三天精通 React，一天理解并发渲染

Change logs	Date
新增 Breaking Change in concurrent render	20230417

> 此文献给没有使用过 React 的小伙伴，帮助小伙伴们迅速精通 React

前言

React 官网文档比较完善，本文更注重结合实际项目中常见的问题，来介绍 React 的用法

1. 第一天基础篇，结合工作中经常遇到的一些实际应用场景，补充介绍官方文档中提及的 基础概念及 API
2. 第二天 React 生态，介绍了一些常用的 React 开发相关的工具、库，以及常见问题的排查定位方法、性能优化方法
3. 第三天进阶篇，介绍了各种副作用的处理范式，受控与非受控组件的选择

New

第四天 React 18 并发渲染原理介绍，分别介绍了 Lane、Scheduler 以及新增的并发渲染相关 API

Fun Facts

	React	Vue	Angular
NPM weekly downloads （由于 cnpm 无法查看包，数据不全）	12,635,966	2,662,666	823,653
Dependents	59,757	30,212	4,088

从 npm 下载量和被其他包依赖的数量来看，React 的社区最活跃

Why We Choose React

跟 Vue、Angular、相比，React 有以下优势：

1. 丰富、充满活力的开源生态

2. React 是 Just JavaScript，没有其他概念需要学习
3. 学习资料丰富
4. 架构清晰，接口和实现分离较好，易于定制化及扩展
5. 维护团队可靠，FB 背书

第一天：基础篇

在开始基础篇前，先声明一下本文将不介绍 class component 相关的任何 api 和使用范式。class component 的写法已不再推荐使用。

JSX 是啥?

Try it on Codesandbox

JSX 可以理解为一种 JS 语法糖。它让固定传参的函数嵌套调用看上去像 xml，以使代码更清晰易懂。

下面这段代码

```
React.createElement(
    ComponentA,
    {attr1: 'A', attr2: 0},
    React.createElement(
        ComponentB1,
        {attr1: 'B', attr2: 1},
        React.createElement(ComponentC1, propsC1),
        React.createElement(ComponentC2, propsC2)
    ),
    React.createElement(
        'h1',
        {style: {backgroundColor: 'red'}},
        `Hello ${varWorld}`
    ),
)
```

如果用 JSX 语法来写，就会变成这样

```
<ComponentA attr1="A" attr2={0}>
    <ComponentB1 attr1="B" attr2={1}>
        <ComponentC1 {...propsC1} />
        <ComponentC2 {...propsC2} />
    </ComponentB1>
    <h1 style={{backgroundColor: 'red'}}>Hello {varWorld}</h1>
</ComponentA>
```

JSX 语法在运行前会通过编译工具转成普通的 JavaScript 语法。

> **Before React V17**
>
> 在 React V17 之前需要注意，编译工具只是改语法，对于编译结果 `React.createElement` 的调用，React 本身的引入需要自己处理。所以，**凡是用到 JSX 语法的文件，一定要在头部写 `import React from 'react';`** 来导入 React 这个变量。如果不导入，那么 React 变量就不存在，会导致 `React is not defined` 报错。
>
> **After React V17**
>
> 在 React V17 中，依靠 @babel/plugin-transform-react-jsx (包含在 @babel/preset-react 中），可以省略 `import React from 'react';` 这段代码。如果使用的 TypeScript **>= 4.1.2**，可以在 tsconfig.json 文件中通过 jsx 配置 来省略手动引入 React 这一句代码。

JSX 需要注意的几点

关键字冲突

因为 JSX 只是 JS 的语法糖，所以 React 在设计 DOM Element API 时，避免了与 JavaScript 的关键字冲突的一些属性名，下面举几个常见例子：

```
<label for="xxx"> // html
<label htmlFor="xxx"> // JSX

<div class="box"> // html
<div className="box"> // JSX
```

> 有 TypeScript 的支持，可以通过静态检查、智能提示保证代码正确性，无需记忆

使用 html 字符串

Try it on Codesandbox

React 会对插入在文字做 html 转义，避免一些安全问题。但我们有时需要直接插入 html 代码片段，此时可以使用 `dangerouslySetInnerHTML` 属性来设置某个 DOM Element 的 innerHTML。

什么是组件?

> 组件就是个函数而已

Try it on Codesandbox

React 应用是以组件化的方式搭建的。这使得 React 代码易于实现、维护、复用及测试。

在 React 中，编写一个组件就跟编写一个函数一样简单。我们称为函数组件。

让我们来编写一个按钮组件，这个按钮可以指定颜色、绑定点击事件、设置按钮文案。

UI 组件一般都是纯函数组件，所谓纯函数就是有特定的输入就能得到特定的输出，是完全可以预测可逆推的输入和输出关系。即 ***UI = f(x)***。

让我们来从输出（**UI**）开始，来思考如何设计这个组件。

首先，因为是一个按钮组件，所以输出使用 `<button>` 作为标准 html 按钮输出

```
function Button(props) {
  return <button></button>;
}
```

其次，按钮需要能通过 props.color 设置按钮颜色

```
function Button(props) {
  const { color } = props;
  const style = {
    backgroundColor: color,
  };
  return <button style={style}></button>
}
```

然后，添加 onClick 事件监听

```
function Button(props) {
  const { color, onClick } = props;
  const style = {
    backgroundColor: color,
  };
  return <button style={style} onClick={onClick}></button>
}
```

最后，为了让 Button 组件用起来和 html button 标签类似，我们将 Button 内容的渲染从 props.children 这个特殊属性上去取（特殊在，JSX 语法定义非闭合标签中的内容会作为 props.children 传递）

```
import React from 'react';

export function Button(props) {
  const { color, onClick, children } = props;
  const style = {
    backgroundColor: color,
  };
  return <button style={style} onClick={onClick}>{children}</button>
```

```
}
```

现在我们可以这样使用 Button 组件

```
import React from 'react';
import { Button } from './Button';

export function MyReactApp() {
  return (
    <Button
        color="red"
        onClick={() => { alert('Hello'); }}
    >
        Hello React.
    </Button>
  );
}
```

现在我们已经编写了第一个 React 组件 🎉🎉🎉 ，是不是 **So Easy** 🤣 ，实际项目中，React 应用就是由这样一个个功能专注，逻辑简单的组件拼装起来的。

在 Button 中，我们没有任何的内部状态需要维护。样式渲染、点击行为、文案都是通过 props 参数传递进来的，我们称这种自身没有任何状态的组件叫**无状态组件 (Stateless Function Component)**。

大多数 UI 组件都是无状态组件，上层组件计算、维护 UI 状态，将 UI 状态传给这些无状态组件，来达到 UI 组件的高易复用性和易维护性。

让我们再看看有状态组件 **(Stateful Function Component)**

带状态的组件

在组件中要维护一个状态时，主要涉及两个 API 来保存状态，`useState`、`useRef`

useState

Try it on Codesandbox

`useState` 入参是**初始状态**。如果传入的是一个 function，则将此 function 的**返回作为初始状态**。

```
// 下面两行代码是等价的
const [count, setCount] = useState(0);
const [count, setCount] = useState(() => 0);
```

`count` 初始值为 `0`。function 传参**只会在组件首次渲染时执行**，也就是整个组件生命周期，function 传参**只会执行一次**。如果重绘的时候 `useState` 传参变了，也不会改变当前 `state` 的值。

当调用 `useState` 返回数组的第二个元素（即：`setCount`）时，React 会重绘当前组件，更新 html 文档，触发浏览器重绘。

可以看出，每次使用 `setCount` 时，都会重新执行 Component 的函数，所以，**绝对不能**在 Component 函数中同步调用 `setCount`，这样会导致无限重绘，页面假死。

```
function BadComponent() {
  const [count, setCount] = useState(0);
  setCount(count + 1); // Oops! Trigger rerender again and again
  return count;
}
```

实际上，页面的 UI 变更，总是有原因的：

- 用户触发的交互，如：键盘输入、鼠标点击、屏幕滑动等
- 定时器的触发，如：`setTimeout`、`requestAnimationFrame`、`setInterval`
- IO 事件回调触发，如：AJAX 请求返回的回调

总结就是，`setCount` 操作必须在某个回调中调用，不应该出现在 Component 函数的同步调用栈中执行。

以下情况 useState 需使用 function 作为入参

当初始状态需要复杂计算时

假设，count 的初始值需要根据 props 传入的数据相加来确定时，我们对比两种初始化方式：

```
// bad
const initialCount = props.data.reduce((acc, cur) => acc + cur, 0);
const [count, setCount] = useState(initialCount);

// good
const [count, setCount] = useState(
    () => props.data.reduce((acc, cur) => acc + cur, 0)
);
```

利用 function 传参只会执行一次的特点，组件重绘时就不需要再执行无用的 reduce 计算。

当初始状态是复杂对象时

创建一个复杂对象的性能开销是很大的。假设现在有一个初始 state 包含 50+ 个字段，我们就需要使用 function 只执行一次的特点，让这个复杂对象的声明只在第一次渲染时执行。（function 的声明所耗性能与 function 所含代码量无关的，但对象、数组是增长的）

> 各类型初始化 benchmark

```
// bad
const [initialState, setState] = useState({
    attr1: 'xxxx',
    attr2: 'xxxx',
    ...
    attr50: 'xxxx',
});

// good
const [initialState, setState] = useState(() => ({
    attr1: 'xxxx',
    attr2: 'xxxx',
    ...
    attr50: 'xxxx',
}));
```

关于 setCount 传入 function 的问题

上面的 `setCount` 除了可以传入 number 外，还可以传函数来更新状态：

```
setCount(oldCount => {
  const newCount = oldCount + 1;
  return newCount
});
```

传函数的好处是，当依赖 state 自身最新状态来更新状态时，不需要访问外部变量。举个例子：

Try it on Codesandbox

```
useEffect(() => {
  if (loading.loading) {
    const sub = interval(0, animationFrameScheduler)
      .pipe(
        take(201),
        map((n) => 200 - n)
      )
      .subscribe({
        next(d) {
          setLoadingTime(d);
        },
        complete() {
          ~~setCount(count + 1);~~
          setCount((count) => count + 1);
```

```
        }
      });
    return () => sub.unsubscribe();
  }
}, [loading]);
```

上面的代码，Loading 状态时点击 +1 按钮来增加 count，观察 loading 结束之后，count 数值变化。

然后将 Line 13 的代码换成 Line 14 的代码，再在 loading 时点击 +1 看下效果。

useRef

当需要存放一个数据，需要无论在哪里都取到最新状态时，需要使用 useRef。

ref 是一种可变数据。

首先我们来通过一个例子来解释一下函数组件中常见的闭包问题：

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

  // 这里 useEffect 表示在第一次渲染完成后，执行回调函数，具体 useEffect 用法下面讲
  useEffect(() => {
    const id = setInterval(() => {
        console.log(count);
        setCount(currentCount => currentCount + 1);
    });
    return () => { clearInterval(id); }
  }, []);
  return <h1>See what's printed in console.</h1>
}
```

> 观察 console 打印的值是什么 **See it on Codesandbox**

上面的代码，console 永远打印 0。因为函数声明时（第一次运行时），count 是 0，之后无论这个函数调用多少次，都会是 0。这时候，如果我们想要拿到 count 的最新值，就可以使用 useRef 声明一个可变数据对象，来存储 count。由于对象引用是不变的，当我们更新对象某个字段时，闭包函数就能访问到最新的值了。

代码改写如下：

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

  const countRef = useRef(count);
  countRef.current = count;

```

```
  // 这里 useEffect 表示在第一次渲染完成后，执行回调函数，具体 useEffect 用法下面讲
  useEffect(() => {
    const id = setInterval(() => {
        console.log(countRef.current);
        setCount(currentCount => currentCount + 1);
    });
    return () => { clearInterval(id); }
  }, []);
  return <h1>See what's printed in console.</h1>
}
```

避免基于可变对象的 ref 更新

对于 useRef 的值的更新，需要注意如果是在 Component 函数中同步赋值的情况，不要做基于其他任何可变数据的增量更新，比如：

```
// bad
countRef.current = countRef.current + 1;

// good
countRef.current = immutableState.count + 1;
```

因为在 StrictMode 下，React 每次渲染会执行两次 Component 函数，来检查函数组件的幂等性。这时基于可变数据的更新，会导致两次执行结果不一致，这是不允许的（会带来意想不到的更新结果，React 没有提供很好的 Warning 信息，很难排查）。**See it on Codesandbox**

不要使用 useRef 获取子组件 instance

React 社区有个组件约定，对于要拿到组件实例情况下，一般通过 ref 传参去取得某组件的实例。比如，对于 DOM Element，使用 ref 可以拿到 dom 实例。但这种方式并不推荐，进阶篇将讲解为什么

useState、useRef 如何决策用哪种来维护状态

useRef 生成的可变对象，因为使用起来就跟普通对象一样，赋值时候 React 是无法感知到值变更的，所以也不会触发组件重绘。利用其与 useState 的区别，我们一般这样区分使用：

- 维护与 UI 相关的状态，使用 useState

> 确保更改时刷新 UI

- 值更新不需要触发重绘时，使用 useRef
- 不需要变更的数据、函数，使用 useState

> 比如，需要声明一个不可变的值时，可以这样：

> `const [immutable] = useState(someState);`
>
> 不返回变更入口函数。useRef 虽然可以借助 TypeScript 达到语法检测上的 immutable，但实际还是 mutable 的。

