// After
import { createRoot } from 'react-dom/client';
const container = document.getElementById('app');
const root = createRoot(container); // createRoot(container!) if you use TypeScript
root.render(<App tab="home" />);
```

We've also changed `unmountComponentAtNode` to `root.unmount`:

```
// Before
unmountComponentAtNode(container);

// After
root.unmount();
```

We've also removed the callback from render, since it usually does not have the expected result when using Suspense:

```
// Before
const container = document.getElementById('app');
render(<App tab="home" />, container, () => {
  console.log('rendered');
});

// After
function AppWithCallbackAfterRender() {
  useEffect(() => {
    console.log('rendered');
  });

  return <App tab="home" />
}

const container = document.getElementById('app');
const root = createRoot(container);
root.render(<AppWithCallbackAfterRender />);
```

Note:

There is no one-to-one replacement for the old render callback API — it depends on your use case. See the working group post for Replacing render with createRoot for more information.

Finally, if your app uses server-side rendering with hydration, upgrade `hydrate` to `hydrateRoot`:

```
// Before
import { hydrate } from 'react-dom';
const container = document.getElementById('app');
hydrate(<App tab="home" />, container);

// After
import { hydrateRoot } from 'react-dom/client';
const container = document.getElementById('app');
const root = hydrateRoot(container, <App tab="home" />);
```

```
// Unlike with createRoot, you don't need a separate root.render() call
here.
```

For more information, see the working group discussion here.

Note

If your app doesn't work after upgrading, check whether it's wrapped in `<StrictMode>`. Strict Mode has gotten stricter in React 18, and not all your components may be resilient to the new checks it adds in development mode. If removing Strict Mode fixes your app, you can remove it during the upgrade, and then add it back (either at the top or for a part of the tree) after you fix the issues that it's pointing out.

Updates to Server Rendering APIs

In this release, we're revamping our `react-dom/server` APIs to fully support Suspense on the server and Streaming SSR. As part of these changes, we're deprecating the old Node streaming API, which does not support incremental Suspense streaming on the server.

Using this API will now warn:

- `renderToNodeStream`: **Deprecated** ⛔

Instead, for streaming in Node environments, use:

- `renderToPipeableStream`: **New** ✨

We're also introducing a new API to support streaming SSR with Suspense for modern edge runtime environments, such as Deno and Cloudflare workers:

- `renderToReadableStream`: **New** ✨

The following APIs will continue working, but with limited support for Suspense:

- `renderToString`: **Limited** ⚠
- `renderToStaticMarkup`: **Limited** ⚠

Finally, this API will continue to work for rendering e-mails:

- `renderToStaticNodeStream`

For more information on the changes to server rendering APIs, see the working group post on Upgrading to React 18 on the server, a deep dive on the new Suspense SSR Architecture, and Shaundai Person's talk on Streaming Server Rendering with Suspense at React Conf 2021.

Updates to TypeScript definitions

If your project uses TypeScript, you will need to update your `@types/react` and `@types/react-dom` dependencies to the latest versions. The new types are safer and catch issues that used to be ignored by the type checker. The most notable change is that the `children` prop now needs to be listed explicitly when defining props, for example:

```
interface MyButtonProps {
  color: string;
  children?: React.ReactNode;}
```

See the React 18 typings pull request for a full list of type-only changes. It links to example fixes in library types so you can see how to adjust your code. You can use the automated migration script to help port your application code to the new and safer typings faster.

If you find a bug in the typings, please file an issue in the DefinitelyTyped repo.

Automatic Batching

React 18 adds out-of-the-box performance improvements by doing more batching by default. Batching is when React groups multiple state updates into a single re-render for better performance. Before React 18, we only batched updates inside React event handlers. Updates inside of promises, setTimeout, native event handlers, or any other event were not batched in React by default:

```
// Before React 18 only React events were batched

function handleClick() {
  setCount(c => c + 1);
  setFlag(f => !f);
  // React will only re-render once at the end (that's batching!)
}

setTimeout(() => {
  setCount(c => c + 1);
  setFlag(f => !f);
  // React will render twice, once for each state update (no batching)
}, 1000);
```

Starting in React 18 with `createRoot`, all updates will be automatically batched, no matter where they originate from. This means that updates inside of timeouts, promises, native event handlers or any other event will batch the same way as updates inside of React events:

```
// After React 18 updates inside of timeouts, promises,
// native event handlers or any other event are batched.

function handleClick() {
  setCount(c => c + 1);
  setFlag(f => !f);
  // React will only re-render once at the end (that's batching!)
}

setTimeout(() => {
  setCount(c => c + 1);
  setFlag(f => !f);
  // React will only re-render once at the end (that's batching!)
}, 1000);
```

This is a breaking change, but we expect this to result in less work rendering, and therefore better performance in your applications. To opt-out of automatic batching, you can use `flushSync`:

```
import { flushSync } from 'react-dom';

function handleClick() {
  flushSync(() => {
    setCounter(c => c + 1);
  });
  // React has updated the DOM by now
  flushSync(() => {
    setFlag(f => !f);
  });
  // React has updated the DOM by now
}
```

For more information, see the Automatic batching deep dive.

New APIs for Libraries

In the React 18 Working Group we worked with library maintainers to create new APIs needed to support concurrent rendering for use cases specific to their use case in areas like styles, and external stores. To support React 18, some libraries may need to switch to one of the following APIs:

- `useSyncExternalStore` is a new hook that allows external stores to support concurrent reads by forcing updates to the store to be synchronous. This new API is recommended for any library that integrates with state external to React. For more information, see the useSyncExternalStore overview post and useSyncExternalStore API details.
- `useInsertionEffect` is a new hook that allows CSS-in-JS libraries to address performance issues of injecting styles in render. Unless you've already built a CSS-in-JS library we don't expect you to ever use this. This hook will run after the DOM is mutated, but before layout effects read the new layout. This solves an issue that already exists in React 17 and below, but is even more important in React 18 because React yields to the browser during concurrent rendering, giving it a chance to recalculate layout. For more information, see the Library Upgrade Guide for `<style>`.

React 18 also introduces new APIs for concurrent rendering such as `startTransition`, `useDeferredValue` and `useId`, which we share more about in the release post.

Updates to Strict Mode

In the future, we'd like to add a feature that allows React to add and remove sections of the UI while preserving state. For example, when a user tabs away from a screen and back, React should be able to immediately show the previous screen. To do this, React would unmount and remount trees using the same component state as before.

This feature will give React better performance out-of-the-box, but requires components to be resilient to effects being mounted and destroyed multiple times. Most effects will work without any changes, but some effects assume they are only mounted or destroyed once.

To help surface these issues, React 18 introduces a new development-only check to Strict Mode. This new check will automatically unmount and remount every component, whenever a component mounts for the first time, restoring the previous state on the second mount.

Before this change, React would mount the component and create the effects:

