React Avançado
Análises aprofundadas, investigações, padrões de desempenho e técnicas
por Nadia Makarevich
Ilustrações e design da capa do livro: Nadia Makarevich
Revisão técnica: Andrew Grischenko

Índice

Introdução: como ler este livro

O React é um dos frameworks front-end mais populares disponíveis. Não há como negar isso. Como resultado, a internet está repleta de cursos, livros e blogs sobre o React. Além disso, a documentação mais recente é muito boa. Então, qual é o propósito deste livro? Ele pode realmente preencher alguma lacuna?

Acredito que sim. Tenho escrito um blog dedicado a padrões avançados para desenvolvedores do React (https://www.developerway.com) há algum tempo. Um dos feedbacks mais consistentes que recebo é que há falta de conteúdo de nível avançado.

A documentação é muito boa para começar com o React. Existem milhões de livros, cursos e blogs voltados para iniciantes. Mas o que fazer depois de começar com sucesso? Para onde ir se você quiser entender como as coisas funcionam em um nível mais profundo? O que ler se você já está usando o React há algum tempo e cursos para iniciantes ou até mesmo de nível intermediário não são suficientes?

Existem poucos recursos disponíveis para isso. Essa é a lacuna que este livro pretende preencher.

O livro assume um certo conhecimento básico do React. Ele espera que você compreenda o que é o estado e como usá-lo. O que são os hooks e como escrevê-los. O que é um componente e como renderizar componentes dentro de outros componentes. Em outras palavras, ele assume que você pode escrever com sucesso uma aplicação "todo" básica.

O que ele pretende fornecer é o conhecimento que permite que você avance da "aplicação básica" para "um guru do React na minha equipe". Ele começa imediatamente investigando e corrigindo um bug de desempenho. Ele aprofunda-se no que

explica como funcionam as re-renderizações e como elas afetam o desempenho. Apresenta o funcionamento do algoritmo de reconciliação, como lidar com closures no React, vários padrões de composição que podem substituir a memoização, como a memoização funciona, como implementar o debouncing corretamente, e muito mais.

O livro é estruturado em capítulos. Cada capítulo é uma história, investigação ou análise aprofundada focada em um único tópico. No entanto, eles não são completamente isolados: cada capítulo se baseia no conhecimento apresentado nos capítulos anteriores. Além disso, cada capítulo tenta apresentar apenas o conhecimento necessário para entender o conceito discutido, sem mais. Para facilitar a leitura e o foco. Por exemplo, não faz sentido começar com a introdução do algoritmo de reconciliação para entender o que é uma re-renderização e como ela afeta o desempenho.

Portanto, recomendo que você leia o livro na ordem dos capítulos. Se você já possui conhecimento além do aplicativo "todo" simples, é muito provável que já conheça muitos dos conceitos. Neste caso, cada capítulo tem uma lista de tópicos que você pode esperar aprender, além de uma seção "Principais aprendizados", com um resumo conciso dos tópicos apresentados. Apenas espreitar esses primeiros tópicos já lhe dará uma boa ideia do que está dentro.

Por fim, o livro não diferencia entre React, react-dom e o transpiler JSX. Tecnicamente, é a biblioteca react-dom que é responsável por coisas como Portals, e transpilers como o Babel convertem a sintaxe JSX em funções. No entanto, para entender o conteúdo do livro, essa distinção realmente não importa. Usamos todas as três sem nem perceber ao escrever interfaces para a web. Portanto, por simplicidade, tudo é referido como "React".

Espero que esteja pronto para começar agora! Vamos investigar um problema de desempenho, aprender o grande mito das re-renderizações e explorar o padrão de otimização de desempenho mais simples, mas muito poderoso, com composição.
Pronto? Vamos começar!

Vamos direto ao ponto, não é? E vamos falar sobre desempenho logo de cara:
é um dos tópicos mais importantes atualmente, quando se trata de
desenvolver aplicações, e, como resultado, é um tema central deste
livro.

E quando se trata de React e desempenho no React, é crucial entender as re-renderizações e sua influência. Como elas são acionadas, como se propagam pela aplicação, o que acontece quando um componente realiza uma re-renderização e por que, e por que precisamos delas em primeiro lugar.

Este capítulo apresenta estes conceitos, que serão explorados em mais detalhes nos próximos capítulos. E para tornar tudo mais interessante, vamos abordá-lo na forma de uma investigação. Vamos apresentar um problema comum de desempenho em uma aplicação, analisar o que acontece por causa disso, e como corrigi-lo com uma técnica de composição muito simples. Ao fazer isso, você aprenderá:

O que é uma re-renderização, e por que precisamos dela.
Qual é a fonte original de todas as re-renderizações.
Como o React propaga as re-renderizações pela aplicação.
O grande mito das re-renderizações e por que a alteração de propriedades não importa.
A técnica de "mover o estado para baixo" para melhorar o desempenho.
Por que os hooks podem ser perigosos quando se trata de re-renderizações.