组件通信

React 使用单向数据流进行 UI 绘制，只有父组件能控制子组件的状态，子组件不能修改父组件的状态。

单向数据流的优势在于不存在数据绑定、数据作用域等概念，这样使得首屏速度比双向绑定快。其次，排查问题更简单。但不足之处是交互组件的编写相对于双向绑定，比较啰嗦。

父子组件通信

父组件通过向子组件传递 props 通信。子组件通过对父组件暴露注册函数的接口来通知父组件更新自身状态

兄弟组件通信

通过将兄弟组件的状态放到父组件上来进行通信

爷孙组件通信

爷孙组件通信主要有 3 种方式：

1. 将孙子组件的 props 封装在一个固定字段中
2. 通过 children 透传
3. 通过 context 传递

假设有个三层组件，爷爷分别给儿子和孙子发红包

先看青铜解决方案：

See it on Codesandbox

```
function Grandpa() {
  const [someMoneyForMe] = useState(100);
  const [someMoneyForDaddy] = useState(101);
  return <Daddy money={someMoneyForDaddy} moneyForSon={someMoneyForMe} />;
}
function Daddy(props: { money: number; moneyForSon: number }) {
  const { money, moneyForSon } = props;
  return (
    <div className="daddy">
      <h2>This is Daddy, received ${money}</h2>
      <Me money={moneyForSon} />
```

```
    </div>
  );
}
function Me(props: { money: number }) {
  const { money } = props;
  return (
    <div className="son">
      <h3>This is Me, received ${money}</h3>
    </div>
  );
}
```

Daddy 组件会透传爷爷给孙子的组件给 Me。这种方案的缺点很明显，以后爷爷要给 Daddy 和 Me 发糖果的时候，Daddy 还得加字段。

方案一：将孙子组件的 props 封装在一个固定字段中

按照 1 的方案，我们可以固定给 Daddy 添加一个 sonProps 的字段，然后将 Grandpa 需要传给孙子的状态全部通过 sonProps 传递

```
function Grandpa() {
  const [someMoneyForMe] = useState(100);
  const [someMoneyForDaddy] = useState(101);
  return <Daddy money={someMoneyForDaddy} sonProps={{money: someMoneyForMe}} />;
}
function Daddy(props: { money: number; sonProps: Parameters<typeof Me>[0]; }) {
  const { money, sonProps } = props;
  return (
    <div className="daddy">
      <h2>This is Daddy, received ${money}</h2>
      <Me {...sonProps}/>
    </div>
  );
}
function Me(props: { money: number }) {
  const { money } = props;
  return (
    <div className="son">
      <h3>This is Me, received ${money}</h3>
    </div>
  );
}
```

这样以后要给 Me 加字段，就不用改 Daddy 了。但要测试 Daddy 时还得 mock Me 组件的数据，Daddy 和 Son 耦合。

方案二：通过 children 透传

children 类似于 vue 中的 slot，可以完成一些嵌套组件通信的功能

```
function Grandpa() {
  const [someMoneyForMe] = useState(100);
  const [someMoneyForDaddy] = useState(101);
  return (
    <Daddy money={someMoneyForDaddy}>
      <Me money={someMoneyForMe} />
    </Daddy>
  );
}
function Daddy(props: { money: number; children?: React.ChildNode }) {
  const { money, children } = props;
  return (
    <div className="daddy">
      <h2>This is Daddy, received ${money}</h2>
      {children}
    </div>
  );
}
function Me(props: { money: number }) {
  const { money } = props;
  return (
    <div className="son">
      <h3>This is Me, received ${money}</h3>
    </div>
  );
}
```

将 Daddy 的嵌套部分用 children 替代后，解耦了子组件和孙子组件的依赖关系，Daddy 组件更加独立。

方案三：通过 context 透传

Try it on Codesandbox

useContext、createContext

使用 context 分三步

STEP 1 声明 context

使用 createContext 声明一个 Context

`const MyContext = React.createContext({})`

STEP 2 将 Provider 包在顶层

```
<MyContext.Provider value={xxxx}>
</MyContext.Provider>
```

STEP 3 通过 useContext 获取透传数据

```
const contextValue = useContext(MyContext);
```

改写之后的代码变为：

```
const Context = createContext({
  moneyForDaddy: 0,
  moneyForMe: 0
});

function Grandpa() {
  const [moneyForMe] = useState(100);
  const [moneyForDaddy] = useState(101);
  return (
    <Context.Provider value={{moneyForDaddy, moneyForMe}}>
      <Daddy>
        <Me />
      </Daddy>
    </Context.Provider>
  );
}
function Daddy(props: { children?: React.ChildNode }) {
  const { children } = props;
  const ctx = useContext(Context);
  return (
    <div className="Daddy">
      <h2>This is Daddy, received ${ctx.moneyForDaddy}</h2>
      {children}
    </div>
  );
}
function Me() {
  const ctx = useContext(Context);
  return (
    <div className="son">
      <h3>This is Me, received ${ctx.moneyForMe}</h3>
    </div>
  );
```

```
34 }
```

使用 context 之后，Daddy 和 Me 组件的没有任何依赖，而且即使之后改变组件层级关系，只要还在 Provider 下，就没有任何影响。

三种方案的决策

1. 第一种方案一般用于固定结构和跨组件有互相依赖的场景，多见于 UI 框架中的复合组件与原子组件的设计中
2. 第二种常用在嵌套层级不深的业务代码中，比如表单场景。优点是顶层 Grandpa 的业务收敛度很高，一眼能看清 UI 结构及状态绑定关系，相当于拍平了 React 组件树
3. 第三种比较通用，适合复杂嵌套透传场景。缺点是范式代码较多，且会造成 react dev tools 层级过多；Context 无法在父组件看出依赖关系，必须到子组件文件中才能知道数据来源

副作用的处理

useEffect

useEffect 传入的回调会在每次渲染生效之后执行。常见的用法有：

1. AJAX 请求
2. 动画效果
3. 触发数据同步

AJAX 请求范例

```
1  function usePageData(params: { pageIndex?: number; }) {
2    const { pageIndex = 1 } = params;
3    const [loading, setLoading] = useState(false);
4    const [data, setData] = useState({});
5
6    useEffect(() => {
7      let canceled = false; // 用来标记是否异步回调已过期
8      setLoading(true)
9      fetchData(pageIndex).then((resp) => {
10       if (canceled) {
11         return;
12       }
13       setLoading(false);
14       setData(resp.data);
15     });
16     return () => canceled = true;
17   }, [pageIndex]);  // 只有在 pageIndex 变更的时候发起请求
18
19   return { loading, data };
```

```
20 }
```

这个 🌰 中我们添加了一下新的东西：

1. Line 1: 这个函数以驼峰形式命名，以 use 开头，React 中，这类函数我们称之为 hook 组件。与 Component 组件相比，hook 组件更贴近普通函数，它对于入出参没有任何限制。hook 组件更像是面向过程编程中的一段代码。当发现某个 Component 组件中的某段代码可以复用时，可以很方便的 copy and create 一个 hook 组件，进行复用。
2. Line 17: 通过 useEffect 的第二个参数（通常称之为 dependencies，简称 deps），声明我们只根据 pageIndex 是否变更，来决定 effect 是否执行。通常情况下，我们需要把所有 effect 中用到的闭包变量，添加在 deps 数组中。但这也并非绝对的。
3. Line 7、9、16: 我们在 effect 函数中声明了一个变量 `canceled` 来标记此 effect 是否已经过期。所谓过期，就比如：pageIndex 从 1 变成 2 的时候，pageIndex = 1 时的 effect 就是过期的 effect。因为 effect 通常存在异步调用，那么异步函数的回调就要确保不影响 UI 正常渲染。比如这里 pageIndex = 1 的请求耗时 3s，而 pageIndex = 2 的请求耗时 1s，且 pageIndex 从 1 变为 2 的间隔只有 1s，那么此时，fetchData 先执行 pageIndex = 2 时的回调，再执行 pageIndex = 1 时的回调。这就导致最终渲染的是 pageIndex = 1 的数据结果，与预期不符。这里，通过 canceled 标志位，在回收阶段设置为 true，在异步回调的时候再进行判断，来达到回收异步回调的效果。异步回收相关知识在进阶篇单独详细讲解各种场景如何处理。

与其他库结合时，处理异步回调的常规操作

```
1  import { useState, useEffect } from 'react';
2  import { fromEvent } from 'rxjs';
3
4  function useWindowSizeChange(handler: (width: number, height: number) => void) {
5    const [width, setWidth] = useState(() => window.innerWidth);
6    const [height, setHeight] = useState(() => window.innerHeight);
7
8    useEffect(() => {
9      const subscription = fromEvent(window, 'resize').subscribe(handler);
10     return () => subscription.unsubscribe(() => {
11         setWidth(window.innerWidth);
12         setHeight(window.innerHeight);
13     });
14   }, [handler]);
15   return [width, height];
16 }
```

useLayoutEffect vs. useEffect

[Try it on Codesandbox]

useLayoutEffect 和 useEffect 的传参一致，但有以下区别

1. 执行时机不同。useLayoutEffect 的入参函数会在 react 更新 DOM 树后同步调用。useEffect 为异步调用
2. useLayoutEffect 在 development 模式下 SSR 会有警告⚠

通常情况下 useLayoutEffect 会用在做动效和记录 layout 的一些特殊场景。一般不需要使用 useLayoutEffect。

useMemo

useMemo 主要有两个作用：

1. 缓存一些耗时计算，通过声明计算结果的依赖是否变更，来重用上次计算结果
2. 保证引用不变，针对下游使用 React.memo 的组件进行性能优化（useCallback 也有一样的作用）

比如，计算耗时的 fibonacci 数列，就可以用 useMemo 来优化在 n 不变的情况下，二次渲染的性能

```
useMemo(() => {
  return fibonacci(props.n)
}, [props.n]);
```

useCallback

useCallback 是简化版的 useMemo，方便缓存函数引用。下面的代码是等价的：

```
const memoCallback = useCallback((...args) => {
  // DO SOMETHING
}, [...deps]);
```

```
const memoCallback = useMemo(() => (...args) => {
  // DO SOMETHING
}, [...deps]);
```

> 在么有遇到性能问题时，不要使用 useCallback 和 useMemo，性能优化先交给框架处理解决。手工的微优化在没有对框架和业务场景有深入了解时，可能出现性能劣化。
>
> 致命的 useCallback/useMemo（翻译）
>
> useCallback hell问题总结
>
> 关于如何减少 useCallback 看 **第二天**

组件的生命周期

React 函数组件的执行阶段分为：

1. Render 阶段

   此阶段就是函数本体的执行阶段

2. Commit 阶段

   Commit 阶段是拿着 render 返回的结果，去同步 DOM 更新的阶段。render 和 commit 分开以达到批量更新 DOM 的目的，也是 react 之后推出并行模式的设计基础。对于我们代码能感知到的部分就是 useLayoutEffect

3. DOM 更新结束

   此时 DOM 已经更新完成，~~代码能感知到的部分~~ 代码上的体现就是执行 useEffect

第二天：React 生态

React Dev Tools

Chrome、FireFox、Edge 浏览器均有 React Dev Tools 插件。此插件能帮助我们快速定位 ReactComponent、查看 Component 当前状态、查找性能瓶颈。

安装

Chrome

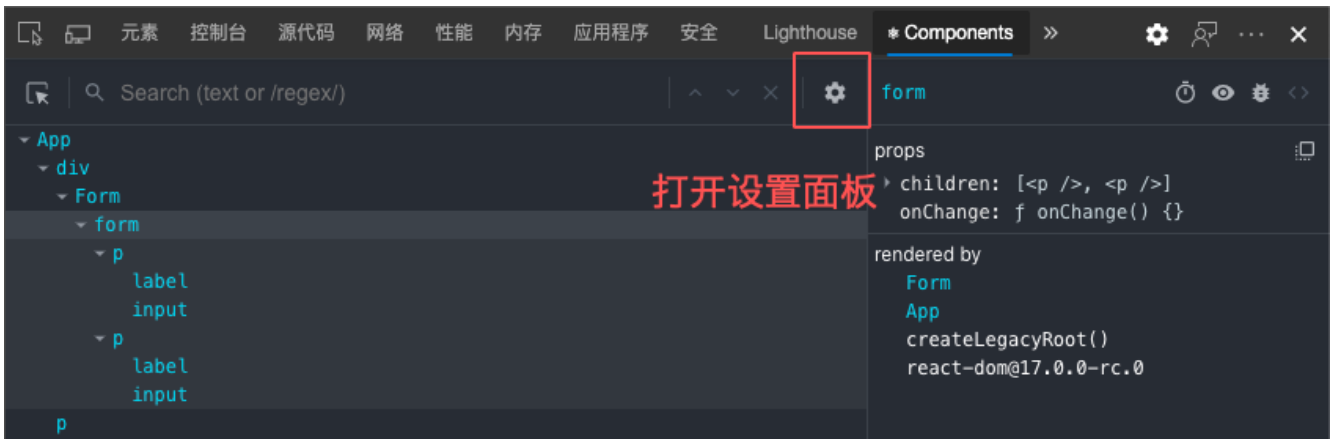
FireFox

Edge

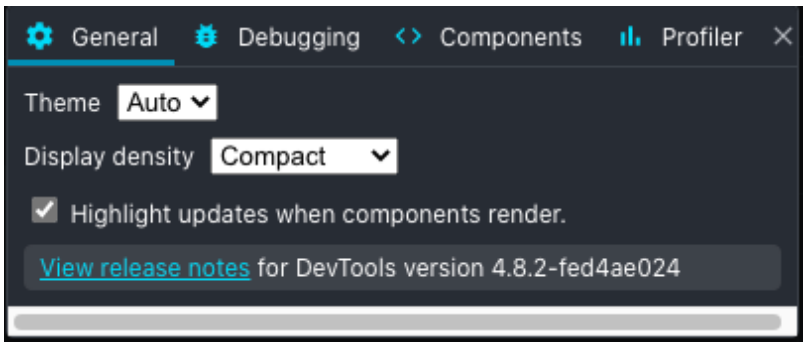
安装成功会在地址栏右侧看到一个 react 图标（对，就是比 CCTV 多个圈的那个）

打开浏览器 DevTools，查看面板上是否有 `* components` 和 `* Profiler`，两者对应的分别是 **React 组件检视器**和 **React 性能面板**

设置

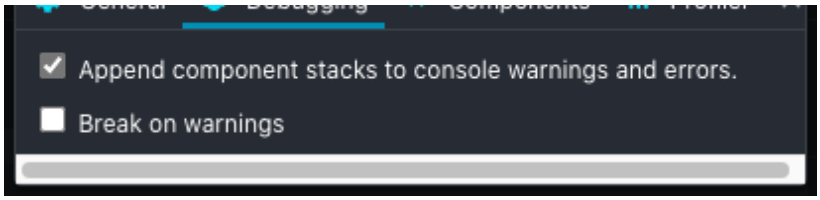


General 设置



General 面板中最重要的功能就是 **"Highlight updates when components render"**。勾选上之后，可以查看 React 重绘时，页面哪些部分有更新。在遇到性能问题时，可以快速帮助决策在哪部分不需要重绘的组件部分添加 React.memo 阻止重绘。

Debugging 设置



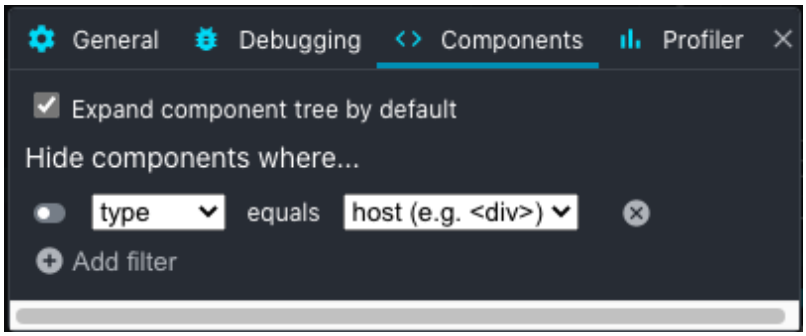
"Append components stacks to console warnings and errors."