```
* React mounts the component.
    * Layout effects are created.
    * Effect effects are created.
```

With Strict Mode in React 18, React will simulate unmounting and remounting the component in development mode:

```
* React mounts the component.
    * Layout effects are created.
    * Effect effects are created.
* React simulates unmounting the component.
    * Layout effects are destroyed.
    * Effects are destroyed.
* React simulates mounting the component with the previous state.
    * Layout effect setup code runs
    * Effect setup code runs
```

For more information, see the Working Group posts for Adding Reusable State to StrictMode and How to support Reusable State in Effects.

Configuring Your Testing Environment

When you first update your tests to use `createRoot`, you may see this warning in your test console:

The current testing environment is not configured to support act(...)

To fix this, set `globalThis.IS_REACT_ACT_ENVIRONMENT` to `true` before running your test:

```
// In your test setup file
globalThis.IS_REACT_ACT_ENVIRONMENT = true;
```

The purpose of the flag is to tell React that it's running in a unit test-like environment. React will log helpful warnings if you forget to wrap an update with `act`.

You can also set the flag to `false` to tell React that `act` isn't needed. This can be useful for end-to-end tests that simulate a full browser environment.

Eventually, we expect testing libraries will configure this for you automatically. For example, the next version of React Testing Library has built-in support for React 18 without any additional configuration.

More background on the the `act` testing API and related changes is available in the working group.

Dropping Support for Internet Explorer

In this release, React is dropping support for Internet Explorer, which is going out of support on June 15, 2022. We're making this change now because new features introduced in React 18 are built using modern browser features such as microtasks which cannot be adequately polyfilled in IE.

If you need to support Internet Explorer we recommend you stay with React 17.

Deprecations

- `react-dom`: `ReactDOM.render` has been deprecated. Using it will warn and run your app in React 17 mode.
- `react-dom`: `ReactDOM.hydrate` has been deprecated. Using it will warn and run your app in React 17 mode.
- `react-dom`: `ReactDOM.unmountComponentAtNode` has been deprecated.
- `react-dom`: `ReactDOM.renderSubtreeIntoContainer` has been deprecated.
- `react-dom/server`: `ReactDOMServer.renderToNodeStream` has been deprecated.

Other Breaking Changes

- **Consistent useEffect timing**: React now always synchronously flushes effect functions if the update was triggered during a discrete user input event such as a click or a keydown event. Previously, the behavior wasn't always predictable or consistent.
- **Stricter hydration errors**: Hydration mismatches due to missing or extra text content are now treated like errors instead of warnings. React will no longer attempt to "patch up" individual nodes by inserting or deleting a node on the client in an attempt to match the server markup, and will revert to client rendering up to the closest `<Suspense>` boundary in the tree. This ensures the hydrated tree is consistent and avoids potential privacy and security holes that can be caused by hydration mismatches.
- **Suspense trees are always consistent:** If a component suspends before it's fully added to the tree, React will not add it to the tree in an incomplete state or fire its

effects. Instead, React will throw away the new tree completely, wait for the asynchronous operation to finish, and then retry rendering again from scratch. React will render the retry attempt concurrently, and without blocking the browser.

- **Layout Effects with Suspense**: When a tree re-suspends and reverts to a fallback, React will now clean up layout effects, and then re-create them when the content inside the boundary is shown again. This fixes an issue which prevented component libraries from correctly measuring layout when used with Suspense.
- **New JS Environment Requirements**: React now depends on modern browsers features including `Promise`, `Symbol`, and `Object.assign`. If you support older browsers and devices such as Internet Explorer which do not provide modern browser features natively or have non-compliant implementations, consider including a global polyfill in your bundled application.

Other Notable Changes

React

- **Components can now render `undefined`:** React no longer warns if you return `undefined` from a component. This makes the allowed component return values consistent with values that are allowed in the middle of a component tree. We suggest to use a linter to prevent mistakes like forgetting a `return` statement before JSX.
- **In tests, `act` warnings are now opt-in:** If you're running end-to-end tests, the `act` warnings are unnecessary. We've introduced an opt-in mechanism so you can enable them only for unit tests where they are useful and beneficial.
- **No warning about `setState` on unmounted components:** Previously, React warned about memory leaks when you call `setState` on an unmounted component. This warning was added for subscriptions, but people primarily run into it in scenarios where setting state is fine, and workarounds make the code worse. We've removed this warning.
- **No suppression of console logs:** When you use Strict Mode, React renders each component twice to help you find unexpected side effects. In React 17, we've suppressed console logs for one of the two renders to make the logs easier to read. In response to community feedback about this being confusing, we've removed the suppression. Instead, if you have React DevTools installed, the second log's renders will be displayed in grey, and there will be an option (off by default) to suppress them completely.
- **Improved memory usage:** React now cleans up more internal fields on unmount, making the impact from unfixed memory leaks that may exist in your application code less severe.

React DOM Server

- **`renderToString`:** Will no longer error when suspending on the server. Instead, it will emit the fallback HTML for the closest `<Suspense>` boundary and then

retry rendering the same content on the client. It is still recommended that you switch to a streaming API
like `renderToPipeableStream` or `renderToReadableStream` instead.

- **`renderToStaticMarkup`:** Will no longer error when suspending on the server. Instead, it will emit the fallback HTML for the closest `<Suspense>` boundary.

How to Upgrade to React 18

March 08, 2022 by Rick Hanlon

As we shared in the release post, React 18 introduces features powered by our new concurrent renderer, with a gradual adoption strategy for existing applications. In this post, we will guide you through the steps for upgrading to React 18.

Please report any issues you encounter while upgrading to React 18.

Note for React Native users: React 18 will ship in a future version of React Native. This is because React 18 relies on the New React Native Architecture to benefit from the new capabilities presented in this blogpost. For more information, see the React Conf keynote here.

Installing

To install the latest version of React:

```
npm install react react-dom
```

Or if you're using yarn:

```
yarn add react react-dom
```

Updates to Client Rendering APIs

When you first install React 18, you will see a warning in the console:

ReactDOM.render is no longer supported in React 18. Use createRoot instead. Until you switch to the new API, your app will behave as if it's running React 17. Learn more: https://reactjs.org/link/switch-to-createroot

React 18 introduces a new root API which provides better ergonomics for managing roots. The new root API also enables the new concurrent renderer, which allows you to opt-into concurrent features.