O problema

Imagine-se como um desenvolvedor que herdou uma aplicação grande, complexa e muito sensível ao desempenho. Muitas coisas estão acontecendo lá,

Muitas pessoas trabalharam nisso ao longo dos anos, e milhões de clientes estão usando-a atualmente. Como sua primeira tarefa no emprego, você é solicitado a adicionar um botão simples que abre um diálogo modal logo acima deste aplicativo.

Você analisa o código e encontra o local onde o diálogo deve ser acionado:
```javascript
const App = () => {
  // muito código aqui
  return (
    <div className="layout">
      {/* o botão deve ir em algum lugar aqui */}
      <VerySlowComponent />
      <BunchOfStuff />
      <OtherStuffAlsoComplicated />
    </div>
  );
};
```
Então você implementa. A tarefa parece trivial. Nós já fizemos isso centenas de vezes:

```javascript
const App = () => {
  // adicione um estado
  const [isOpen, setIsOpen] = useState(false);

  return (
    <div className="layout">
      {/* adicione o botão */}
      <Button onClick={() => setIsOpen(true)}>
        Abrir diálogo
      </Button>
      {/* adicione o próprio diálogo */}
      {isOpen ? (
        <ModalDialog onClose={() => setIsOpen(false)} />
      ) : null}
      <VerySlowComponent />
```

```
      <BunchOfStuff />
      <OtherStuffAlsoComplicated />
    </div>
  );
};
```

Basta adicionar um estado que armazena se o diálogo está aberto ou fechado. Adicione o botão que dispara a atualização do estado ao clicar. E o próprio diálogo que é renderizado se a variável de estado for verdadeira.

Você inicia o aplicativo, tenta e - oh! - leva quase um segundo para abrir esse diálogo simples!

Exemplo interativo e código completo
https://advanced-react.com/examples/01/01
Pessoas com experiência em lidar com o desempenho do React podem estar tentadas a dizer algo como: "Ah, claro! Você está re-renderizando toda a aplicação ali, você só precisa envolver tudo em React.memo e usar os hooks useCallback para evitar isso." E tecnicamente isso é verdade. Mas não se apresse. A memorização é completamente desnecessária aqui e causará mais danos do que benefícios. Existe uma maneira mais eficiente.
Mas primeiro, vamos revisar exatamente o que está acontecendo aqui e por quê.
Atualização de estado, componentes aninhados e re-renderizações
Vamos começar do começo: a vida do nosso componente e as etapas mais importantes que precisamos considerar quando falamos sobre desempenho. Essas são: montagem, desmontagem e re-renderização.

Quando um componente aparece pela primeira vez na tela, chamamos isso de montagem. Nesse momento, o React cria uma instância desse componente pela primeira vez, inicializa seu estado, executa seus hooks e adiciona elementos ao DOM. O resultado final: vemos o que renderizamos neste componente na tela.

Em seguida, temos a desmontagem: quando o React detecta que um componente não é mais necessário. Assim, ele realiza a limpeza final, destrói a instância desse componente e tudo o que está associado a ele, como o estado do componente, e finalmente remove o elemento do DOM associado a ele.

E, finalmente, a re-renderização. Quando o React atualiza um componente já existente com novas informações. Comparado com a montagem, a re-renderização é leve: o React simplesmente reutiliza a instância existente, executa os hooks, realiza todos os cálculos necessários e atualiza o elemento do DOM existente com os novos atributos.

Cada re-renderização começa com o estado. No React, sempre que usamos um hook como useState, useReducer ou qualquer uma das bibliotecas externas de gerenciamento de estado como Redux, adicionamos interatividade a um componente.

A partir de agora, um componente terá um pedaço de dados que será preservado ao longo de seu ciclo de vida. Se algo acontecer que exija uma resposta interativa, como um usuário clicar em um botão ou algum dados externos chegarem, atualizamos o estado com os novos dados.

A re-renderização é uma das coisas mais importantes a entender no React. É quando o React atualiza o componente com os novos dados e ativa todos os hooks que dependem desses dados. Sem eles, não haverá atualizações de dados no React e, como resultado, não haverá interatividade. O aplicativo será completamente estático. E a atualização do estado é a fonte inicial de todas as re-renderizações em aplicativos React. Se tomarmos nosso aplicativo inicial como exemplo:

```
const App = () => {
  const [isOpen, setIsOpen] = useState(false);
```

```
  return (
    <Button onClick={() => setIsOpen(true)}>
      Abrir diálogo
    </Button>
  );
};
```

Quando clicamos no botão, ativamos a função `setIsOpen`, que atualiza o estado `isOpen` de `false` para `true`. Como resultado, o componente `App` que mantém esse estado é renderizado novamente.

Após o estado ser atualizado e o componente `App` ser renderizado novamente, os novos dados precisam ser entregues a outros componentes que dependem deles. O React faz isso automaticamente para nós: ele identifica todos os componentes que o componente inicial renderiza, renderiza-os novamente, e então renderiza os componentes aninhados dentro deles, e assim por diante, até chegar ao final da cadeia de componentes.