可以帮助我们定位 React 报错信息来自哪个组件

~~"Break on warnings"~~

~~笔者也不知道啥作用...没有遇到开启和关闭会不同的 warning case...~~

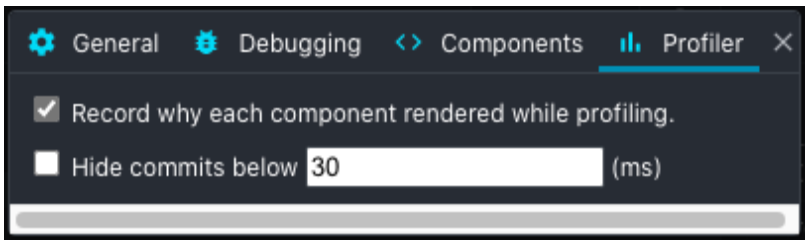
Components 设置



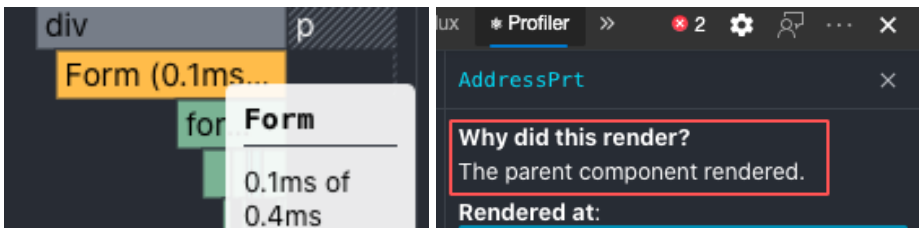
"Hide components where..."

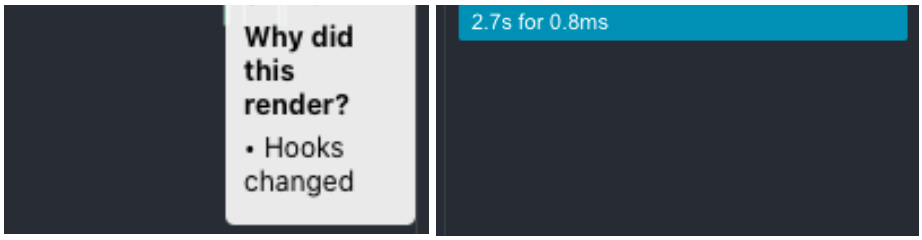
此项在过滤组件树时有用。比如，大多数应用可能会在顶层有很多 Context.Provider，就可以过滤 Context 类型、不显示。当我们只关心业务逻辑层的组件时，可以过滤掉 DOM 组件，减少树的节点

Profiler 设置



"Record why each component rendered while profiling"





勾选之后会在渲染火焰图的 hover 面板中看到 **"Why did this render"**。

现在有哪些 render 原因呢?

1. Props changed

   顾名思义，传入组件的 props 变更

2. The parent component rendered

   父组件渲染导致的子组件渲染。一般要做性能优化都是找这类重绘原因的组件。但是要注意，如果组件中有用到 useContext，Provider 的 value 变更导致的重绘也是被标记为 The parent component rendered，需要注意

3. Hooks changed

   Hook 状态变更导致的重绘，一般就是指 useState 返回的更新函数被调用了

4. ~~State changed~~

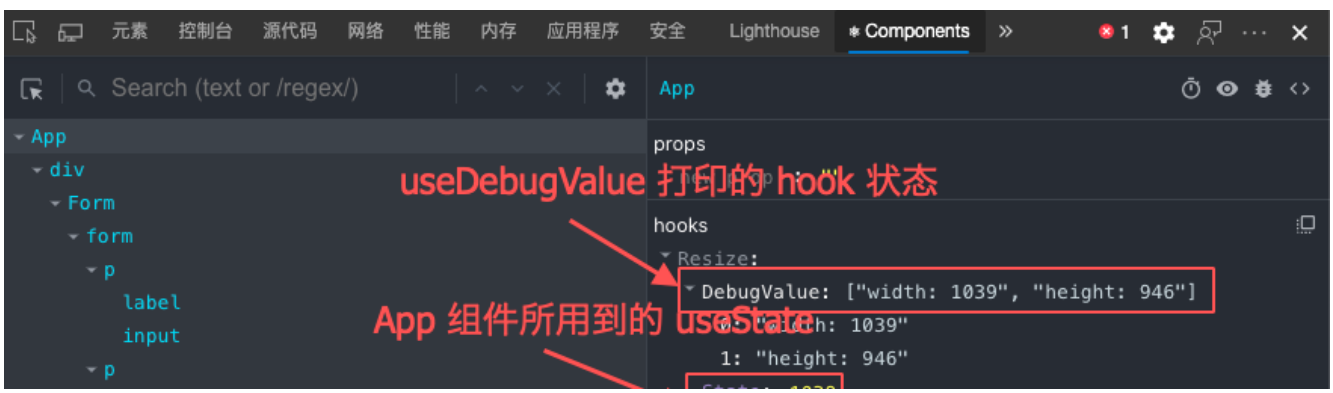
   ~~这个只会在 class component 中有，大家忽略~~

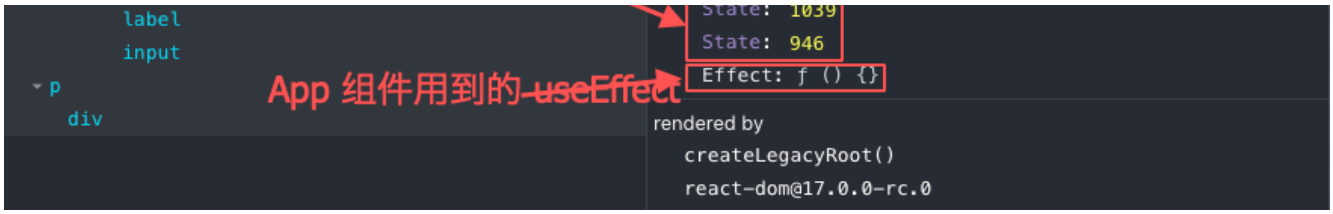
"Hide commits below xx (ms)"

设置一个更新耗时的阈值，低于该阈值的渲染不显示。用来快速过滤哪些渲染有性能问题

React 组件检视器

用法和 DevTools 的元素面板类似，可以直接在页面上定位到元素对应的 React Component。并且可以实时查看当前组件内部的 hooks 状态，返回的组件树





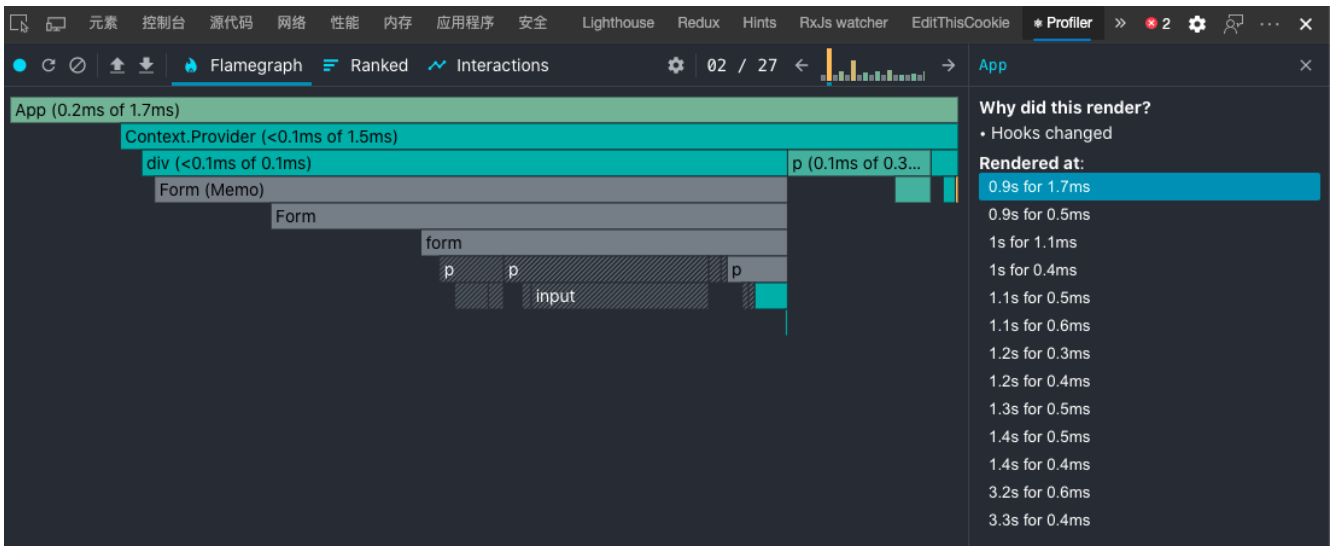
右上角的四个图标

分别作用如下：

- 模拟 ReactLazy 懒加载组件 Suspense 状态
- 定位组件所渲染的 DOM 节点
- 在 Console 中打印 Component 内部状态
- 跳转到组件所在源文件（配合 sourcemap）

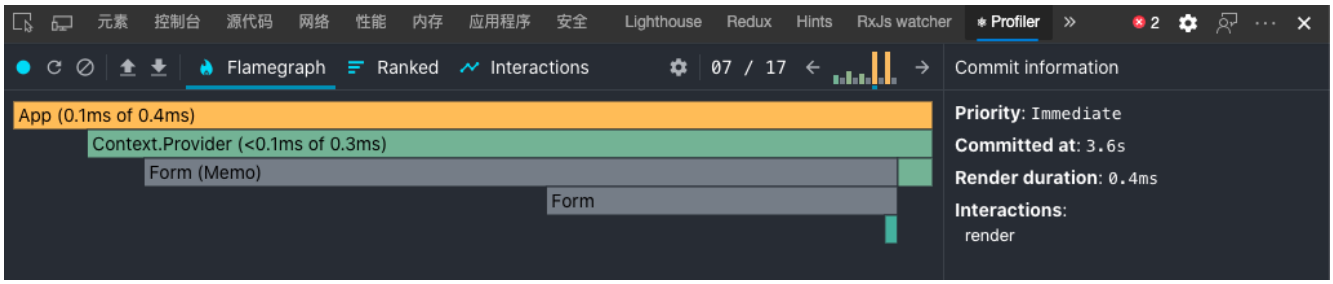
React 性能分析面板

Profiling 面板如下图：



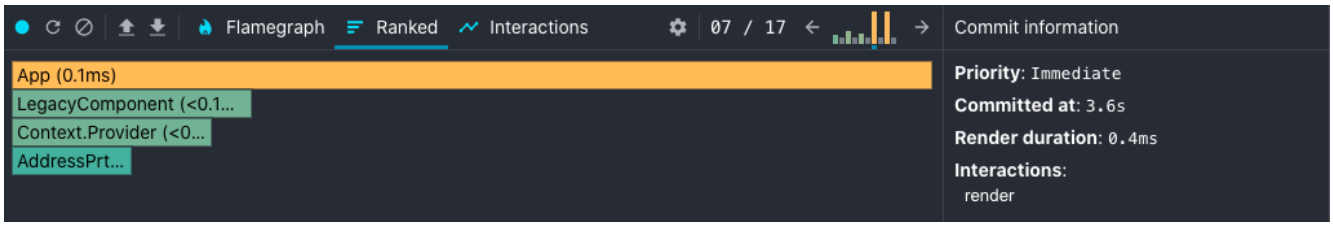
Toolbar 部分的功能和 Chrome dev tools 的 performance panel 一模一样，分别是开始录制、刷新页面并录制、清除记录、加载 Profiling 数据、下载 Profiling 数据。