```
// Before
import { render } from 'react-dom';
const container = document.getElementById('app');
render(<App tab="home" />, container);

// After
import { createRoot } from 'react-dom/client';
const container = document.getElementById('app');
const root = createRoot(container); // createRoot(container!) if you use TypeScript
root.render(<App tab="home" />);
```

We've also changed `unmountComponentAtNode` to `root.unmount`:

```
// Before
unmountComponentAtNode(container);

// After
root.unmount();
```

We've also removed the callback from render, since it usually does not have the expected result when using Suspense:

```
// Before
const container = document.getElementById('app');
render(<App tab="home" />, container, () => {
  console.log('rendered');
});

// After
function AppWithCallbackAfterRender() {
  useEffect(() => {
    console.log('rendered');
  });

  return <App tab="home" />
}

const container = document.getElementById('app');
const root = createRoot(container);
root.render(<AppWithCallbackAfterRender />);
```

Note:

There is no one-to-one replacement for the old render callback API — it depends on your use case. See the working group post for Replacing render with createRoot for more information.

Finally, if your app uses server-side rendering with hydration, upgrade `hydrate` to `hydrateRoot`:

```
// Before
import { hydrate } from 'react-dom';
const container = document.getElementById('app');
hydrate(<App tab="home" />, container);

// After
import { hydrateRoot } from 'react-dom/client';
const container = document.getElementById('app');
const root = hydrateRoot(container, <App tab="home" />);
```

```
// Unlike with createRoot, you don't need a separate root.render() call
here.
```

For more information, see the working group discussion here.

Note

If your app doesn't work after upgrading, check whether it's wrapped in `<StrictMode>`. Strict Mode has gotten stricter in React 18, and not all your components may be resilient to the new checks it adds in development mode. If removing Strict Mode fixes your app, you can remove it during the upgrade, and then add it back (either at the top or for a part of the tree) after you fix the issues that it's pointing out.

Updates to Server Rendering APIs

In this release, we're revamping our `react-dom/server` APIs to fully support Suspense on the server and Streaming SSR. As part of these changes, we're deprecating the old Node streaming API, which does not support incremental Suspense streaming on the server.

Using this API will now warn:

- `renderToNodeStream`: **Deprecated** ⛔

Instead, for streaming in Node environments, use:

- `renderToPipeableStream`: **New** ✨

We're also introducing a new API to support streaming SSR with Suspense for modern edge runtime environments, such as Deno and Cloudflare workers:

- `renderToReadableStream`: **New** ✨

The following APIs will continue working, but with limited support for Suspense:

- `renderToString`: **Limited** ⚠
- `renderToStaticMarkup`: **Limited** ⚠

Finally, this API will continue to work for rendering e-mails:

- `renderToStaticNodeStream`

For more information on the changes to server rendering APIs, see the working group post on Upgrading to React 18 on the server, a deep dive on the new Suspense SSR Architecture, and Shaundai Person's talk on Streaming Server Rendering with Suspense at React Conf 2021.

Updates to TypeScript definitions

If your project uses TypeScript, you will need to update your `@types/react` and `@types/react-dom` dependencies to the latest versions. The new types are safer and catch issues that used to be ignored by the type checker. The most notable change is that the `children` prop now needs to be listed explicitly when defining props, for example:

```
interface MyButtonProps {
  color: string;
  children?: React.ReactNode;}
```

See the React 18 typings pull request for a full list of type-only changes. It links to example fixes in library types so you can see how to adjust your code. You can use the automated migration script to help port your application code to the new and safer typings faster.

If you find a bug in the typings, please file an issue in the DefinitelyTyped repo.

Automatic Batching

React 18 adds out-of-the-box performance improvements by doing more batching by default. Batching is when React groups multiple state updates into a single re-render for better performance. Before React 18, we only batched updates inside React event handlers. Updates inside of promises, setTimeout, native event handlers, or any other event were not batched in React by default:

```
// Before React 18 only React events were batched

function handleClick() {
  setCount(c => c + 1);
  setFlag(f => !f);
  // React will only re-render once at the end (that's batching!)
}

setTimeout(() => {
  setCount(c => c + 1);
  setFlag(f => !f);
  // React will render twice, once for each state update (no batching)
}, 1000);
```

Starting in React 18 with `createRoot`, all updates will be automatically batched, no matter where they originate from. This means that updates inside of timeouts, promises, native event handlers or any other event will batch the same way as updates inside of React events:

```
// After React 18 updates inside of timeouts, promises,
// native event handlers or any other event are batched.

function handleClick() {
  setCount(c => c + 1);
  setFlag(f => !f);
  // React will only re-render once at the end (that's batching!)
}

setTimeout(() => {
  setCount(c => c + 1);
  setFlag(f => !f);
  // React will only re-render once at the end (that's batching!)
}, 1000);
```

This is a breaking change, but we expect this to result in less work rendering, and therefore better performance in your applications. To opt-out of automatic batching, you can use `flushSync`:

```
import { flushSync } from 'react-dom';

function handleClick() {
  flushSync(() => {
    setCounter(c => c + 1);
  });
  // React has updated the DOM by now
  flushSync(() => {
    setFlag(f => !f);
  });
  // React has updated the DOM by now
}
```

For more information, see the Automatic batching deep dive.

New APIs for Libraries

In the React 18 Working Group we worked with library maintainers to create new APIs needed to support concurrent rendering for use cases specific to their use case in areas like styles, and external stores. To support React 18, some libraries may need to switch to one of the following APIs:

- `useSyncExternalStore` is a new hook that allows external stores to support concurrent reads by forcing updates to the store to be synchronous. This new API is recommended for any library that integrates with state external to React. For more information, see the useSyncExternalStore overview post and useSyncExternalStore API details.
- `useInsertionEffect` is a new hook that allows CSS-in-JS libraries to address performance issues of injecting styles in render. Unless you've already built a CSS-in-JS library we don't expect you to ever use this. This hook will run after the DOM is mutated, but before layout effects read the new layout. This solves an issue that already exists in React 17 and below, but is even more important in React 18 because React yields to the browser during concurrent rendering, giving it a chance to recalculate layout. For more information, see the Library Upgrade Guide for `<style>`.

React 18 also introduces new APIs for concurrent rendering such as `startTransition`, `useDeferredValue` and `useId`, which we share more about in the release post.

Updates to Strict Mode

In the future, we'd like to add a feature that allows React to add and remove sections of the UI while preserving state. For example, when a user tabs away from a screen and back, React should be able to immediately show the previous screen. To do this, React would unmount and remount trees using the same component state as before.

This feature will give React better performance out-of-the-box, but requires components to be resilient to effects being mounted and destroyed multiple times. Most effects will work without any changes, but some effects assume they are only mounted or destroyed once.

To help surface these issues, React 18 introduces a new development-only check to Strict Mode. This new check will automatically unmount and remount every component, whenever a component mounts for the first time, restoring the previous state on the second mount.

Before this change, React would mount the component and create the effects:

```
* React mounts the component.
    * Layout effects are created.
    * Effect effects are created.
```

With Strict Mode in React 18, React will simulate unmounting and remounting the component in development mode:

```
* React mounts the component.
    * Layout effects are created.
    * Effect effects are created.
* React simulates unmounting the component.
    * Layout effects are destroyed.
    * Effects are destroyed.
* React simulates mounting the component with the previous state.
    * Layout effect setup code runs
    * Effect setup code runs
```

For more information, see the Working Group posts for Adding Reusable State to StrictMode and How to support Reusable State in Effects.

Configuring Your Testing Environment

When you first update your tests to use `createRoot`, you may see this warning in your test console:

The current testing environment is not configured to support act(...)

To fix this, set `globalThis.IS_REACT_ACT_ENVIRONMENT` to `true` before running your test:

```
// In your test setup file
globalThis.IS_REACT_ACT_ENVIRONMENT = true;
```

The purpose of the flag is to tell React that it's running in a unit test-like environment. React will log helpful warnings if you forget to wrap an update with `act`.

You can also set the flag to `false` to tell React that `act` isn't needed. This can be useful for end-to-end tests that simulate a full browser environment.

Eventually, we expect testing libraries will configure this for you automatically. For example, the next version of React Testing Library has built-in support for React 18 without any additional configuration.

More background on the the `act` testing API and related changes is available in the working group.

Dropping Support for Internet Explorer

In this release, React is dropping support for Internet Explorer, which is going out of support on June 15, 2022. We're making this change now because new features introduced in React 18 are built using modern browser features such as microtasks which cannot be adequately polyfilled in IE.

If you need to support Internet Explorer we recommend you stay with React 17.

Deprecations

- `react-dom`: `ReactDOM.render` has been deprecated. Using it will warn and run your app in React 17 mode.
- `react-dom`: `ReactDOM.hydrate` has been deprecated. Using it will warn and run your app in React 17 mode.
- `react-dom`: `ReactDOM.unmountComponentAtNode` has been deprecated.
- `react-dom`: `ReactDOM.renderSubtreeIntoContainer` has been deprecated.
- `react-dom/server`: `ReactDOMServer.renderToNodeStream` has been deprecated.

Other Breaking Changes

- **Consistent useEffect timing**: React now always synchronously flushes effect functions if the update was triggered during a discrete user input event such as a click or a keydown event. Previously, the behavior wasn't always predictable or consistent.
- **Stricter hydration errors**: Hydration mismatches due to missing or extra text content are now treated like errors instead of warnings. React will no longer attempt to "patch up" individual nodes by inserting or deleting a node on the client in an attempt to match the server markup, and will revert to client rendering up to the closest `<Suspense>` boundary in the tree. This ensures the hydrated tree is consistent and avoids potential privacy and security holes that can be caused by hydration mismatches.
- **Suspense trees are always consistent:** If a component suspends before it's fully added to the tree, React will not add it to the tree in an incomplete state or fire its

effects. Instead, React will throw away the new tree completely, wait for the asynchronous operation to finish, and then retry rendering again from scratch. React will render the retry attempt concurrently, and without blocking the browser.

- **Layout Effects with Suspense**: When a tree re-suspends and reverts to a fallback, React will now clean up layout effects, and then re-create them when the content inside the boundary is shown again. This fixes an issue which prevented component libraries from correctly measuring layout when used with Suspense.
- **New JS Environment Requirements**: React now depends on modern browsers features including `Promise`, `Symbol`, and `Object.assign`. If you support older browsers and devices such as Internet Explorer which do not provide modern browser features natively or have non-compliant implementations, consider including a global polyfill in your bundled application.

Other Notable Changes

React

- **Components can now render `undefined`:** React no longer warns if you return `undefined` from a component. This makes the allowed component return values consistent with values that are allowed in the middle of a component tree. We suggest to use a linter to prevent mistakes like forgetting a `return` statement before JSX.
- **In tests, `act` warnings are now opt-in:** If you're running end-to-end tests, the `act` warnings are unnecessary. We've introduced an opt-in mechanism so you can enable them only for unit tests where they are useful and beneficial.
- **No warning about `setState` on unmounted components:** Previously, React warned about memory leaks when you call `setState` on an unmounted component. This warning was added for subscriptions, but people primarily run into it in scenarios where setting state is fine, and workarounds make the code worse. We've removed this warning.
- **No suppression of console logs:** When you use Strict Mode, React renders each component twice to help you find unexpected side effects. In React 17, we've suppressed console logs for one of the two renders to make the logs easier to read. In response to community feedback about this being confusing, we've removed the suppression. Instead, if you have React DevTools installed, the second log's renders will be displayed in grey, and there will be an option (off by default) to suppress them completely.
- **Improved memory usage:** React now cleans up more internal fields on unmount, making the impact from unfixed memory leaks that may exist in your application code less severe.

React DOM Server

- **`renderToString`:** Will no longer error when suspending on the server. Instead, it will emit the fallback HTML for the closest `<Suspense>` boundary and then

retry rendering the same content on the client. It is still recommended that you switch to a streaming API
like `renderToPipeableStream` or `renderToReadableStream` instead.

- **`renderToStaticMarkup`:** Will no longer error when suspending on the server. Instead, it will emit the fallback HTML for the closest `<Suspense>` boundary.

How to Upgrade to React 18

March 08, 2022 by Rick Hanlon

As we shared in the release post, React 18 introduces features powered by our new concurrent renderer, with a gradual adoption strategy for existing applications. In this post, we will guide you through the steps for upgrading to React 18.

Please report any issues you encounter while upgrading to React 18.

Note for React Native users: React 18 will ship in a future version of React Native. This is because React 18 relies on the New React Native Architecture to benefit from the new capabilities presented in this blogpost. For more information, see the React Conf keynote here.

Installing

To install the latest version of React:

```
npm install react react-dom
```

Or if you're using yarn:

```
yarn add react react-dom
```

Updates to Client Rendering APIs

When you first install React 18, you will see a warning in the console:

ReactDOM.render is no longer supported in React 18. Use createRoot instead. Until you switch to the new API, your app will behave as if it's running React 17. Learn more: https://reactjs.org/link/switch-to-createroot

React 18 introduces a new root API which provides better ergonomics for managing roots. The new root API also enables the new concurrent renderer, which allows you to opt-into concurrent features.