Se você imaginar um aplicativo React típico como uma árvore, tudo a partir do ponto onde a atualização do estado foi iniciada será renderizado novamente.

No caso do nosso aplicativo, tudo que ele renderiza, todos aqueles componentes muito lentos, serão renderizados novamente quando o estado for alterado:

```javascript
const App = () => {
  const [isOpen, setIsOpen] = useState(false);

  // tudo o que é retornado aqui será renderizado novamente quando o
  // estado for atualizado
  return (
    <div className="layout">
      <Button onClick={() => setIsOpen(true)}>
        Abrir diálogo
      </Button>
      {isOpen ? (
        <ModalDialog onClose={() => setIsOpen(false)} />
      ) : null}
      <VerySlowComponent />
      <BunchOfStuff />
      <OtherStuffAlsoComplicated />
    </div>
  );
};
```

```
  );
};
```

Como resultado, leva quase um segundo para abrir o diálogo - o React precisa re-renderizar tudo antes que o diálogo apareça na tela.

O importante a se lembrar aqui é que o React nunca vai "para cima" na árvore de renderização quando re-renderiza componentes. Se uma atualização de estado originou-se em algum lugar no meio da árvore de componentes, apenas os componentes "abaixo" da árvore serão re-renderizados.

A única maneira para que os componentes na "base" afetem os componentes no "topo" da hierarquia é que eles chamem explicitamente a atualização de estado nos componentes "superiores" ou passem os componentes como funções.

O grande mito das re-renderizações

Você notou que eu não mencionei nada sobre "props" aqui?

Você pode ter ouvido essa afirmação: "Re-renderização de componentes quando as suas propriedades mudam". É um dos equívocos mais comuns em React: todos acreditam nisso, ninguém duvida, e simplesmente não é verdade.

O comportamento normal do React é que, se uma atualização de estado for acionada, o React re-renderizará todos os componentes aninhados, independentemente das suas propriedades. E se uma atualização de estado não for acionada, então alterar as propriedades será apenas "ignorado": O React não as monitora.

Se eu tiver um componente com propriedades, e eu tentar alterar essas propriedades sem acionar uma atualização de estado, algo como este:

```javascript
const App = () => {
  // a variável local não funcionará
  let isOpen = false;

  return (
    <div className="layout">
      {/* nada acontecerá */}
      <Button onClick={() => (isOpen = true)}>
        Abrir diálogo
      </Button>
      {/* nunca aparecerá */}
      {isOpen ? (
        <ModalDialog onClose={() => (isOpen = false)} />
      ) : null}
    </div>
  );
};
```

Simplesmente não funcionará. Quando o botão é clicado, a variável `isOpen` será alterada. Mas o ciclo de vida do React não é acionado, portanto, a saída de renderização nunca é atualizada, e o `ModalDialog` nunca aparecerá.

Exemplo interativo e código completo
https://advanced-react.com/examples/01/02

No contexto de re-renderizações, se as propriedades mudaram ou não em um componente, isso só importa em um caso específico: se o referido componente for envolvido no componente de ordem superior React.memo. Nesse caso, e apenas nesse caso, o React parará sua cadeia natural de re-renderizações e primeiro verificará as propriedades. Se nenhuma das propriedades mudar, então as re-renderizações pararão. Se mesmo uma única propriedade mudar, elas continuarão normalmente.

Impedir re-renderizações com memoização de forma adequada é um tópico complexo com várias ressalvas. Leia mais detalhes sobre isso no Capítulo 5.
Memoização com useMemo, useCallback e React.memo.
Mover o estado para baixo.

Agora que ficou claro como o React redesenha os componentes, é hora de aplicar esse conhecimento para resolver o problema original. Vamos analisar o código com mais detalhes, especialmente onde utilizamos o estado do diálogo modal:

```
const App = () => {
  // nosso estado é declarado aqui
  const [isOpen, setIsOpen] = useState(false);

  return (
    <div className="layout">
      {/* o estado é usado aqui */}
      <Button onClick={() => setIsOpen(true)}>
        Abrir diálogo
      </Button>
      {/* o estado é usado aqui */}
      {isOpen ? (
        <ModalDialog onClose={() => setIsOpen(false)} />
      ) : null}
      <VerySlowComponent />
      <BunchOfStuff />
      <OtherStuffAlsoComplicated />
    </div>
  );
};
```

Como você pode ver, ele é relativamente isolado: utilizamos apenas no componente Button e no próprio ModalDialog. O restante do código, todos os componentes muito lentos, não dependem dele e, portanto, não precisam ser redesenhados quando esse estado muda. É um exemplo clássico do que se chama um redesenho desnecessário.