Flamegraph



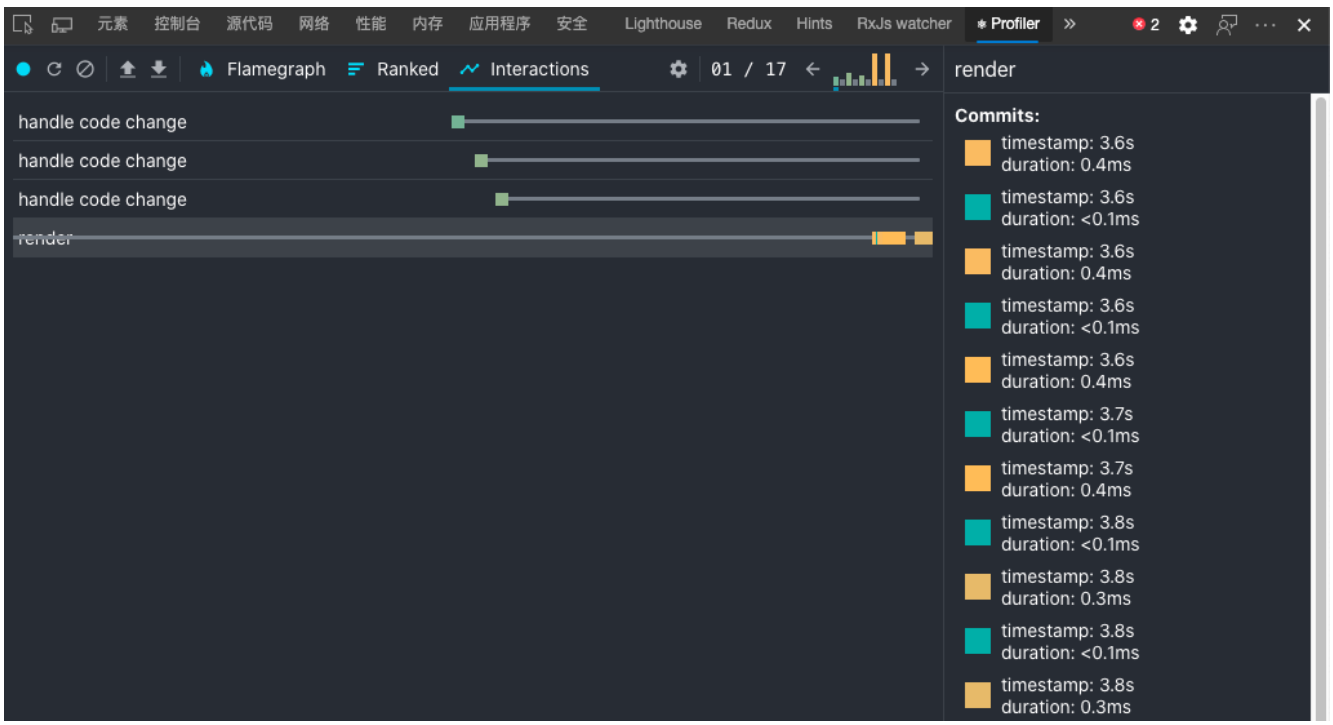
查看组件层级的耗时及关联层级、累加关系。颜色深浅代表耗时长短，是一个相对的着色，跟具体耗时没关系（比如，0.1ms 可能是黄色，16ms 可能是绿色，具体要看最长耗时的度量是多少）

~~Ranked~~



~~纯看每个组件的渲染速度排序，没有任何层级关联关系，意义不大~~

Interactions



显示交互信息对于每次渲染的影响（目前 R17-rc.0 该功能不可用，有 bug）

这块需要写专门跟踪 interactions 的代码，具体用法：

1. 单次触发重绘的性能跟踪

```
1 import React, { useState } from 'react';
2 import { unstable_trace } from 'scheduler/tracing';
3
```

```
function Counter() {
  const [count, setCount] = useState(0);
  const handleClick = () => {
    unstable_trace('handle click', performance.now(), () => {
      setCount(s => s + 1);
    });
  }

  return <h1>Clicked Times: {count}</h1>
}
```

2. 流式触发性能跟踪，适合需要对某个连续异步流程做跟踪或者相关联的异步任务做渲染比较的场景

```
import React, { useState, useEffect } from 'react';
import { unstable_trace, unstable_wrap } from 'scheduler/tracing';

unstable_trace('trace an action stream', performance.now(), () => {
  const wrappedWorkflow1 = unstable_wrap(async(...args) => { /* do something */});
  const wrappedWorkflow2 = unstable_wrap(async(...args) => { /* do something */});
  wrappedWorkflow1(params).then(resp => {
    return wrappedWorkflow2(resp);
  });
});
```

> 不用担心生产环境的性能问题，**scheduler/tracing** 本身做了 production 和 development 打包的区分，production 打包会是一些透传函数，不起实际作用

Try it on Codesandbox

在上面的 Codesandbox 中，对所有 resize 动作当做一个 stream 动作流进行 profiling。对列表中的 Next、Prev、PageSizeChange 做了单次操作的 profiling。可以点击 Codesandbox 预览面板部分的 "Open In New Window" 打开 React DevTools 查看 Profiling 面板的效果。

组件性能优化

React 组件是一个树形结构，且每个节点都是懒计算的（类似于 Thunk 的概念）。当一个节点不需要重新计算（重绘）时，他的子树都不会计算（重绘）。**所以我们做性能优化的目标，就是在尽量离根节点近的位置，拦截不必要的节点重算，从而减少重绘的计算量。**

React.memo

阻止节点重绘主要通过 React.memo 方法生成特殊的组件节点。它接受两个传参：

```
React.memo(Component, areEqual);
```

1. Component

   组件

2. areEqual

   比较函数，比较函数的入参有两个，arg0 为 前一次渲染的 props, arg1 为本次渲染的 props。如果返回 true，则该节点本次渲染将被标记为无需重新计算，从而使其所有子节点、孙子节点都无需计算。

   areEqual 如果不传，默认使用

```
(prevProps, nextProps) => shallowEqual(prevProps, nextProps)
```

   做为比较函数。

如何定位优化点

使用前一章节提到 React DevTools 中的 Profiling 功能，record 发生卡顿的操作，从耗时长的组件逐个查看，找到那些跟此次操作无关的上层渲染节点，尝试使用 React.memo 包裹这些组件。

不要为了优化而优化

在没有性能问题前，不用去纠结是否要用 Profiling、React.memo、useMemo、useCallback 去优化性能，这些不一定能带来性能提升，反而肯定会带来首屏的性能下降。大多数情况下，React 现有算法已能满足性能需求。

对于一个组件，有三样东西会让她重绘

1. State 变更
2. 依赖的 context 变更
3. 父组件重绘

所以用 React.memo 包裹之后，并不是说性能就会有多大的提高。如果组件中依赖的 context 中，有一部分并不是此组件需要的数据，但会经常变更，也会导致组件经常重绘。这时候我们可以增加一层组件，把依赖 context 中的数据，通过增加的一层父组件取出来，然后通过 props 传给真正渲染的组件，把 React.memo 加在真正渲染的组件上，来达到屏蔽 context 变更引起的重绘问题。

```
import { useMemo, useContext } from 'react';
import { SomeContext } from './SomeContext';

function PickContextData(props) {
  const ctx = useContext(SomeContext);
```

```
  const someDataFromContext = useMemo(() => {
    return ctx.data;
  }, [ctx.data]);
  return <RenderComponent data={someDataFromContext} {...props} />
}

const RenderComponent = React.memo((props) => {
  // 略
});
```

更通用点，可以封装出 react-redux 的 connect 函数，传入 selector 来取所需的 Context 数据。

通过 reducer 收敛业务逻辑

在复杂组件中，随着 state 的增加，常常会导致以下问题：

1. useCallback/useMemo/useEffect 的依赖图谱逐渐复杂
2. useCallback/useMemo/useEffect 形成层叠关系的依赖，找不到源头，或者写了多余的依赖，难以梳理依赖关系

```
const [state1, setState1] = useState();
const [state2, setState2] = useState();
const fn1 = useCallback(xxxx, [state1.something, state2.other]);
const fn2 = useCallback(xxxx, [fn1]);
const fn3 = useCallback(() => {fn1(state1); fn2(state2);}, [fn1, fn2, state1, state2]);

useEffect(xxxx, [fn1, fn2, fn3]);
```

> 上面的代码在 CodeReview 过程中经常看到。
>
> 一旦开始用 useCallback/useMemo，就发现为了引用不变，deps 开始病毒传播...

随着 deps 的增多，代码维护成本、理解成本也直线上升。这种情况下，我们可以通过 reducer 函数来收敛逻辑，减少 deps

1. 首先，我们编写 state 的赋值逻辑，这里，我们将所有页面用到的 useState 状态都放到一起

```
// 定义 reducer
const reducer = (state, action) => {
  const { type, payload } = action;
  switch(type) {
    case 'fn1': {
      // return nextState;
    }
    case 'fn2': {
```

```
      // return nextState;
    }
    case 'fn3': {
      // return nextState;
    }
  }
}

// 如果配合 immer，会更香
import produce from 'immer';
const reducer = produce((draft, action) => { });
```

2. 然后，我们创建类似 redux 的 dispatch 方法

```
const [state, setState] = useState(() => ({ state1, state2 }));
const { current: dispatch } = useRef((action) => {
  setState((currentState) => {
    return reducer(currentState, action);
  });
});
```

> 有同学这里可能会问，为什么不用 useReducer？因为 useReducer 返回的 dispatch 传参只能传一个，有时候就是希望有多个传参。
>
> 如果要做一些骚操作，需要再封装一次 useReducer 的 dispatch。所以这里我一般用 useState 来承载 reducer 逻辑，方便在函数中插入特殊需求（比如做变更日志记录、undo、redo）

如果代码中，有许多 deps 来自于不同的 useState，那就可以通过把 useState 合并在一起，通过 setState 传入函数，来获取当前最新 state 的状态，从而减少这部分的 deps。

React 常见 TypeScript 问题

详见：React@16.8.4+ 常用 Typescript 定义

常用库

react-router

react-router 是用来处理 React 应用单页路由跳转的核心包。在浏览器环境，我们需要安装 react-router-dom。

官网地址：https://reacttraining.com/react-router/web

redux

Redux 是一个流行的状态管理库，在较复杂的应用中，为了管理全局应用的状态，会使用到。Redux 由于其简单的 api 和强大中间件的扩展机制，已经衍生出了很多基于 Redux 设计理念的生态库、其他平台的实现（flutter/redux、vuex），比如:

- Reduck
- dva
- rematch

官网地址：https://redux.js.org/

immer

immer 常用来做复杂数据的更新，能帮助你更新复杂数据的整个引用信息，方便做 shallowEqual。

第三天：进阶

最佳实践

将其他 UI 库封装为 React 组件

UI 库无外乎输入一些配置 + 某个 DOM 节点，渲染出其他 DOM。我们只要抓住 UI 库的 update、destroy 方法，用 useEffect 在 render 之后调用 update 方法，在卸载的时候调用 destroy 方法，就基本完成了 React 化的封装。如果没有 destroy 方法，就用 key 去强制卸载组件。

举个简单的例子，我们有个 VanillaJS 风格的 Tooltip 库，需要将它封装成 React 组件，思路是这样的：

1. 找到 Tooltip 所有触发 UI 更新的 update 方法
2. 找到 Tooltip 的 destroy 方法
3. 将 Tooltip 支持的 props 设置为 ReactTooltip 的 props 类型
4. 在 ReactTooltip 的 useEffect 中将影响 UI 的 props 作为 dependencies，调用 Tooltip 的 update 方法
5. 在 useEffect 的回调函数的返回函数中，调用 destroy 方法

下面的 Codesandbox 中，VanillaTooltip.ts 作为一个普通的 UI 库，提供简单的 tooltip 功能。Tooltip.tsx 文件对 VanillaTooltip 进行了封装，使之可以当 React 组件使用

Try it on Codesandbox

异步操作回收

浏览器环境中异步执行的代码无非以下几种：

1. 计时器 setTimeout / setInterval / requestAnimationFrame
2. Promise fufilled 或者 rejected 后回调

3. 各种浏览器原生事件
4. 各种 XXXObserver 的回调函数

计时器回收

针对第一类，计时器，我们可以用对应的注销方法来回收计时器的异步回调

```
useEffect(() => {
  const id = requestAnimationFrame(startAnimate1);
  const timerId = setTimeout(startAnimate2);
  const intervalId = setInterval(startAnimate3);
  return () => {
    cancelAnimationFrame(id);
    clearTimeout(timerId);
    clearInterval(intervalId);
  }
}, []);
```

插桩回收

针对 Promise 这类没有原生回收方案的的异步操作，比较通用的方法是插桩回收

```
useEffect(() => {
  let canceled = false
  fetchData(pageIindex).then(resp => {
    if (canceled) {
      return;
    }
    // DO SOMETHING UI UPDATE
  });
  return () => canceled = true;
}, [pageIndex]);
```

上面的代码，定义了一个标志位（canceled），通过函数闭包在异步回调的时候，判断是否标志位已经过期（canceled = true），如果过期，不执行回调。

这是一种比较通用的做法，针对计时器的回调也适用。

```
useEffect(() => {
  let canceled = false;
  const id = requestAnimationFrame(() => {
    if (canceled) return; startAnimate1();
  });
  const timerId = setTimeout(() => {
    if (canceled) return; startAnimate2();
  });
  const intervalId = setInterval(() => {
```

```
    if (canceled) return; startAnimate3();
  });  // setInterval 不能这么玩，必须用 `clearInterval` 清掉
  return () => canceled = true;
}, []);
```

此外，也可以利用一些第三方扩展的异步工具库，处理 cancel。比如：axios 的 cancelToken，rxjs 的 unsubscribe、takeUntil，bluebird 的 cancel 等。

深入原理

Valid JSX Element

一个 JSX Element 合法的返回类型有：

- ReactElement (`<Component />`)
- 数字 (`1`)
- 数组 (`[1, <Component />, 'str', null, [11, <Component />, false]]`)
- 字符串 (`"string"`)
- null (`null`)
- false (`false`)

但由于 TS 定义问题，如果我们一个 Component 返回的是数字、数组、字符串、false，不能以 `<Component />` 方式调用，只能 `{Component()}` 调用

Fragment 的作用

如果觉得 `{Component()}` 这种方式调用很不爽，不整齐，可以考虑用 Fragment 包裹一下。

```
<Fragment>{1}<Component />{'str'}{null}{[11, <Component />, false]}</Fragment>
```