```
// Before
import { render } from 'react-dom';
const container = document.getElementById('app');
render(<App tab="home" />, container);

// After
import { createRoot } from 'react-dom/client';
const container = document.getElementById('app');
const root = createRoot(container); // createRoot(container!) if you use TypeScript
root.render(<App tab="home" />);
```

We've also changed `unmountComponentAtNode` to `root.unmount`:

```
// Before
unmountComponentAtNode(container);

// After
root.unmount();
```

We've also removed the callback from render, since it usually does not have the expected result when using Suspense:

```
// Before
const container = document.getElementById('app');
render(<App tab="home" />, container, () => {
  console.log('rendered');
});

// After
function AppWithCallbackAfterRender() {
  useEffect(() => {
    console.log('rendered');
  });

  return <App tab="home" />
}

const container = document.getElementById('app');
const root = createRoot(container);
root.render(<AppWithCallbackAfterRender />);
```

Note:

There is no one-to-one replacement for the old render callback API — it depends on your use case. See the working group post for Replacing render with createRoot for more information.

Finally, if your app uses server-side rendering with hydration, upgrade `hydrate` to `hydrateRoot`:

```
// Before
import { hydrate } from 'react-dom';
const container = document.getElementById('app');
hydrate(<App tab="home" />, container);

// After
import { hydrateRoot } from 'react-dom/client';
const container = document.getElementById('app');
const root = hydrateRoot(container, <App tab="home" />);
```

```
// Unlike with createRoot, you don't need a separate root.render() call
here.
```

For more information, see the working group discussion here.

Note

If your app doesn't work after upgrading, check whether it's wrapped in `<StrictMode>`. Strict Mode has gotten stricter in React 18, and not all your components may be resilient to the new checks it adds in development mode. If removing Strict Mode fixes your app, you can remove it during the upgrade, and then add it back (either at the top or for a part of the tree) after you fix the issues that it's pointing out.

Updates to Server Rendering APIs

In this release, we're revamping our `react-dom/server` APIs to fully support Suspense on the server and Streaming SSR. As part of these changes, we're deprecating the old Node streaming API, which does not support incremental Suspense streaming on the server.

Using this API will now warn:

- `renderToNodeStream`: **Deprecated** ⛔

Instead, for streaming in Node environments, use:

- `renderToPipeableStream`: **New** ✨

We're also introducing a new API to support streaming SSR with Suspense for modern edge runtime environments, such as Deno and Cloudflare workers:

- `renderToReadableStream`: **New** ✨

The following APIs will continue working, but with limited support for Suspense:

- `renderToString`: **Limited** ⚠
- `renderToStaticMarkup`: **Limited** ⚠

Finally, this API will continue to work for rendering e-mails:

- `renderToStaticNodeStream`

For more information on the changes to server rendering APIs, see the working group post on Upgrading to React 18 on the server, a deep dive on the new Suspense SSR Architecture, and Shaundai Person's talk on Streaming Server Rendering with Suspense at React Conf 2021.

Updates to TypeScript definitions

If your project uses TypeScript, you will need to update your `@types/react` and `@types/react-dom` dependencies to the latest versions. The new types are safer and catch issues that used to be ignored by the type checker. The most notable change is that the `children` prop now needs to be listed explicitly when defining props, for example:

```
interface MyButtonProps {
  color: string;
  children?: React.ReactNode;}
```

See the React 18 typings pull request for a full list of type-only changes. It links to example fixes in library types so you can see how to adjust your code. You can use the automated migration script to help port your application code to the new and safer typings faster.

If you find a bug in the typings, please file an issue in the DefinitelyTyped repo.

Automatic Batching

React 18 adds out-of-the-box performance improvements by doing more batching by default. Batching is when React groups multiple state updates into a single re-render for better performance. Before React 18, we only batched updates inside React event handlers. Updates inside of promises, setTimeout, native event handlers, or any other event were not batched in React by default:

```
// Before React 18 only React events were batched

function handleClick() {
  setCount(c => c + 1);
  setFlag(f => !f);
  // React will only re-render once at the end (that's batching!)
}

setTimeout(() => {
  setCount(c => c + 1);
  setFlag(f => !f);
  // React will render twice, once for each state update (no batching)
}, 1000);
```

Starting in React 18 with `createRoot`, all updates will be automatically batched, no matter where they originate from. This means that updates inside of timeouts, promises, native event handlers or any other event will batch the same way as updates inside of React events:

```
// After React 18 updates inside of timeouts, promises,
// native event handlers or any other event are batched.

function handleClick() {
  setCount(c => c + 1);
  setFlag(f => !f);
  // React will only re-render once at the end (that's batching!)
}

setTimeout(() => {
  setCount(c => c + 1);
  setFlag(f => !f);
  // React will only re-render once at the end (that's batching!)
}, 1000);
```

This is a breaking change, but we expect this to result in less work rendering, and therefore better performance in your applications. To opt-out of automatic batching, you can use `flushSync`:

```
import { flushSync } from 'react-dom';

function handleClick() {
  flushSync(() => {
    setCounter(c => c + 1);
  });
  // React has updated the DOM by now
  flushSync(() => {
    setFlag(f => !f);
  });
  // React has updated the DOM by now
}
```

For more information, see the Automatic batching deep dive.

New APIs for Libraries

In the React 18 Working Group we worked with library maintainers to create new APIs needed to support concurrent rendering for use cases specific to their use case in areas like styles, and external stores. To support React 18, some libraries may need to switch to one of the following APIs:

- `useSyncExternalStore` is a new hook that allows external stores to support concurrent reads by forcing updates to the store to be synchronous. This new API is recommended for any library that integrates with state external to React. For more information, see the useSyncExternalStore overview post and useSyncExternalStore API details.
- `useInsertionEffect` is a new hook that allows CSS-in-JS libraries to address performance issues of injecting styles in render. Unless you've already built a CSS-in-JS library we don't expect you to ever use this. This hook will run after the DOM is mutated, but before layout effects read the new layout. This solves an issue that already exists in React 17 and below, but is even more important in React 18 because React yields to the browser during concurrent rendering, giving it a chance to recalculate layout. For more information, see the Library Upgrade Guide for `<style>`.

React 18 also introduces new APIs for concurrent rendering such as `startTransition`, `useDeferredValue` and `useId`, which we share more about in the release post.

Updates to Strict Mode

In the future, we'd like to add a feature that allows React to add and remove sections of the UI while preserving state. For example, when a user tabs away from a screen and back, React should be able to immediately show the previous screen. To do this, React would unmount and remount trees using the same component state as before.

This feature will give React better performance out-of-the-box, but requires components to be resilient to effects being mounted and destroyed multiple times. Most effects will work without any changes, but some effects assume they are only mounted or destroyed once.

To help surface these issues, React 18 introduces a new development-only check to Strict Mode. This new check will automatically unmount and remount every component, whenever a component mounts for the first time, restoring the previous state on the second mount.

Before this change, React would mount the component and create the effects:

```
* React mounts the component.
    * Layout effects are created.
    * Effect effects are created.
```

With Strict Mode in React 18, React will simulate unmounting and remounting the component in development mode:

```
* React mounts the component.
    * Layout effects are created.
    * Effect effects are created.
* React simulates unmounting the component.
    * Layout effects are destroyed.
    * Effects are destroyed.
* React simulates mounting the component with the previous state.
    * Layout effect setup code runs
    * Effect setup code runs
```

For more information, see the Working Group posts for Adding Reusable State to StrictMode and How to support Reusable State in Effects.

Configuring Your Testing Environment

When you first update your tests to use `createRoot`, you may see this warning in your test console:

The current testing environment is not configured to support act(...)

To fix this, set `globalThis.IS_REACT_ACT_ENVIRONMENT` to `true` before running your test:

```
// In your test setup file
globalThis.IS_REACT_ACT_ENVIRONMENT = true;
```

The purpose of the flag is to tell React that it's running in a unit test-like environment. React will log helpful warnings if you forget to wrap an update with `act`.

You can also set the flag to `false` to tell React that `act` isn't needed. This can be useful for end-to-end tests that simulate a full browser environment.

Eventually, we expect testing libraries will configure this for you automatically. For example, the next version of React Testing Library has built-in support for React 18 without any additional configuration.

More background on the the `act` testing API and related changes is available in the working group.

Dropping Support for Internet Explorer

In this release, React is dropping support for Internet Explorer, which is going out of support on June 15, 2022. We're making this change now because new features introduced in React 18 are built using modern browser features such as microtasks which cannot be adequately polyfilled in IE.

If you need to support Internet Explorer we recommend you stay with React 17.

Deprecations

- `react-dom`: `ReactDOM.render` has been deprecated. Using it will warn and run your app in React 17 mode.
- `react-dom`: `ReactDOM.hydrate` has been deprecated. Using it will warn and run your app in React 17 mode.
- `react-dom`: `ReactDOM.unmountComponentAtNode` has been deprecated.
- `react-dom`: `ReactDOM.renderSubtreeIntoContainer` has been deprecated.
- `react-dom/server`: `ReactDOMServer.renderToNodeStream` has been deprecated.

Other Breaking Changes

- **Consistent useEffect timing**: React now always synchronously flushes effect functions if the update was triggered during a discrete user input event such as a click or a keydown event. Previously, the behavior wasn't always predictable or consistent.
- **Stricter hydration errors**: Hydration mismatches due to missing or extra text content are now treated like errors instead of warnings. React will no longer attempt to "patch up" individual nodes by inserting or deleting a node on the client in an attempt to match the server markup, and will revert to client rendering up to the closest `<Suspense>` boundary in the tree. This ensures the hydrated tree is consistent and avoids potential privacy and security holes that can be caused by hydration mismatches.
- **Suspense trees are always consistent:** If a component suspends before it's fully added to the tree, React will not add it to the tree in an incomplete state or fire its

effects. Instead, React will throw away the new tree completely, wait for the asynchronous operation to finish, and then retry rendering again from scratch. React will render the retry attempt concurrently, and without blocking the browser.

- **Layout Effects with Suspense**: When a tree re-suspends and reverts to a fallback, React will now clean up layout effects, and then re-create them when the content inside the boundary is shown again. This fixes an issue which prevented component libraries from correctly measuring layout when used with Suspense.
- **New JS Environment Requirements**: React now depends on modern browsers features including `Promise`, `Symbol`, and `Object.assign`. If you support older browsers and devices such as Internet Explorer which do not provide modern browser features natively or have non-compliant implementations, consider including a global polyfill in your bundled application.

Other Notable Changes

React

- **Components can now render `undefined`:** React no longer warns if you return `undefined` from a component. This makes the allowed component return values consistent with values that are allowed in the middle of a component tree. We suggest to use a linter to prevent mistakes like forgetting a `return` statement before JSX.
- **In tests, `act` warnings are now opt-in:** If you're running end-to-end tests, the `act` warnings are unnecessary. We've introduced an opt-in mechanism so you can enable them only for unit tests where they are useful and beneficial.
- **No warning about `setState` on unmounted components:** Previously, React warned about memory leaks when you call `setState` on an unmounted component. This warning was added for subscriptions, but people primarily run into it in scenarios where setting state is fine, and workarounds make the code worse. We've removed this warning.
- **No suppression of console logs:** When you use Strict Mode, React renders each component twice to help you find unexpected side effects. In React 17, we've suppressed console logs for one of the two renders to make the logs easier to read. In response to community feedback about this being confusing, we've removed the suppression. Instead, if you have React DevTools installed, the second log's renders will be displayed in grey, and there will be an option (off by default) to suppress them completely.
- **Improved memory usage:** React now cleans up more internal fields on unmount, making the impact from unfixed memory leaks that may exist in your application code less severe.

React DOM Server

- **`renderToString`:** Will no longer error when suspending on the server. Instead, it will emit the fallback HTML for the closest `<Suspense>` boundary and then

retry rendering the same content on the client. It is still recommended that you switch to a streaming API
like `renderToPipeableStream` or `renderToReadableStream` instead.

- **`renderToStaticMarkup`:** Will no longer error when suspending on the server. Instead, it will emit the fallback HTML for the closest `<Suspense>` boundary.

How to Upgrade to React 18

March 08, 2022 by Rick Hanlon

As we shared in the release post, React 18 introduces features powered by our new concurrent renderer, with a gradual adoption strategy for existing applications. In this post, we will guide you through the steps for upgrading to React 18.

Please report any issues you encounter while upgrading to React 18.

Note for React Native users: React 18 will ship in a future version of React Native. This is because React 18 relies on the New React Native Architecture to benefit from the new capabilities presented in this blogpost. For more information, see the React Conf keynote here.

Installing

To install the latest version of React:

```
npm install react react-dom
```

Or if you're using yarn:

```
yarn add react react-dom
```

Updates to Client Rendering APIs

When you first install React 18, you will see a warning in the console:

ReactDOM.render is no longer supported in React 18. Use createRoot instead. Until you switch to the new API, your app will behave as if it's running React 17. Learn more: https://reactjs.org/link/switch-to-createroot

React 18 introduces a new root API which provides better ergonomics for managing roots. The new root API also enables the new concurrent renderer, which allows you to opt-into concurrent features.