Envolver esses componentes com React.memo evitará que eles se redesenhem neste caso, isso é verdade. Mas React.memo tem muitas ressalvas e complexidades em torno dele (veja mais no Capítulo 5. Memoização com useMemo, useCallback e React.memo). Existe uma melhor maneira. Tudo o que precisamos fazer é extrair os componentes que dependem desse estado e do próprio estado para um componente menor:

```
const ButtonWithModalDialog = () => {
  const [isOpen, setIsOpen] = useState(false);

  // Render apenas o Botão e o ModalDialog aqui
  return (
    <>
      <Button onClick={() => setIsOpen(true)}>
        Abrir diálogo
      </Button>
      {isOpen ? (
        <ModalDialog onClose={() => setIsOpen(false)} />
      ) : null}
    </>
  );
};
```

E, em seguida, renderize este novo componente no App original:

```
const App = () => {
  return (
    <div className="layout">
      {/* Aqui, o componente com o estado é incluído */}
      <ButtonWithModalDialog />
      <VerySlowComponent />
      <BunchOfStuff />
      <OtherStuffAlsoComplicated />
    </div
  );
};
```

Exemplo interativo e código completo
https://advanced-react.com/examples/01/03

Agora, a atualização do estado quando o Botão é clicado ainda é disparada, e
alguns componentes são renderizados novamente devido a isso. Mas! Isso só acontece com

componentes dentro do componente ButtonWithModalDialog. E
é apenas um botão pequeno e o diálogo que deveria ser exibido de qualquer forma.
O restante do aplicativo está seguro.
Em essência, criamos um novo sub-ramo dentro da nossa árvore de renderização e
movemos nosso estado para ele.
Como resultado, o diálogo modal aparece instantaneamente. Nós simplesmente corrigimos um grande problema de desempenho com uma técnica simples!
O perigo dos hooks personalizados
Outro conceito muito importante que não devemos esquecer ao lidar
com o estado, re-renderizações e desempenho são os hooks personalizados. Afinal, eles
foram introduzidos exatamente para que pudéssemos abstrair a lógica baseada em estado. É
muito comum ver a lógica como a que temos acima extraída para algo como o hook useModalDialog.
Uma versão simplificada poderia ser:

```
const useModalDialog = () => {
  const [isOpen, setIsOpen] = useState(false);

  return {
    isOpen,
    open: () => setIsOpen(true),
    close: () => setIsOpen(false),
```

```
  };
};
```

E então use este hook em nosso App em vez de definir o estado diretamente:

```
const App = () => {
  // o estado está agora no hook
  const { isOpen, open, close } = useModalDialog();

  return (
    <div className="layout">
      {/* basta usar o método "open" do hook */}
      <Button onClick={open}>Abrir diálogo</Button>
      {/* basta usar o método "close" do hook */}
      {isOpen ? <ModalDialog onClose={close} /> : null}
      <VerySlowComponent />
      <BunchOfStuff />
      <OtherStuffAlsoComplicated />
    </div>
  );
};
```

Por que eu chamei isso de "o perigo"? Parece um padrão razoável, e o código é um pouco mais limpo. Porque o hook esconde o fato de que temos um estado no App. Mas o estado ainda está lá! Sempre que ele mudar, ainda irá disparar uma re-renderização do componente que usa este hook. Não importa se este estado é usado diretamente no App ou mesmo se o hook retorna algo.

Exemplo interativo e código completo
https://advanced-react.com/examples/01/04

Por exemplo, se eu quiser ser mais elaborado com o posicionamento deste diálogo e introduzir algum estado dentro deste hook que ouve o redimensionamento da janela:

```
const useModalDialog = () => {
  const [width, setWidth] = useState(0);

  useEffect(() => {
    const listener = () => {
      setWidth(window.innerWidth);
    }

    window.addEventListener('resize', listener);

    return () => window.removeEventListener('resize', listener);
  }, []);

  // O retorno é o mesmo
  return ...
}
```

O componente App será renderizado novamente em cada redimensionamento, mesmo que este valor não seja retornado do hook!

Exemplo interativo e código completo
https://advanced-react.com/examples/01/05

Os hooks são, essencialmente, apenas bolsos em suas calças. Se, em vez de carregar um peso de 10 quilos nas mãos, você o colocasse no bolso, isso não mudaria o fato de que ainda é difícil correr: você teria 10 quilos de peso adicional em seu corpo. Mas, se você colocasse esses dez quilos em um carrinho autônomo, você poderia correr livremente e com energia, e talvez até fazer uma pausa para tomar um café: o carrinho cuidaria de si mesmo. Os componentes para o estado são esse carrinho.