这样就能绕过 TS 类型检查的问题。

Fragment 实质上是个特殊渲染片段，相当于以数组的方式包裹一组组件进行渲染。

key 和 ref

React 组件中 props 有两个保留字段，key 和 ref。

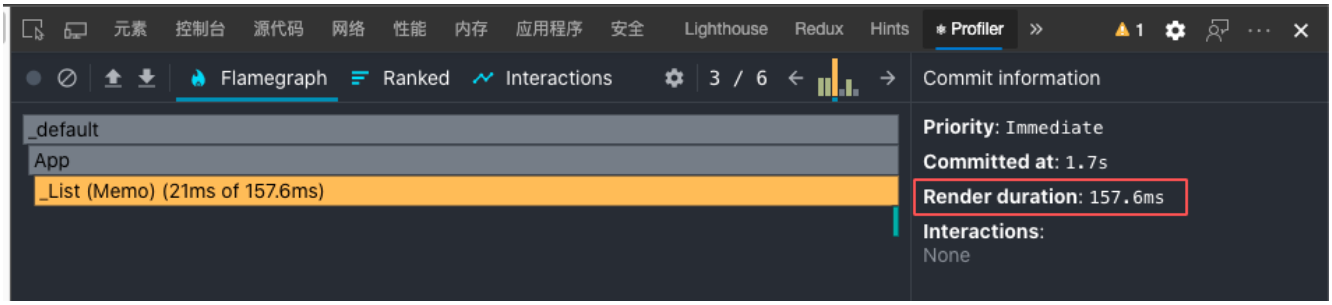
key

key 是用来追踪 React Component 和实际渲染的 DOM 节点用的。默认使用组件所在位置进行标记。

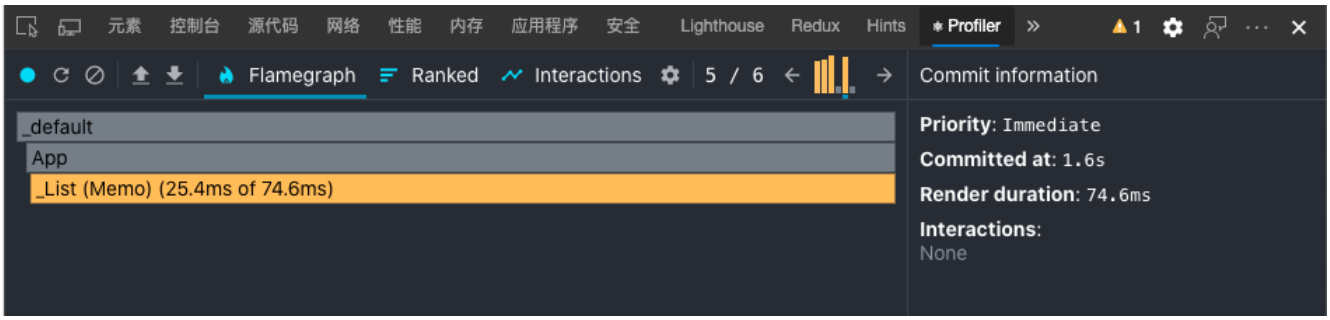
在渲染数组数据时，提供 key 可以提升 React 复用 DOM 节点的能力。

Try it on Codesandbox

上面的 Codesandbox 中，当一页有 3000 条数据时，使用 id 作为 key 的时候，翻页渲染性能数据如下：



当使用 index 作为 key 时，翻页渲染性能如下：



可见 index 作为 key 比 id 快一倍

原因是当组件树某位置的 key 跟之前渲染的同位置节点有变更时，react 会认为源组件不可复用，会执行完整的 unmount 步骤，删除包括真实 DOM 节点在内的所有数据，完全重新初始化该节点。这个性能差距会随着节点复杂度成几何级别的增大。所以，**不要听信一些最佳实践所谓的要将 id 作为 key 渲染。弄清楚 react 运行的原理，才能做出恰当的选择。**

> 总结一下：
>
> 当渲染的组件是**完全受控的组件**时，就**应该用 index 作为 key**，以最大限度复用已有节点数据。
>
> 当渲染的组件有内部 state 时，可以通过改变 key，来重置组件内部 state。

ref

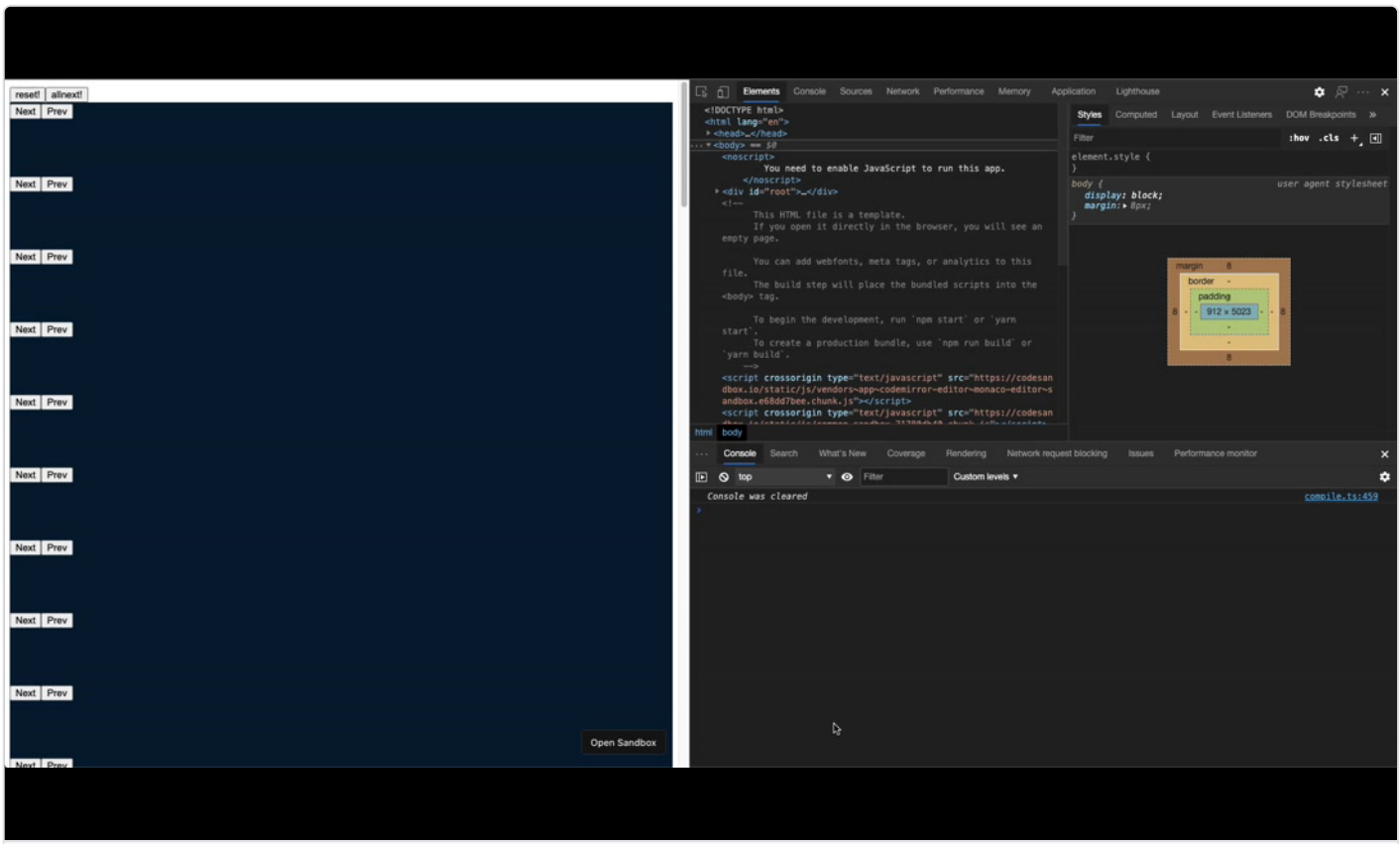
ref 一般用来获取 DOM 节点。

react 本质上将 ref 作为 Mutable 对象来看待，通过 ref 可以反向将子组件的内部方法和状态通过 Mutable 的 ref 传递给父组件。

如果是自定义组件，在这里不推荐用 ref，因为写起来麻烦，且容易内存泄漏。

> 补内存泄漏的例子：https://codesandbox.io/s/how-ref-cause-mmo-hhqd3

> 例子中，MemoryLeakComponent 组件通过 ref 向外暴露内部状态（button DOM 节点）。Row 组件在 onMount 时把 MemoryLeakComponent 的 button 点击函数封装暴露给了 App。App 持有了 onclick 点击函数。
>
> 页面中通过 reset all 按钮改变 Row 组件的 key 来达到替换 Row 组件的目的。
>
> 打开 DevTools Performance Monitor，观察 JS EventListener、JSHeap、DOM Nodes 的变化：



📎屏幕录制2021-01-19 下午3.25.33.mov

举个 ref 例子：现在需要对外暴露自定义组件的 reset 方法来重置内部状态

```
const initialState = Object.freeze({/*initalState */});

function _Component(props, ref) {
  const [internalState, setInternalState] = useState(initialState);

  useImperativeHandle(ref, () => {
    return {
      reset: () => { setInternalState(initialState); }
      getCurrentState: () => { return internalState; }
    };
  }, [internalState]);

  return <SomeUI />;
}

export const Component = React.forwardRef(_Component);
```

> 在 TS 中，使用 React.ForwardRefRenderFunction 类型来定义 ref 组件

~~forward-ref 这里是 TS 体操广场~~

为了使用 ref 这个 props 中的保留字，我们需要增加一个组件名 _Component，增加一个跟 context 位置重叠的第二个形参，React DevTools 中也会增加一层组件。

第二个形参在有 Component.ContextTypes 存时，会传入 context。这种情况下，这里的 ref 到底是 Context 还是 ref 就有点微妙了。容易出问题。

所以，在封装自定义组件时，完全可以不用 ref，自己添加一个 props 属性替换掉 ref 即可：

```
function Component(props~~, ref~~) {
  const [internalState, setInternalState] = useState(initialState);

  useImperativeHandle(props.withRef, () => {
    return {
      reset: () => { setInternalState(initialState); }
      getCurrentState: () => { return internalState; }
    };
  }, [internalState]);

  return <SomeUI />;
}

~~export const Component = React.forwardRef(_Component);~~
```

这样避免 context 形参位置的冲突，少了一层 forwardRef 的组件层级，少了一次纠结怎么起变量名字的过程。

原则上尽可能避免子组件给父组件添加自身内部方法和数据的行为。通过 props 暴露子组件行为才是正道。比如：要暴露 focus 方法的话，可以像 input[type="checkbox"] 的 checked 和 onChange 那样去实现：

```
function MyInput(props) {
  const { focus, onFocusChange, ...restProps } = props;
  const inputRef = useRef(null);

  useEffect(() => {
    onFocusChange();
  }, [focus]);

  return <input ref={inputRef} {...restProps} onBlur={() => {
    onFocusChange(false);
  }} onFocus={() => { onFocusChange(true); }} />
}
```

这样修改后，如果要 focus 到 MyInput 组件，就可以从拿着组件的 ref 去命令式的调用 `ref.current.focus()`

改成声明式的

`<MyInput focus={isFocus} onFocusChange={handleFocusChange} />`

ref 的正确使用方式及副作用回收

ref 作为 React 中唯一的 Mutable 传递方式，形成了一套独特的使用范例。如果传给 ref 的是一个函数，这个函数的调用有以下规则：

1. 如果 ref 函数跟上一次的 ref 函数不一致（引用比较），那么会在上一次渲染的 useLayoutEffect / useEffect 的回收函数调用后调用，且调用参数为 null，在本次渲染的 useLayoutEffect / useEffect 的回调函数之前，用 reference 实例调用一次 ref 函数。
2. 如果 ref 函数跟上一次的 ref 函数一致，则重绘时不会调用 ref 函数

Try it on Codesandbox

这个例子中，可以看到，在点击 forceUpdate 触发组件重绘的时候，anonymous 所在的 ref 会被调用两次，而 standalone 不会。当点击 hide / show 触发组件卸载和挂载的时候，两个 refCallback 都会被调用。

最后的最后，为了最大限度保证在使用 ref 时不会有内存泄漏，我们应该避免使用 useRef 来获取 ref，因为这样就少了 ref 是 null 的处理步骤。使用函数（如下面的 refCallback），通过 TS 的类型时刻提醒自己 ref 是 null 的处理。

```
const refCallback = useCallback((reference: null | YourReferenceType) => {
  if (reference === null) {
    dispose();
  } else {
    reference.doSomething();
  }
}, []);

<input ref={refCallback} />
```

受控与非受控的决策

受控组件：

没有内部状态或内部状态完全由 props 决定的组件

非受控组件：

存在不受 props 控制的内部状态的组件

受控和非受控常见于与用户交互相关的组件中，典型的例子是原生的 input 组件，根据不同写法，可以是受控也可以是非受控

```
/* 受控的 input 写法 */
<input value={text} onChange={handleChange} />

/* 非受控 input 写法 */
<input defaultValue={initialText} onChange={handleChange} />
```

React 社区已经形成一种共识，如果一个组件既可以受控也可以非受控运行，一般传入 defaultValue/defaultChecked 的表示运行在非受控模式，传入 value/checked 和 onChange 表示运行在受控模式。

受控组件的优缺点

优点

受控组件由于完全受父组件的传参控制，意味着使用多个受控组件时，可以在父组件自然而然的访问、修改所有组件状态。当有多个受控组件状态通信、联动的时候，父组件可以方便的根据需求更新子组件状态。