```
// Before
import { render } from 'react-dom';
const container = document.getElementById('app');
render(<App tab="home" />, container);

// After
import { createRoot } from 'react-dom/client';
const container = document.getElementById('app');
const root = createRoot(container); // createRoot(container!) if you use TypeScript
root.render(<App tab="home" />);
```

We've also changed `unmountComponentAtNode` to `root.unmount`:

```
// Before
unmountComponentAtNode(container);

// After
root.unmount();
```

We've also removed the callback from render, since it usually does not have the expected result when using Suspense:

```
// Before
const container = document.getElementById('app');
render(<App tab="home" />, container, () => {
  console.log('rendered');
});

// After
function AppWithCallbackAfterRender() {
  useEffect(() => {
    console.log('rendered');
  });

  return <App tab="home" />
}

const container = document.getElementById('app');
const root = createRoot(container);
root.render(<AppWithCallbackAfterRender />);
```

Note:

There is no one-to-one replacement for the old render callback API — it depends on your use case. See the working group post for Replacing render with createRoot for more information.

Finally, if your app uses server-side rendering with hydration, upgrade `hydrate` to `hydrateRoot`:

```
// Before
import { hydrate } from 'react-dom';
const container = document.getElementById('app');
hydrate(<App tab="home" />, container);

// After
import { hydrateRoot } from 'react-dom/client';
const container = document.getElementById('app');
const root = hydrateRoot(container, <App tab="home" />);
```

```
// Unlike with createRoot, you don't need a separate root.render() call
here.
```

For more information, see the working group discussion here.

Note

If your app doesn't work after upgrading, check whether it's wrapped in `<StrictMode>`. Strict Mode has gotten stricter in React 18, and not all your components may be resilient to the new checks it adds in development mode. If removing Strict Mode fixes your app, you can remove it during the upgrade, and then add it back (either at the top or for a part of the tree) after you fix the issues that it's pointing out.

Updates to Server Rendering APIs

In this release, we're revamping our `react-dom/server` APIs to fully support Suspense on the server and Streaming SSR. As part of these changes, we're deprecating the old Node streaming API, which does not support incremental Suspense streaming on the server.

Using this API will now warn:

- `renderToNodeStream`: **Deprecated** ⛔

Instead, for streaming in Node environments, use:

- `renderToPipeableStream`: **New** ✨

We're also introducing a new API to support streaming SSR with Suspense for modern edge runtime environments, such as Deno and Cloudflare workers:

- `renderToReadableStream`: **New** ✨

The following APIs will continue working, but with limited support for Suspense:

- `renderToString`: **Limited** ⚠
- `renderToStaticMarkup`: **Limited** ⚠

Finally, this API will continue to work for rendering e-mails:

- `renderToStaticNodeStream`

For more information on the changes to server rendering APIs, see the working group post on Upgrading to React 18 on the server, a deep dive on the new Suspense SSR Architecture, and Shaundai Person's talk on Streaming Server Rendering with Suspense at React Conf 2021.

Updates to TypeScript definitions

If your project uses TypeScript, you will need to update your `@types/react` and `@types/react-dom` dependencies to the latest versions. The new types are safer and catch issues that used to be ignored by the type checker. The most notable change is that the `children` prop now needs to be listed explicitly when defining props, for example:

```
interface MyButtonProps {
  color: string;
  children?: React.ReactNode;}
```

See the React 18 typings pull request for a full list of type-only changes. It links to example fixes in library types so you can see how to adjust your code. You can use the automated migration script to help port your application code to the new and safer typings faster.

If you find a bug in the typings, please file an issue in the DefinitelyTyped repo.

Automatic Batching

React 18 adds out-of-the-box performance improvements by doing more batching by default. Batching is when React groups multiple state updates into a single re-render for better performance. Before React 18, we only batched updates inside React event handlers. Updates inside of promises, setTimeout, native event handlers, or any other event were not batched in React by default:

```
// Before React 18 only React events were batched

function handleClick() {
  setCount(c => c + 1);
  setFlag(f => !f);
  // React will only re-render once at the end (that's batching!)
}

setTimeout(() => {
  setCount(c => c + 1);
  setFlag(f => !f);
  // React will render twice, once for each state update (no batching)
}, 1000);
```

Starting in React 18 with `createRoot`, all updates will be automatically batched, no matter where they originate from. This means that updates inside of timeouts, promises, native event handlers or any other event will batch the same way as updates inside of React events:

```
// After React 18 updates inside of timeouts, promises,
// native event handlers or any other event are batched.

function handleClick() {
  setCount(c => c + 1);
  setFlag(f => !f);
  // React will only re-render once at the end (that's batching!)
}

setTimeout(() => {
  setCount(c => c + 1);
  setFlag(f => !f);
  // React will only re-render once at the end (that's batching!)
}, 1000);
```

This is a breaking change, but we expect this to result in less work rendering, and therefore better performance in your applications. To opt-out of automatic batching, you can use `flushSync`:

```
import { flushSync } from 'react-dom';

function handleClick() {
  flushSync(() => {
    setCounter(c => c + 1);
  });
  // React has updated the DOM by now
  flushSync(() => {
    setFlag(f => !f);
  });
  // React has updated the DOM by now
}
```

For more information, see the Automatic batching deep dive.

New APIs for Libraries

In the React 18 Working Group we worked with library maintainers to create new APIs needed to support concurrent rendering for use cases specific to their use case in areas like styles, and external stores. To support React 18, some libraries may need to switch to one of the following APIs:

- `useSyncExternalStore` is a new hook that allows external stores to support concurrent reads by forcing updates to the store to be synchronous. This new API is recommended for any library that integrates with state external to React. For more information, see the useSyncExternalStore overview post and useSyncExternalStore API details.
- `useInsertionEffect` is a new hook that allows CSS-in-JS libraries to address performance issues of injecting styles in render. Unless you've already built a CSS-in-JS library we don't expect you to ever use this. This hook will run after the DOM is mutated, but before layout effects read the new layout. This solves an issue that already exists in React 17 and below, but is even more important in React 18 because React yields to the browser during concurrent rendering, giving it a chance to recalculate layout. For more information, see the Library Upgrade Guide for `<style>`.

React 18 also introduces new APIs for concurrent rendering such as `startTransition`, `useDeferredValue` and `useId`, which we share more about in the release post.

Updates to Strict Mode

In the future, we'd like to add a feature that allows React to add and remove sections of the UI while preserving state. For example, when a user tabs away from a screen and back, React should be able to immediately show the previous screen. To do this, React would unmount and remount trees using the same component state as before.

This feature will give React better performance out-of-the-box, but requires components to be resilient to effects being mounted and destroyed multiple times. Most effects will work without any changes, but some effects assume they are only mounted or destroyed once.

To help surface these issues, React 18 introduces a new development-only check to Strict Mode. This new check will automatically unmount and remount every component, whenever a component mounts for the first time, restoring the previous state on the second mount.

Before this change, React would mount the component and create the effects:

```
* React mounts the component.
    * Layout effects are created.
    * Effect effects are created.
```

With Strict Mode in React 18, React will simulate unmounting and remounting the component in development mode:

```
* React mounts the component.
    * Layout effects are created.
    * Effect effects are created.
* React simulates unmounting the component.
    * Layout effects are destroyed.
    * Effects are destroyed.
* React simulates mounting the component with the previous state.
    * Layout effect setup code runs
    * Effect setup code runs
```

For more information, see the Working Group posts for Adding Reusable State to StrictMode and How to support Reusable State in Effects.

Configuring Your Testing Environment

When you first update your tests to use `createRoot`, you may see this warning in your test console:

The current testing environment is not configured to support act(...)

To fix this, set `globalThis.IS_REACT_ACT_ENVIRONMENT` to `true` before running your test:

```
// In your test setup file
globalThis.IS_REACT_ACT_ENVIRONMENT = true;
```

The purpose of the flag is to tell React that it's running in a unit test-like environment. React will log helpful warnings if you forget to wrap an update with `act`.

You can also set the flag to `false` to tell React that `act` isn't needed. This can be useful for end-to-end tests that simulate a full browser environment.

Eventually, we expect testing libraries will configure this for you automatically. For example, the next version of React Testing Library has built-in support for React 18 without any additional configuration.

More background on the the `act` testing API and related changes is available in the working group.

Dropping Support for Internet Explorer

In this release, React is dropping support for Internet Explorer, which is going out of support on June 15, 2022. We're making this change now because new features introduced in React 18 are built using modern browser features such as microtasks which cannot be adequately polyfilled in IE.

If you need to support Internet Explorer we recommend you stay with React 17.

Deprecations

- `react-dom`: `ReactDOM.render` has been deprecated. Using it will warn and run your app in React 17 mode.
- `react-dom`: `ReactDOM.hydrate` has been deprecated. Using it will warn and run your app in React 17 mode.
- `react-dom`: `ReactDOM.unmountComponentAtNode` has been deprecated.
- `react-dom`: `ReactDOM.renderSubtreeIntoContainer` has been deprecated.
- `react-dom/server`: `ReactDOMServer.renderToNodeStream` has been deprecated.

Other Breaking Changes

- **Consistent useEffect timing**: React now always synchronously flushes effect functions if the update was triggered during a discrete user input event such as a click or a keydown event. Previously, the behavior wasn't always predictable or consistent.
- **Stricter hydration errors**: Hydration mismatches due to missing or extra text content are now treated like errors instead of warnings. React will no longer attempt to "patch up" individual nodes by inserting or deleting a node on the client in an attempt to match the server markup, and will revert to client rendering up to the closest `<Suspense>` boundary in the tree. This ensures the hydrated tree is consistent and avoids potential privacy and security holes that can be caused by hydration mismatches.
- **Suspense trees are always consistent:** If a component suspends before it's fully added to the tree, React will not add it to the tree in an incomplete state or fire its

effects. Instead, React will throw away the new tree completely, wait for the asynchronous operation to finish, and then retry rendering again from scratch. React will render the retry attempt concurrently, and without blocking the browser.

- **Layout Effects with Suspense**: When a tree re-suspends and reverts to a fallback, React will now clean up layout effects, and then re-create them when the content inside the boundary is shown again. This fixes an issue which prevented component libraries from correctly measuring layout when used with Suspense.
- **New JS Environment Requirements**: React now depends on modern browsers features including `Promise`, `Symbol`, and `Object.assign`. If you support older browsers and devices such as Internet Explorer which do not provide modern browser features natively or have non-compliant implementations, consider including a global polyfill in your bundled application.

Other Notable Changes

React

- **Components can now render `undefined`:** React no longer warns if you return `undefined` from a component. This makes the allowed component return values consistent with values that are allowed in the middle of a component tree. We suggest to use a linter to prevent mistakes like forgetting a `return` statement before JSX.
- **In tests, `act` warnings are now opt-in:** If you're running end-to-end tests, the `act` warnings are unnecessary. We've introduced an opt-in mechanism so you can enable them only for unit tests where they are useful and beneficial.
- **No warning about `setState` on unmounted components:** Previously, React warned about memory leaks when you call `setState` on an unmounted component. This warning was added for subscriptions, but people primarily run into it in scenarios where setting state is fine, and workarounds make the code worse. We've removed this warning.
- **No suppression of console logs:** When you use Strict Mode, React renders each component twice to help you find unexpected side effects. In React 17, we've suppressed console logs for one of the two renders to make the logs easier to read. In response to community feedback about this being confusing, we've removed the suppression. Instead, if you have React DevTools installed, the second log's renders will be displayed in grey, and there will be an option (off by default) to suppress them completely.
- **Improved memory usage:** React now cleans up more internal fields on unmount, making the impact from unfixed memory leaks that may exist in your application code less severe.

React DOM Server

- **`renderToString`:** Will no longer error when suspending on the server. Instead, it will emit the fallback HTML for the closest `<Suspense>` boundary and then

retry rendering the same content on the client. It is still recommended that you switch to a streaming API
like `renderToPipeableStream` or `renderToReadableStream` instead.

- **`renderToStaticMarkup`:** Will no longer error when suspending on the server. Instead, it will emit the fallback HTML for the closest `<Suspense>` boundary.