A mesma lógica se aplica aos hooks que utilizam outros hooks: qualquer coisa que possa desencadear uma renderização, não importa o quão profundo na cadeia de hooks esteja acontecendo, irá desencadear uma renderização no componente que a utiliza.

aquele primeiro "hook". Se eu extrair esse estado adicional para um "hook" que retorna null, o App ainda renderizará em cada redimensionamento:

```javascript
const useResizeDetector = () => {
  const [width, setWidth] = useState(0);

  useEffect(() => {
    const listener = () => {
      setWidth(window.innerWidth);
    };

    window.addEventListener('resize', listener);

    return () => window.removeEventListener('resize', listener);
  }, []);

  return null;
}

const useModalDialog = () => {
  // Eu nem sequer o uso, apenas o chamo aqui
  useResizeDetector();

  // o retorno é o mesmo
  return ...
}

const App = () => {
  // este hook utiliza useResizeDetector para disparar
  atualização de estado em redimensionamento
  // o App inteiro renderizará em cada redimensionamento!
  const { isOpen, open, close } = useModalDialog();

  return // o mesmo retorno
}
```

Exemplo interativo e código completo

https://advanced-react.com/examples/01/06
Portanto, tenha cuidado com eles.
Para corrigir nosso aplicativo, você ainda precisaria extrair aquele botão, diálogo e o hook personalizado em um componente:

```
const ButtonWithModalDialog = () => {
  const { isOpen, open, close } = useModalDialog();

  // render apenas o Button e o ModalDialog aqui
  return (
    <>
      <Button onClick={open}>Abrir diálogo</Button>
      {isOpen ? <ModalDialog onClose={close} /> : null}
    </>
  );
};
```

Exemplo interativo e código completo
https://advanced-react.com/examples/01/07
Portanto, onde você coloca o estado é muito importante. Idealmente, para evitar problemas de desempenho no futuro, você gostaria de isolá-lo tanto quanto possível, criando componentes o mais pequenos e leves possível. No próximo capítulo (Capítulo 2: Elementos, filhos como props e re-renderizações), veremos outro padrão que ajuda exatamente nisso.
Principais pontos a serem destacados
Este é apenas o começo. Nos próximos capítulos, exploraremos mais detalhes sobre como tudo isso funciona. Enquanto isso, aqui estão alguns pontos principais:

pontos importantes deste capítulo:

A re-renderização é como o React atualiza os componentes com novos dados.
Sem re-renderizações, não haverá interação nas nossas aplicações.
A atualização de estado é a fonte inicial de todas as re-renderizações.
Se a re-renderização de um componente for acionada, todos os componentes aninhados dentro desse componente serão re-renderizados.
Durante o ciclo normal de re-renderização do React (sem o uso de memoização), a alteração de props não importa: os componentes re-renderizarão, mesmo que não tenham nenhuma prop.
Podemos usar o padrão conhecido como "mover o estado para baixo" para evitar re-renderizações desnecessárias em aplicações grandes.
Uma atualização de estado em um hook irá acionar a re-renderização de um componente que utiliza este hook, mesmo que o estado em si não seja usado.
No caso de hooks que utilizam outros hooks, qualquer atualização de estado dentro dessa cadeia de hooks irá acionar a re-renderização de um componente que utiliza o primeiro hook.

Capítulo 2. Elementos, componentes e re-renderizações

No capítulo anterior, exploramos como as mudanças de estado desencadeiam re-renderizações subsequentes em nossos aplicativos e como isso pode ser gerenciado usando o padrão "mover o estado para baixo". No entanto, o exemplo ali era relativamente simples, e o estado era bastante isolado. Portanto, movê-lo para um componente foi fácil. Quais são nossas opções quando a situação é um pouco mais complexa?

É hora de continuarmos nossa exploração de como as re-renderizações funcionam, realizarmos mais uma análise de desempenho e aprofundarmos nos detalhes. Neste capítulo, você aprenderá:

Como passar componentes como propriedades pode melhorar o desempenho de nossos aplicativos.
Como o React exatamente desencadeia re-renderizações.
Por que os componentes como propriedades não são afetados por re-renderizações.
O que é um Elemento, como ele difere de um Componente, e por que é importante conhecer essa distinção.
Os fundamentos da reconciliação e diferenciação do React.
O que é o padrão "children como propriedades", e como ele pode impedir re-renderizações.

O problema

Imagine novamente que você herdou um aplicativo grande, complexo e muito sensível ao desempenho. E esse aplicativo tem uma área de conteúdo que pode ser rolada.

Provavelmente, um layout elaborado com uma barra de cabeçalho fixa, uma barra lateral dobrável à esquerda e a restante da funcionalidade no centro.
O código para essa área principal e rolável se parece com o seguinte:

```javascript
const App = () => {
  return (
    <div className="scrollable-block">
      <VerySlowComponent />
      <BunchOfStuff />
      <OtherStuffAlsoComplicated />
    </div>
  );
};
```

Apenas um `div` com uma classe e `overflow: auto` abaixo.
E muitos componentes muito lentos dentro desse `div`. No seu primeiro dia de trabalho, você é solicitado a implementar um recurso muito criativo: um bloco que aparece na parte inferior da área quando um usuário rola um pouco e se move lentamente para cima à medida que o usuário continua rolando. Ou, se o usuário rola para cima, ele se move lentamente para baixo e desaparece. Algo como um bloco de navegação secundário com alguns links úteis. E, claro, o rolamento e tudo o que está associado a ele devem ser suaves e sem atrasos.