缺点

组件状态不闭环，性能差。受控组件所有状态存放在父组件，导致受控组件需要更新 UI 时，需要通过触发父组件的状态更新来更新自身，父组件的更新会触发所有子组件更新。

性能问题常见在 CRUD 列表和复杂表单业务中出现。受控组件即使完全独立于其他兄弟组件，更新时也会触发兄弟组件的重绘。

使用较复杂，因为 props 传参多。不利于父组件分离关注点。

非受控组件优缺点

优点

非受控组件的优缺点正好和受控组件相反。优点是性能好，更新不依赖父组件，从而避免触发兄弟组件更新。由于逻辑高内聚，对父组件传参依赖少，使用也更简单。

缺点

非受控组件的重置和关联更新比较困难、复杂，需要先卸载掉组件再重新初始化，一般使用 key 来解决

下面我们来看一个简单的表单分别受控与非受控实现的代码区别

Try it on Codesandbox

React Conf 2018 第一次介绍 Hooks 时的现场例子就是非受控组件用 key 重置状态

https://www.youtube.com/watch?v=wXLf18DsV-I

受控组件的特点是，**value/checked 和 onChange 成对出现**（也可以使用事件代理在父元素上冒泡处理所有 onChange 事件，但 React 对于受控组件的判断是 value/checked 和 onChange 成对出现，不然会在 development 模式下有个 warning）。受控组件需要在 onChange 的时候更改 state，来触发重绘。

非受控组件的特点是只有一个 defaultValue/defaultChecked 的属性。不需要更新记录 state，因此也不会造成重绘。实现功能的代码也更少。但对于复杂的联动需求（关联交验、联动更新）的场景，难以支持（或者说需要额外的技巧）。所以大多数没有性能瓶颈的情况下，**推荐大家使用受控组件开发**。

> 通过封装高阶组件可以简化受控组件的范式代码，对于表单类大量受控组件的场景，推荐使用成熟的表单解决方案，如：
>
> - formik 特点：易用，接口设计好，性能差
> - react-final-form 特点：较难用，性能好
> - react-hook-form 特点：更接近原生 html form 的实现，直接操作 DOM，非受控，性能最好

其他知识点

Server Side Render

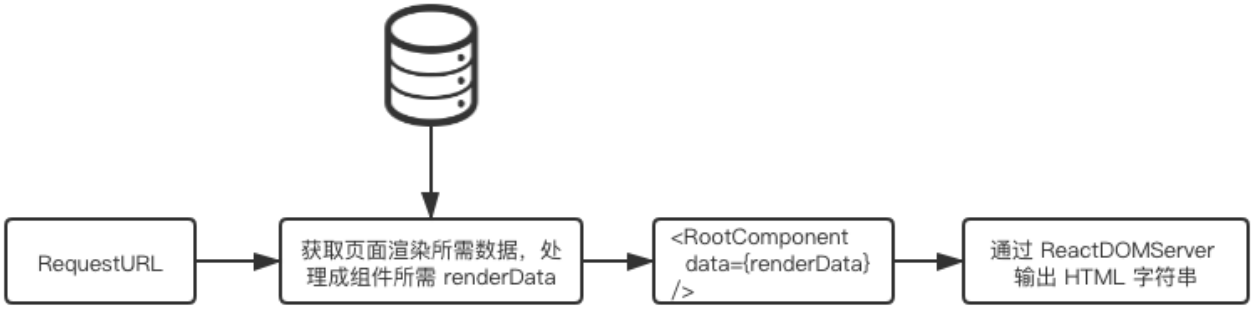
React 可以作为 UI 模板在服务端渲染

```
ReactDOMServer.renderToString(
  <h1>Hello World.</h1>
);
```

React 本身当字符串模板渲染 HTML 其实很简单。但由于 HTML 有很多除了 JS 之外的资源产物，问题就变得复杂起来。这里先不展开对于非 JS 类型资源及其他副作用的处理，从简单的函数理解 ServerSideRender 的原理。

对于每个请求地址（输入参数），服务端渲染（函数）应该是一定的页面呈现（输出）

React 的服务端渲染流程和原理上跟其他 jade、ejs 模板引擎的渲染没有任何区别：



服务端渲染的时候，useEffect / useLayoutEffect / useImperativeHandle 是不会执行的，所有 Component 都是同步调用，一次完成渲染。所以需要在渲染前准备好渲染所需的数据。

整个 React 服务端渲染流程跟其他服务端渲染技术一模一样，没有什么特别的。为了最大限度复用前端组件的逻辑，社区产生了各种服务端渲染的范式，通过一些约定、配置，来复用前端逻辑、简化服务端渲染的复杂度（主要是提前获取渲染所需数据的步骤和一些副作用，做同构设计。比如：为了复用处理 document.title 的值，react-helmet 有自身的服务端渲染范式，这样就不需要在服务端根据每个页面设置不同的 title 了；styled-components、jss 也都有各自的 css 服务端依赖收集范式；next.js 则同构增加约定的静态函数，来进行接口请求的同构）。

第四天：R18 Concurrent Features

R18 引入时间分片的方式执行渲染任务，通过 Lane 计算任务的过期时间给到 scheduler 库进行优先级排序，使得在渲染页面的同时，尽可能保持页面对于用户操作的响应。

Lane 简介

Lane 是 React 18 实现任务优先级排序的数据结构，一个 Lane 一共 31 条，具体为什么是 31 条，可以看 https://github.com/reactwg/react-18/discussions/27#discussioncomment-800354~~（看了还是不懂，求解答）~~（JS 位运算使用带符号 32位 格式，React 在计算 Lane 编号之后会将计算结果作为 index 使用，所以要去掉最高位符号位，保证正值），编号越小的任务越紧急。

React 每次重绘时都会挑一条高优的 Lane。执行中对于每个组件的 update，比较 update 生成时的 Lane 编号，相同 Lane 编号或 Lane 编号小（即，优先级高于渲染任务 Lane）的 update 会执行，编号大的会跳过。

下面通过 useState 这个 hook 上 Lane 的作用来更具体地说明一下

从一个计数器看 useState 如何运行

TypeScript

```
const [count, setCount] = useState(1);
...
<button
  type="button"
  onClick={
    () => { setCount(count + 1); }
  }
>+1</button>
...
```

setCount 生成 Update

button 每次点击执行 +1 操作。setCount 在执行的时候做了下面的事情：

1. 获取当前调用栈的 Lane 编号。Lane 的编号会由以下几种方式生成：
   a. 如果是事件触发的，比如本例中就是在 onClick 中触发的，那么 Lane 编号就是事件类型对应的 Lane 编号。事件分为离散事件（DiscreteEvent，如，click、touchstart、change 等）、连续事件（ContinuousEvent，如，mouse[enter|move|out]、touchmove、drag[enter|exit|leave|over] 等），离散类型的优先级非常高，意味着 React 会优先响应离散事件，而连续事件优先级低于离散事件

   在本例中，点击 button 会先触发 React 事件代理程序。代理程序会根据事件类型，将全局变量 currentLane 设置为离散事件对应的 Lane 编号，同时保存之前的 currentLane，在执行完用户注册的回调之后，恢复到之前的 Lane 编号。伪代码如下

TypeScript

```
function eventDelegator(eventName, userCallback, event) {
  const prevLane = currentLane;
  currentLane = getLaneByEventName(evetName); // 根据 event 类型获取 Lane 编号
  userCallback(event); // 执行用户注册的事件响应函数
  currentLane = prevLane; // 恢复之前调用栈的 Lane
}
```

   b. 如果是除了事件之外的异步系统触发的，使用默认的 Lane 编号

   比如，我们把 `setCount(count + 1);` 放到 setTimeout 中，此时，由于是异步调用 setCount，currentLane 一般是个空值（假设没有其他正在运行的渲染），这样就会获得一个默认的 Lane 编号，这个默认编号优先级低于事件触发的 Lane 编号

   常见的，如果我们在 useEffect 中获取数据，那么数据回调函数中的 setXXX 调用也是默认的 Lane 编号

c. 跟 a 类似的，startTransition API 也会像 eventDelegator 函数一样，把 currentLane 暂时赋值为某个 TransitionLane。这样，在 startTransition API 中调用的 setCount，获取到的 Lane 就是 TransitionLane。TransitionLane 优先级低于默认的 Lane 编号（即，优先级比 setTimeout 低）

d. 其他的 Lane 还有注水、错误处理、Suspense 处理，这些可以先不关注

2. 为 useState 所在的 fiberNode 节点生成更新链表，下面把更新链表称为 updateQueue。updateQueue 是一个成环的单链表结构，长下面这样

TypeScript

```typescript
interface UpdateQueue {
  lanes: number;     // 所有 Update lane 的合集
  baseState: State; // useState 当前状态
  next: Update;
}

interface Update {
  lane: number;    // 生成 Update 时的 Lane 上下文
  action: State | (currentState: State) => State; // setXXX 的传参
  next: Update; // 下一个 setXXX 注册 Update
}
```

在示例中，我们在 onClick 回调函数中同步执行 `setCount(count + 1)` 后，就会向 updateQueue 中添加如下的一个 Update

TypeScript

```typescript
const update = {
  lane: SyncLane,   // 因为 onClick 中同步调用，所以拿到的 currentLane 是 SyncLane
  action: 2,        // 第一次调用时 count + 1 等于 2
  next: null
}
```

如果调用的是 `setCount(s => s + 1)`，那么 update.action 就是 `s => s + 1`。

如果 setCount 是在 `mousemove` 的时候调用的，那么生成的 update.lane 就是 `InputContinuousLane`。

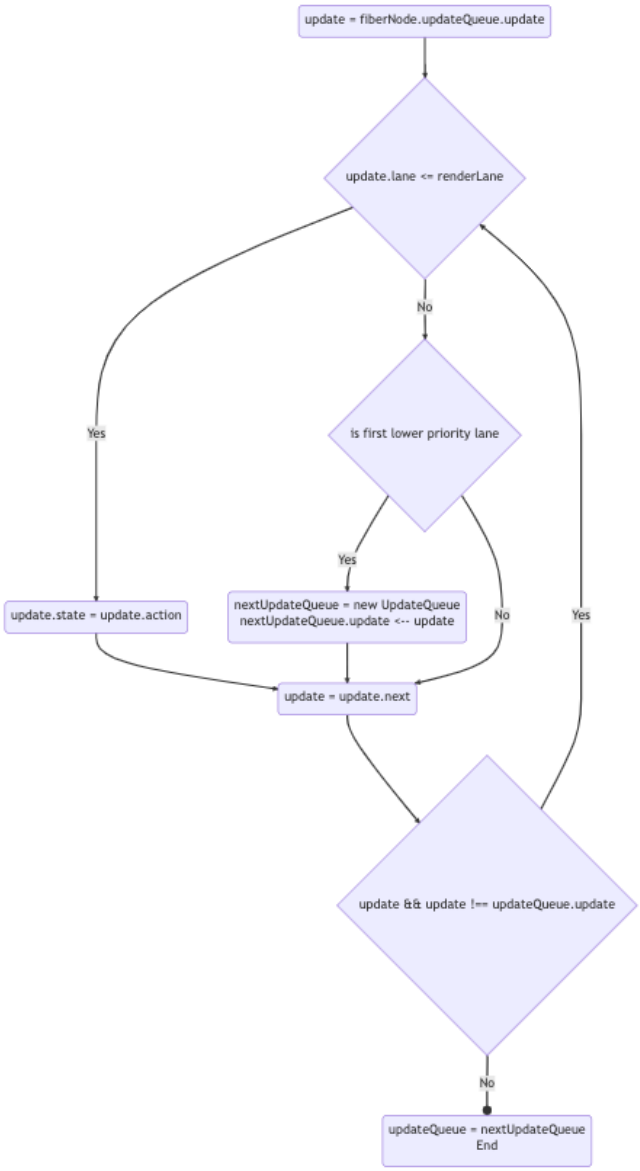
如果 setCount 是在 `startTransition` 中调用的，那么生成的 update.lane 就是 `TransitionLane[n]`。

3. 把当前 updateQueue 中的 Lane 冒泡到 FiberRootNode 记录，并注册重绘任务。FiberRootNode 对所有 Lane 计算出重绘任务的过期时间，进行排序，确定出下次重绘的 Lane 编号，向 scheduler 提交重绘任务

useState 消费 Update

React 重绘时，会再次执行到 useState，此时，会消费 useState hook 下的 updateQueue，计算出最新的 state，步骤如下：

1. 循环读取 updateQueue.next 链表，判断每个 update.lane 的优先级是否在本次重绘的 Lane 下，即 update.lane 编号小于等于（即优先级大于等于）重绘的 Lane。如果 TRUE，走 2；如果 FALSE，走 3
2. 读取 action，如果 action 是函数，将 updateQueue.baseState 作为参数传入，得到此 update 更新结果。回到步骤 1，读取下一个 Update
3. 如果当前 Update 是第一个优先级低于当前重绘的 Lane，走 4，跳过当前 update；如果当前 update.lane 不是第一个优先级低于本次重绘的 Lane，回到步骤 1
4. 生成一个新的 updateQueue，updateQueue.baseState 记录为最新的 state 状态，新 updateQueue 为当前 update 作为头的链表，即原 updateQueue 的子链表。回到步骤 1



下面通过几个例子加强一下知识点：

例一：同步连续调用

TypeScript

```
onClick={() => { setCount(3); setCount(s => s + 1); }}
```

上面的 onClick 执行后会生成如下 updateQueue

TypeScript

```
updateQueue = {
  lane: SyncLane,
  action: 3,
  next: {
    lane: SyncLane,
    action: s => s + 1,
    next: null
  }
}
```

在下次重绘调用 useState 时，上面的 updateQueue 就会被消费掉，以当前 count 为 baseState，逐个应用 updateQueue 链表上的 update

例二：其中几个异步调用

TypeScript

```
onClick={() => {
  setTimeout(() => {
    setCount(4);
    setCount(5);
  });
  setCount(3);
}}
```

上面的 onClick 执行后（假设 setTimeout 执行前没有重绘而消费掉 `setCount(3)` 调用生成的 update），会生成两个不同 Lane 的 update