A maneira mais simples de implementar esses requisitos seria anexar um manipulador de rolamento ao `div` rolável, capturar o valor rolado e calcular a posição do `div` flutuante com base nisso:

```javascript
const MainScrollableArea = () => {
  const [position, setPosition] = useState(300);

  const onScroll = (e) => {
    // calcular a posição com base no valor rolado
    const calculated = getPosition(e.target.scrollTop);
    // salvar no estado
```

```
    setPosition(calculado);
  };

  return (
    <div className="scrollable-block" onScroll={onScroll}>
      {/* passar o valor da posição para o novo componente móvel */}
      <MovingBlock position={position} />
      <VerySlowComponent />
      <BunchOfStuff />
      <OtherStuffAlsoComplicated />
    </div>
  );
};
```

Exemplo interativo e código completo
https://advanced-react.com/examples/02/01

No entanto, do ponto de vista do desempenho e das re-renderizações, isso está muito longe do ideal. Cada rolagem acionará uma atualização de estado, e, como já sabemos, a atualização de estado acionará uma re-renderização do componente App e de todos os componentes aninhados. Assim, todos os componentes muito lentos serão re-renderizados, e a experiência de rolagem será lenta e com atraso.

Exatamente o oposto do que precisamos.

E, como você pode ver, não podemos simplesmente extrair esse estado para um componente. A função setPosition é usada na função onScroll, que está anexada ao div que envolve tudo.

Então, o que fazer aqui? Memoização ou alguma mágica com a passagem de Ref? Não necessariamente! Como antes, existe uma opção mais simples. Ainda podemos extrair esse estado e tudo o que é necessário para que o estado funcione para um componente:

```
const ScrollableWithMovingBlock = () => {
  const [position, setPosition] = useState(300);
```

```
  const onScroll = (e) => {
    const calculado = getPosition(e.target.scrollTop);
    setPosition(calculado);
  };

  return (
    <div className="scrollable-block" onScroll={onScroll}>
      <MovingBlock position={position} />
      {/* Um conjunto de elementos lentos estava aqui antes, mas não mais */}
    </div>
  );
};
```

E então, basta passar esse conjunto de elementos para aquele componente como propriedades.
Algo como isto:

```
const App = () => {
  const componentesLentos = (
    <>
      <VerySlowComponent />
      <BunchOfStuff />
      <OtherStuffAlsoComplicated />
    </>
  );

  return (
    <ScrollableWithMovingBlock content={componentesLentos} />
  );
};
```

Basta criar uma propriedade "content" no nosso componente ScrollableWithMovingBlock que aceita elementos React (mais detalhes sobre eles mais tarde). E então, dentro de ScrollableWithMovingBlock, aceite essa propriedade e coloque-a onde ela deveria ser renderizada:

```
// adicionar a propriedade "content" ao componente
const ScrollableWithMovingBlock = ({ content }) => {
  const [position, setPosition] = useState(0);
  const onScroll = () => {...} // igual ao anterior

  return (
    <div className="scrollable-block" onScroll={onScroll}>
      <MovingBlock position={position} />
      {content}
    </div>
  )
}
```

Agora, vamos falar sobre a atualização de estado e a situação de re-renderização. Se uma atualização de estado for
desencadeada, voltamos a desencadear uma re-renderização de um componente, como
de costume. No entanto, neste caso, será o
componente ScrollableWithMovingBlock - apenas um `div` com um
bloco móvel. Os outros componentes lentos são passados como props, e estão fora desse componente. Na árvore de componentes "hierárquica", eles pertencem ao componente pai. E lembre-se? O React nunca
"sobe" nessa árvore quando re-renderiza um componente. Portanto, nossos componentes lentos não re-renderizarão quando o estado for atualizado, e a experiência de rolagem será suave e sem atrasos.

Exemplo interativo e código completo
https://advanced-react.com/examples/02/02

Espere um momento, alguns podem pensar aqui. Isso não faz muito sentido.

Sim, esses componentes são declarados no componente pai, mas ainda são renderizados dentro desse componente com o estado. Então, por que eles não re-renderizam? É realmente uma pergunta muito razoável.

Para entender tudo isso, precisamos entender algumas coisas: o que realmente queremos dizer com "re-renderizar" no React, qual a diferença entre um

Elemento e Componente, e os fundamentos dos algoritmos de reconciliação e comparação.
Elementos, Componentes e re-renderizações

Em primeiro lugar, um Componente - o que é? Aqui está o mais simples:

```
const Parent = () => {
  return <Child />;
};
```

Como pode ver, é apenas uma função. O que diferencia um componente de qualquer outra função é que ele retorna Elementos, que o React então converte em elementos DOM e envia ao navegador para serem desenhados na tela. Se ele tiver propriedades, essas seriam apenas os primeiros argumentos dessa função:

```
const Parent = (props) => {
  return <Child />;
};
```

Esta função retorna <Child />, que é um Elemento de um Componente Filho. Sempre que usamos esses colchetes em um componente, criamos um Elemento. O Elemento do componente pai seria <Parent />.
Um Elemento é simplesmente um objeto que define um componente que precisa ser renderizado na tela. Na verdade, a sintaxe HTML-like é apenas açúcar sintático para a função React.createElement[2].
Podemos até substituir esse elemento por isto:
React.createElement(Child, null, null) e tudo funcionará conforme o esperado.

A definição do elemento `<Child />` seria algo como:

```
{
  type: Child,
  props: {}, // se o Child tivesse propriedades
  ... // muitas outras coisas internas do React
}
```

Isso nos diz que o componente Pai, que retorna essa definição,
quer que renderizemos o componente Child sem propriedades. A saída do
componente Child terá suas próprias definições, e assim por diante, até
chegarmos ao final dessa cadeia de componentes.

Os elementos não estão limitados a componentes; eles podem ser apenas elementos DOM normais. Nosso Child poderia retornar uma tag <h1>, por exemplo:

```
const Child = () => {
  return <h1>Algum título</h1>;
};
```

Nesse caso, o objeto de definição será exatamente o mesmo e terá o mesmo comportamento, apenas o tipo será uma string:

```
{
  type: "h1",
  ... // propriedades e coisas internas do React
}
```

Agora, para re-renderizar. O que geralmente chamamos de "re-renderizar" é o React chamando essas funções e executando tudo o que precisa ser executado no processo (como os hooks). A partir da saída dessas funções,
o React constrói uma árvore desses objetos. Nós o conhecemos agora como a "Fiber Tree" ou o "Virtual DOM" às vezes. Na verdade, são até duas árvores: antes e depois do re-
renderizar. Comparando ("diffing") essas, o React então extrai

informações que são enviadas ao navegador: quais elementos do DOM precisam ser atualizados, removidos ou adicionados. Isso é conhecido como o algoritmo de "reconciliação".

A parte que é importante para o problema deste capítulo é a seguinte: se o objeto (Elemento) antes e depois da re-renderização forem exatamente os mesmos, então o React irá pular a re-renderização do Componente que este Elemento representa e seus componentes aninhados. E por "exatamente os mesmos", quero dizer se Object.is(ElementoAntesDaReRenderização, ElementoDepoisDaReRenderização) retorna true. O React não realiza a comparação profunda de objetos. Se o resultado desta comparação for verdadeiro, então o React deixará esse componente em paz e passará para o próximo. Se a comparação retornar falso, isso é o sinal para o React de que algo mudou. Então, ele verificará o tipo. Se o tipo for o mesmo, então o React re-renderizará este componente. Se o tipo mudar, então ele removerá o componente "antigo" e montará o "novo". Analisaremos isso em mais detalhes no Capítulo 6. Uma análise aprofundada de diferenciação e reconciliação.

Vamos analisar novamente o exemplo "pai/filho" e imaginar que nosso pai tem o estado:

```
const Pai = (props) => {
  const [estado, setEstado] = useState();

  return <Filho />;
};
```

Quando setEstado é chamado, o React saberá que deve re-renderizar o componente Pai. Então, ele chamará a função Pai e comparará o que ele retorna antes e depois das alterações no estado. E ele retorna um objeto que é definido localmente para a função Pai. Então, em cada chamada de função (ou seja, re-renderização), este objeto será recriado, e o resultado de Object.is

Quando os objetos "<Child />" são definidos como "antes" e "depois", eles serão falsos. Como resultado,
sempre que o Pai aqui re-renderiza, o Filho também re-renderizará.
O que já sabemos, mas é bom ter essa confirmação, não é?
Agora, imagine o que acontecerá aqui se, em vez de renderizar diretamente o
componente Filho, eu o passasse como uma propriedade?

```
const Pai = ({ filho }) => {
  const [estado, setEstado] = useState();

  return filho;
};

// alguém em algum lugar renderiza o componente Pai assim
<Pai filho={<Filho />} />;
```

Em algum lugar, onde o componente Pai é renderizado, o objeto <Filho
/> é criado e passado para ele como a propriedade filho.
Quando a atualização do estado no Pai é acionada, o React comparará o que
a função Pai retorna "antes" e "depois" da mudança de estado. E neste caso, será uma referência ao filho: um objeto que é criado
fora do escopo da função Pai e, portanto, não muda
quando é chamado. Como resultado, a comparação do filho "antes" e
"depois" retornará verdadeiro, e o React pulará a re-renderização deste
componente.