TypeScript

```
updateQueue = {
  lane: SyncLane,
  action: 3,
  next: {
    lane: DefaultLane,
    action: 4,
    next: {
      lane: DefaultLane,
      action: 5,
      next: null
    }
  }
}
```

这时候触发的重绘任务有两个，一个 `Lane=SyncLane`，另一个 `Lane=DefaultLane`。因为 SyncLane 优先级高，所以先执行 SyncLane。

执行 SyncLane 时，根据上面 **useState 消费 Update** 介绍的步骤，会跳过 DefaultLane 的 update。useState 执行结束之后，会生成新的如下 updateQueue，保留 `setCount(4)` 和 `setCount(5)` 的 update。

TypeScript

```
updateQueue = {
  lane: DefaultLane,
  action: 4,
  next: {
    lane: DefaultLane,
    action: 5,
    next: null
  }
}
```

然后执行第二个 `Lane=DefaultLane` 的重绘任务。

看到这里，React 18 新的 Feature AutoBatch 是怎么实现的，相信你已经 get 到了。在 React 18 以前，这个例子中的代码会触发三次重绘，setTimeout、Promise 等各种异步回调中执行的 setState 操作都会触发一次独立的重绘。现在，**因为 Lane 架构的引入，React 可以标记出任务的优先级，挑选出相同优先级的 update 批量处理更新，这些 update 甚至无需在同一个 setTimeout、或 Promise 中**👋。

那么如果高低优先级 update 互相夹杂在一起，怎么保证最终的执行结果符合程序执行的顺序呢？让我们看下第三个例子

例三：不同优先级 Lane 的调用

要模拟优先级夹杂在一起，最简单的方法是使用 startTransition API。（其他方法可以根据 React 定义的 Lane 与事件类型的映射关系，用模拟事件触发事件回调去注册不同优先级 Lane 的重绘任务）

为了更好说明执行顺序对渲染结果的影响，下面用输入字符的例子作为用例说明，onClick 之后，通过 4个 setText 输入 1234：

TypeScript

```typescript
const [text, setText] = useState('');
//...
onClick={() => {
  setText('1');
  startTransition(() => {
    setText(s => s + '2');
  });
  setText(s => s + '3');
  startTransition(() => {
    setText(s => s + '4');
  });
}}
```

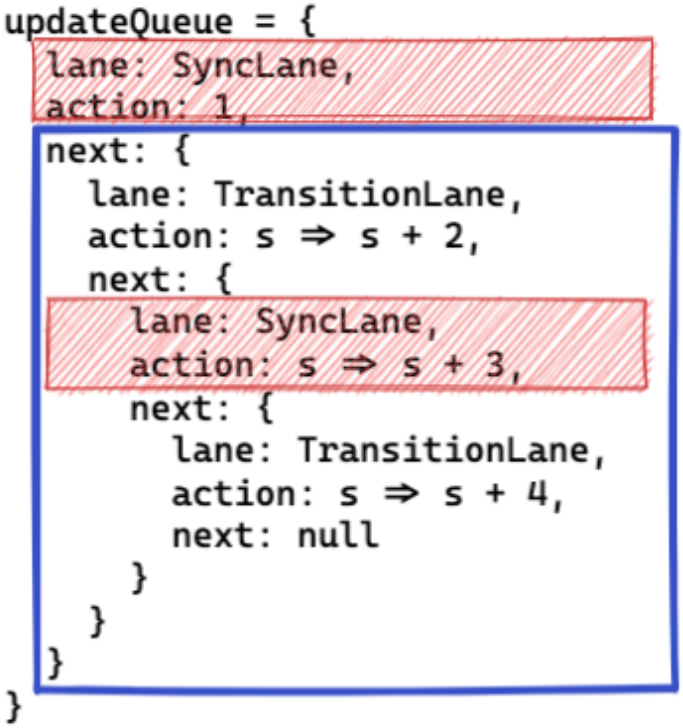
这段代码会生成如下 updateQueue

TypeScript

```typescript
updateQueue = {
  lane: SyncLane,   // <- 取 onClick 的 Lane
  action: 1,
  next: {
    lane: TransitionLane, // <- startTransition 会将 Lane 切换到 TransitionLane
    // TransitionLane 有很多条，会取其中一条可用的 Lane
    // 目前 Transition 相关的 Lane 合并策略都还没定型，处于初级阶段，
    // 可以先不关注怎么取哪条 Lane
    action: s => s + 2,
    next: {
      lane: SyncLane,  // startTransition 结束 Lane 回退到 onClick 的 Lane
      action: s => s + 3,
      next: {
        lane: TransitionLane,
        action: s => s + 4,
        next: null
      }
    }
  }
}
```

因为有 SyncLane 和 TransitionLane 两条 Lane，所以会注册两个重绘任务给 scheduler。

第一次渲染 SyncLane 时，当遍历到 Line 5 的 TransitionLane 低于当前渲染的 SyncLane 时，会将当前 update 开头的子串作为新的 updateQueue 存下来，并且将此新的 updateQueue 执行的 baseState 设置为前一个 update 执行结果，也就是 1



上图红色部分为第一次渲染执行的 update 部分，蓝色框为存下来的 updateQueue。如果之后有新的 setText 调用，会添加在蓝色部分后面。**第一次渲染结果为 '13'**。

第二次渲染 TransitionLane 时，从蓝色部分 `s => s + 2` 、`baseState=1` 开始执行。因为 SyncLane 优先级高于 TransitionLane，`s => s + 3` 会再次执行。**第二次渲染结果为 '1234'**。

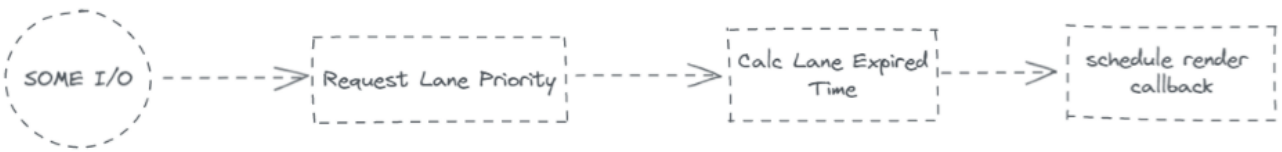
虽然 updateQueue 会因为 Lane 优先级的问题而跳过执行，导致中间渲染结果与预期不符，但因为跳过的子 queue 完整保留原 updateQueue 的执行顺序，所以最终结果可以保证一致性。

Try it on Codesandbox

Lane 作用总结

让我们来总结一下 Lane 的作用：

1. 根据特定场景触发的重绘，标记重绘任务的 Lane 优先级，处理各种 Lane 的合并、混合，将 Lane 转化为重绘任务的超时时间，注册重绘任务到 scheduler。如下图流程：



2. 渲染时，根据 update 的 Lane 标记，筛选高优 update 执行。

关于其他生命周期及 Hooks API 的处理，其实跟 useState 处理流程基本相通。区别只是应用 updateQueue 更新的时机和 FiberNode 遍历顺序不同而已。比如 useEffect 生成的 updateQueue，会在 React 重绘的 commit 阶段执行，然后 FiberNode 遍历是深度优先遍历，自下而上遍历 FiberNode，执行注册的 useEffect hook。

Breaking Change in concurrent render

使用 R18 Concurrent render 时要注意，这个 feature 是一个 breaking change。

Breaking 场景一：

> 下面例子中的 count 之所以用 object 是为了避免引入 fast bailout 导致更奇怪的现象，所以用 object 装箱使得每次 state 的值都更新。感兴趣的可以把 count 改成 number 类型试一下。

Try it on Codesandbox

TypeScript

```typescript
import { useEffect, useState } from "react";
import "./styles.css";

export default function App(props: { isSync?: boolean }) {
  const { isSync } = props;
  const log = useState(() => (...args: any[]) => {
    console.log(`[${isSync ? "sync" : "concurrent"}]: `, ...args);
  })[0];
  const [count, setCount] = useState({ value: 0 });

  useEffect(() => {
    log("count is: " + count.value);
  });

  return (
    <div className="App">
      <h1>Count is {count.value}</h1>
      <button
        type="button"
        onClick={() => {
          Promise.resolve().then(async () => {
            let tmp = 0;
            setCount((s) => {
              tmp = s.value;
              return { value: tmp + 1 };
            });
            setCount({ value: tmp + 2 });
          });
        }}
      >
        +2
      </button>
    </div>
  );
}
```

之前介绍了 updateQueue 的运行和数据结构，这里用 updateQueue 就很好理解为什么 concurrent 模式下为什么点击 +2 button 一直是 2 了：

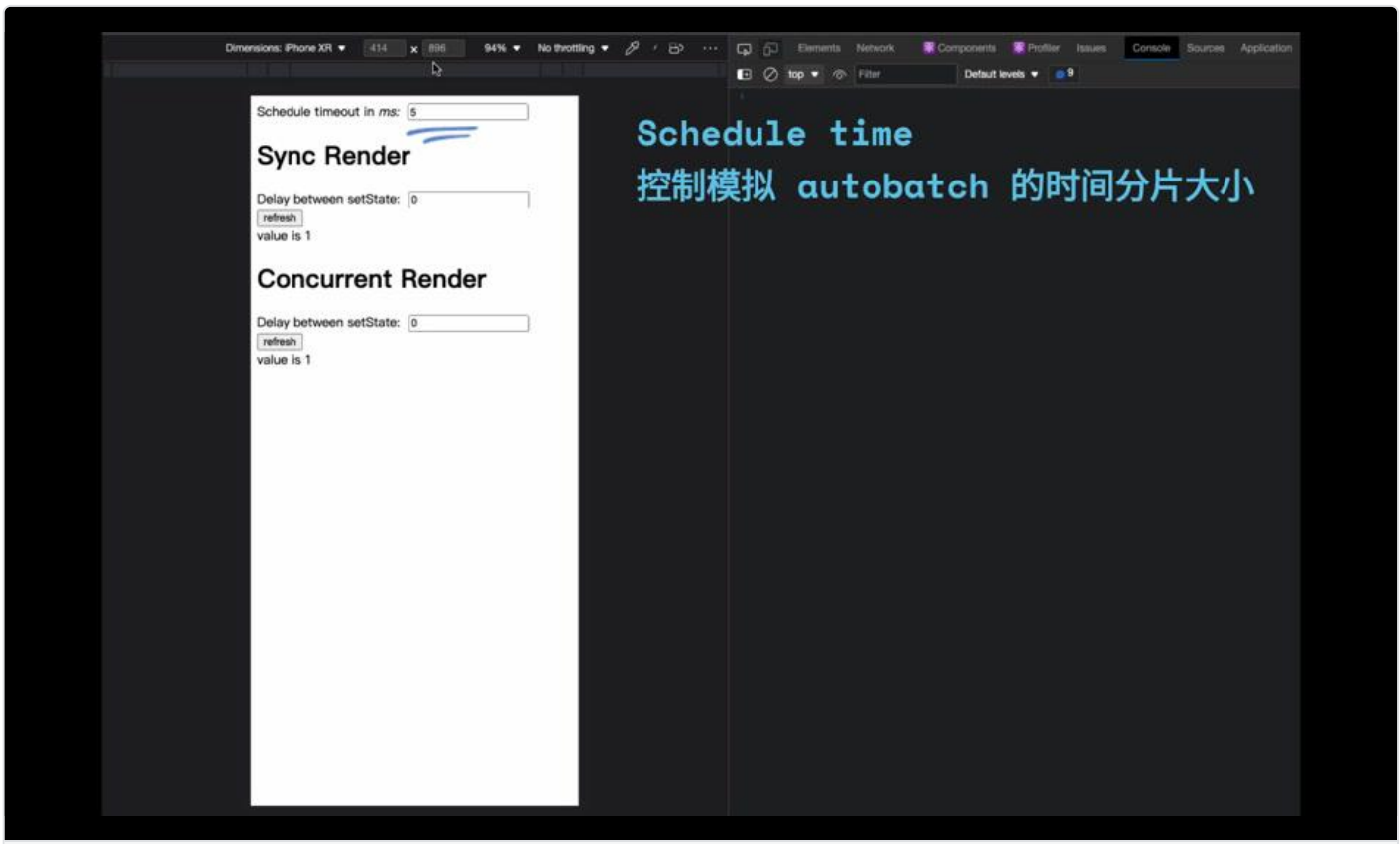
TypeScript

```
updateQueue = {
  action: (s) => {
    tmp = s.value;
    return { value: tmp + 1 };
  },
  next: {
    action: { value: 2 };
  }
}
```

当执行重绘时，因为最后一个 action 永远是 `{ value: 2 }`，第一个 update.action 是再下一次重绘时候才执行的，所以在生成第二个 update 时（`setCount({value: tmp + 2})`），tmp 没有赋值。（前面提到的 bailout 逻辑会导致 lanes 重置为 NoLanes，从而触发 eager state 计算，直接运行 update，在这种情况下，会先执行第一个 update 中的 action，tmp 会被先赋值。有兴趣可以 debug 一下）。

所以在 concurrent 模式下，使用 setState 去拿最新 state 状态变得不太可行。需要使用 flushSync API 进行兼容。

Breaking 场景二：



r18-auto-batch-render-issue2.mp4

Try in on Codesandbox

那么具体 React 的 auto batch render 的合并逻辑是什么呢？这个例子使用 `setImmediate` 对 schedule 进行了一些 hack，来使我们能控制时间分片大小。

```
<script type="text/javascript">
  let inputDOM;
  window.setImmediate = (cb) => {
    if (!inputDOM) {
      inputDOM = document.getElementById("time-slice");
    }
    setTimeout(cb, +inputDOM.value);
  };
</script>
```

当我们两次 setState 调用的间隔小于 schedule 时间分片时，渲染就会合并。在 chrome 中，异步回调触发顺序是 ResolvedPromise > MessageChannel > setTimeout(fn, 0)，因此，在 chrome 下，插入 setTimeout(fn,0) 也是可以安全的把渲染给切分开的。

但切分不开已经到达 Resolved 状态的 Promise：

TypeScript

```
.then(async() => {
  setState(1);
  await cachedPromise; // not work
  setState(2);
});
```

在实际生产中，这类连续调用的 setState 被合并会导致很多 useEffect 漏执行：

1. 埋点触发
2. 数据刷新

如果使用的是老的浏览器，比如降级到了 setTimeout 的话，那么被合并的概率还要大，也更容易丢失 useEffect 的触发。

scheduler 简介

React 注册任务到 scheduler 时，会告诉它任务的优先级和开始时间，scheduler 通过这两个因素算出任务的超时时间进行最小堆优先级排序。scheduler 以每 5ms 为时间分片执行最小堆上的任务。如果超过 5ms 堆内还有任务，通过注册异步回调的方式，让出 CPU 资源，等下一次 JS 线程拿到 CPU 资源时继续执行任务。

scheduler 如何让出 CPU 资源

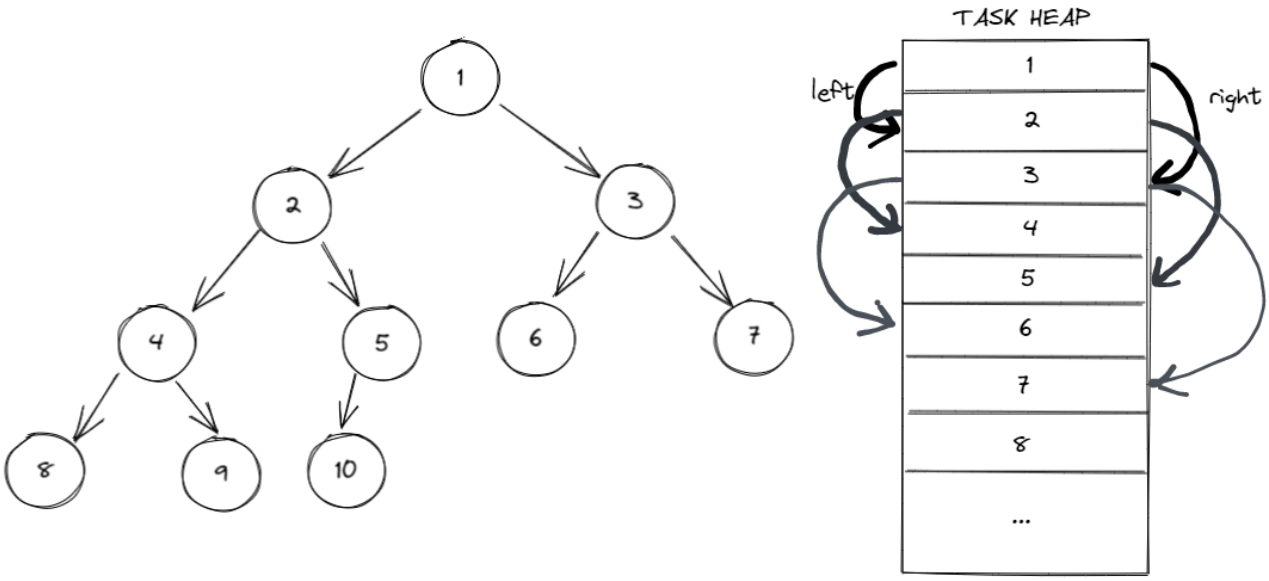
Node.js 或者 IE 下，调用 setImmediate，让出 CPU 资源。

Chromium 87 及以上，调用 MessageChannel。

如果上面两个 API 都不支持，回退到 `setTimeout(xxx, 0)`。

执行任务队列（taskQueue）与延迟任务队列（timerQueue）

scheduler 维护两个队列，正在执行的任务放在 taskQueue 中，还没到执行时间的放在 timerQueue 中。两个队列通过最小堆（根节点是整棵树最小的值）的数据结构，对注册的任务进行优先级排序。



taskQueue 与 timerQueue 的排序参照（sortIndex）有所不同：

- taskQueue 参照 expirationTime 进行排序，根节点为最早过期的任务

  expirationTime = startTime + priorityTimeout

  - startTime - 任务开始时间
  - priorityTimeout - 根据注册任务的优先级枚举，映射到不同的 timeout 时间：
    - ImmediatePriority - -1ms
    - UserBlockingPriority - 250ms
    - NormalPriority - 5_000ms
    - LowPriority - 10_000ms
    - IdlePriority - 2^30-1ms
- timerQueue 参照任务开始时间 startTime 进行排序，根节点为最早要开始执行的任务

  startTime = *now* + *delay*

  React 18 正式版本中，startTransition API 把 delay 传参去掉了，所以不会有任务加到 timerQueue 中

任务队列的消费

TypeScript

```typescript
// 简化的任务队列执行代码，由于 timerQueue 没用到，
// 省略 timerQueue task 转换及计时器注册相关逻辑
function workLoop() {
  while(task = taskQueue[0] && !shouldYield()) {
    task.callback();
    minHeapPop(taskQueue);
  }
  if (task) {
    messageChannel.port2.send(null);
  }
}
messageChannel.port1.onmessage = workLoop();
```

代码逻辑概述：

1. L4: 取 taskQueue 第一个任务，判断时间分片是否到期（`shouldYield()`）
2. L5: 执行任务
3. L6: 最小堆删除堆顶并重新排序
4. L8-9: 判断 taskQueue 中是否还有任务，有的话通过 messageChannel 触发延迟执行，让出 CPU 资源
5. L12: 注册 messageChannel 回调，作为浏览器渲染线程执行完的回调

API 介绍

Concurrent Features 主要涉及一下四个 API：

- react-dom/client 中的 createRoot
- useDeferredValue
- useTransition
- startTransition

接下来，让我们看一下这四个 API 如何配合工作，实现 Time Slice 下的运行。

createRoot(): FiberRootNode

要使用 Concurrent Features，首先要把 ReactDOM.render 换成 ReactDOMClient.createRoot 的调用方式：

TypeScript

```typescript
import { createRoot } from 'react-dom/client';
import { MyApp } from './MyApp';

const root = createRoot(document.querySelector('#root')!);

root.render(<MyApp />);
```

FiberRootNode 作为 Fiber 的新类型，有许多字段存放 Concurrent Features 功能所需的数据：

- Lane 通道
- Lane 通道过期时间
- 中断的 WorkInProgressFiber
- Lane 通道任务的开始时间
- ...

startTransition & useTransition

这两个 API 相似但又有不同之处。不当的使用会引起性能劣化。两者的区别主要是以下两点：

1. useTransition 只能在 FunctionComponent 中使用
2. useTransition 比 startTransition 多一次 SyncLane 优先级的重绘任务，以渲染 pending 状态

让我们先看一个例子，感受一下 useTransition 与 startTransition 的区别，再来说明是什么导致的

例一：startTransition 降低渲染优先级

使用 startTransition 前：

slide-normal.mov

使用 startTransition 后：

slide-with-transition.mov

对比可知：

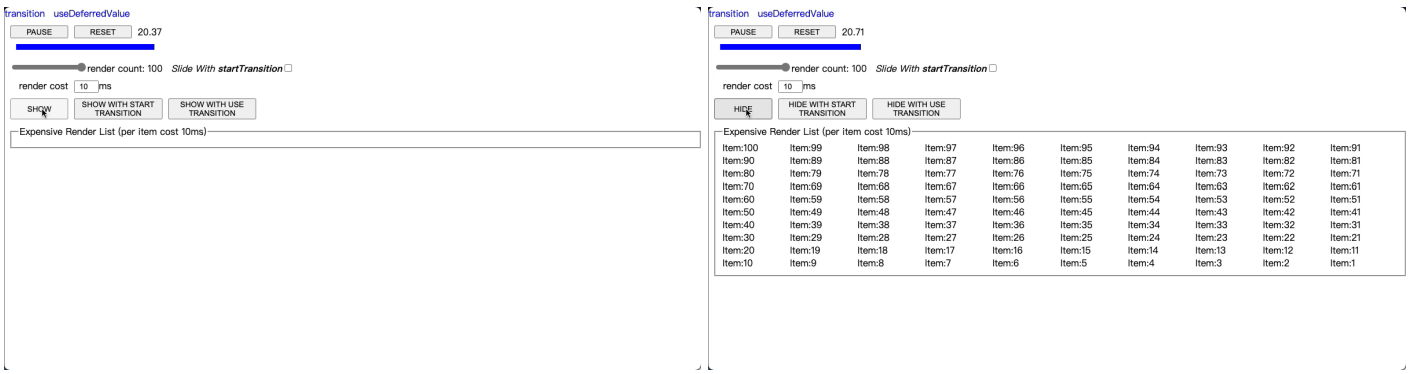
1. startTransition 可以减少重绘列表（Expensive Render List）的次数

   在 slider 滑动时，优先响应 slider 组件的状态变更。当 slider 不变更或变更间隔足够长时，才会重绘

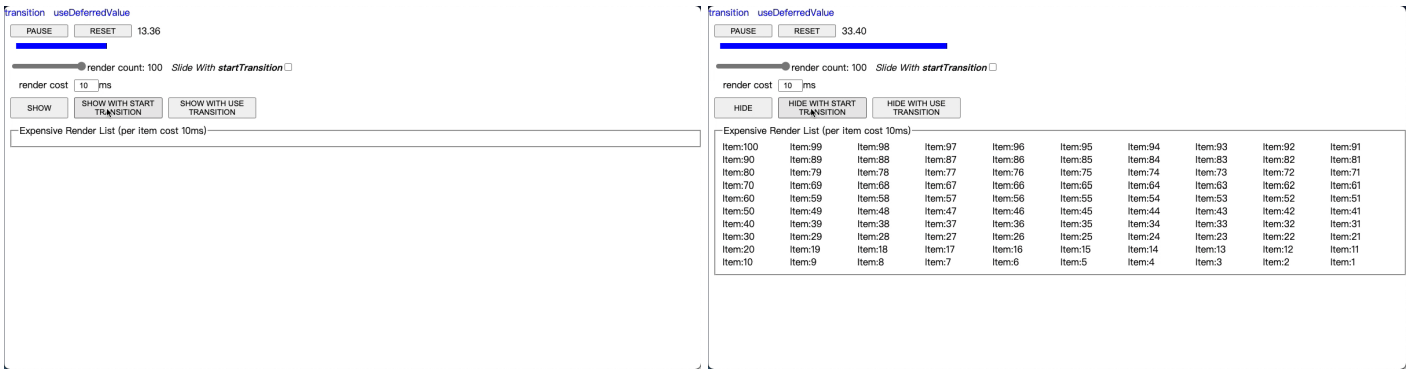
2. startTransition 触发的重绘，虽然页面也会卡住（蓝色时间条为 JS 动画，当卡住时说明 JS 线程阻塞），但渲染完成后继续的时间条会根据循环次数往前有跳跃现象。而 onChange 触发的重绘因为是 SyncLane 优先级，所以会卡死时间条，恢复时，时间条为卡死的位置

下面我调大 render cost 5ms => 10ms，来截两帧恢复前后的对比图来比较一下：

- 下图为 SyncLane 重绘前后两帧的对比图，0.34% * 5s = 17ms，正好是 rAF 间隔时间，说明 rAF 被 block 😢



- 下图为 startTransition 重绘前后两帧的对比图，相差正好是 20% * 5s = 100个 item * 10ms 渲染时间，说明 rAF 正常执行 😀



完整视频：

timer.mov

例二：从组件销毁，看 useTransition 和 startTransition 的区别

先看视频：

useTransition-startTransition.mov

中间按钮是 startTransition，右边按钮是 useTransition，我们可以明显感受到，在销毁耗时渲染组件时，startTransition 是不执行销毁组件的渲染的。

而 useTransition 在挂载和渲染时都会执行组件的渲染。并且，销毁时，useTransition 触发的首次渲染，还出现了页面卡死的现象（disabled 效果没有第一时间渲染出来，这个行为不符合预期，在组件挂载时，能正常渲染 disabled 效果）。

这就是因为 useTransition 需要先更新 pending 状态，这个更新的 Lane 优先级是 SyncLane，而在卸载的时候，由于 Expensive Render List 处于满载状态，所以卡死。而在组件挂载时，更新 pending 的那次重绘 Expensive Render List 是空载状态，所以能很快响应 disabled 效果。

Try it on Codesandbox

startTransition 还是 useTransition 的最佳实践

1. 如果不需要展示 pending 状态，选 startTransition
2. 如果逾期 pending 状态很短，比如业务可以定一个交互延迟阈值时间，低于这个阈值，选 startTransition
3. 如果强依赖 pending 状态（比如，例子中的 disabled 某些操作），记得给耗时渲染组件添加 memo

   可以在 https://codesandbox.io/s/deference-between-transition-4y6kzu?file=/src/TransitionDemo.tsx:425-457 中，`USE_MEMO_ELEMENTS` 设置为 true 后点击 *HIDE WITH USE TRANSITION 按钮*看效果。无论是用 React.memo 还是 useMemo，只要能把耗时 render 的组件、函数在不需要重绘时，不执行，就能解决 pending 状态带来的性能劣化问题
4. useTransition、startTransition 所在组件要与优先响应用户交互的组件保持解耦，比如例子中的 Slider 组件需要自己维护 state，而不能将 state 提升到 useTransition、startTransition 所在组件

   因为 useTransition、startTransition 重绘包含有耗时渲染组件，如果状态提升到同级别组件，会导致重绘必定包含耗时渲染组件，从而无法达到即时响应用户交互的目的

[Later] useDeferredValue

> React 18 与 React 18.1 实现有变，暂时先不写这个

附录扩展

相关文档链接

- React
- ReactRouter
- Redux
- StyledComponents
- ByteDesign
- AntDesign

自定义 React Renderer

https://www.youtube.com/watch?v=CGpMlWVcHok