E é exatamente o que fizemos para o nosso componente com a rolagem!

```
const ScrollávelComBlocoEmMovimento = ({ conteúdo }) => {
  const [posição, setPosição] = useState(300);
  const onScroll = () => {...} // igual ao anterior

  return (
    <div className="scrollable-block" onScroll={onScroll}>
      <BlocoEmMovimento posição={posição} />
      {conteúdo}
```

```
    </div}
}
```
Quando `setPosition` em `ScrollableWithMovingBlock` é
acionado e a renderização é refeita, o React comparará todas as definições de objeto que a função retorna, verá que o objeto de conteúdo é exatamente o mesmo antes e depois, e saltará a renderização de qualquer coisa que esteja lá. No nosso caso - um conjunto de componentes muito lentos.
No entanto, `MovingBlock ...` refezirá a renderização: é criado dentro de `ScrollableWithMovingBlock`. O objeto será recriado em cada renderização, e a comparação "antes" e "depois" retornará `false`.

Crianças como propriedades
Embora este padrão seja ótimo e totalmente válido, existe um pequeno problema com ele: ele parece estranho. Passar todo o conteúdo da página para algumas propriedades aleatórias simplesmente parece... errado por algum motivo. Então, vamos melhorá-lo.
Primeiramente, vamos falar sobre a natureza das propriedades. As propriedades são apenas um objeto que passamos como o primeiro argumento para a função do nosso componente.
Tudo o que extraímos dele é uma propriedade. Tudo. No nosso código de "pai/filho", se eu renomear a propriedade do filho para `children`, nada mudará: continuará funcionando.
```
// antes
const Parent = ({ child }) => {
  return child;
};

// depois
const Parent = ({ children }) => {
  return children;
```

```
};
```
E, do lado do consumidor, a mesma situação: nada muda.
```
// antes
<Parent child={<Child />} />

// depois
<Parent children={<Child />} />
```
No entanto, para as propriedades dos filhos, temos uma sintaxe especial no JSX. Essa estrutura de composição agradável que usamos constantemente com tags HTML, nós nunca consideramos, mas agora prestando atenção:
```
<Parent>
  <Child />
</Parent>
```
Isso funcionará exatamente da mesma forma como se estivéssemos passando a propriedade "children" explicitamente:
```
<Parent children={<Child />} />

// exatamente igual ao acima
<Parent>
  <Child />
</Parent>
```
E será representado da seguinte forma:
```
{
  type: Parent,
  props: {
    // elemento para Child aqui
    children: {
```

```
      tipo: Criança,
      ...
    },
  }
}
```

E terá exatamente os mesmos benefícios de desempenho ao passar elementos como propriedades! Qualquer coisa que seja passada como propriedade não será afetada pela mudança de estado do componente que recebe essas propriedades.

Portanto, podemos reescrever nosso App da seguinte forma:

```
const App = () => {
  const slowComponents = (
    <>
      <VerySlowComponent />
      <BunchOfStuff />
      <OtherStuffAlsoComplicated />
    </>
  );

  return (
    <ScrollableWithMovingBlock content={slowComponents} />
  );
};
```

Para algo muito mais bonito e fácil de entender:

```
const App = () => {
  return (
    <ScrollableWithMovingBlock>
      <VerySlowComponent />
      <BunchOfStuff />
      <OtherStuffAlsoComplicated />
    </ScrollableWithMovingBlock>
  );
};
```

Tudo o que precisamos fazer no componente ScrollableWithMovingBlock é renomear a propriedade "content" para "children", e não mais! Antes:

```
const ScrollableWithMovingBlock = ({ content }) => {
  // ... o restante do código

  return (
    <div ...>
      ...
      {content}
    </div>
  )
}
```

Depois:

```
const ScrollableWithMovingBlock = ({ children }) => {
  // ... o restante do código

  return (
    <div ...>
      ...
      {children}
    </div>
  )
}
```

E aqui está: implementamos um bloco de rolagem muito eficiente em uma aplicação muito lenta, utilizando apenas um pequeno truque de composição.

Exemplo interativo e código completo

https://advanced-react.com/examples/02/03

Principais aprendizados

Espero que isso tenha sido claro e que agora você se sinta confiante com os padrões "componentes como propriedades" e "children como propriedades". No próximo capítulo, vamos ver como os componentes como propriedades podem ser úteis além do desempenho.
Enquanto isso, aqui estão algumas coisas para lembrar:
Um Componente é simplesmente uma função que aceita um argumento (propriedades) e retorna Elementos que devem ser renderizados quando este
Componente é renderizado na tela. `const A = () => <B />` é um Componente.
Um Elemento é um objeto que descreve o que deve ser renderizado
na tela, com o tipo sendo uma string para elementos do DOM ou uma
referência a um Componente para componentes. `const b = <B />` é um Elemento.
A re-renderização é simplesmente o React chamando a função do Componente.
Um componente re-renderiza quando seu objeto de elemento muda, conforme determinado pela comparação `Object.is` antes e depois da re-
renderização.
Quando elementos são passados como propriedades para um componente, e este
componente aciona uma re-renderização através de uma atualização de estado, os elementos passados como propriedades não serão re-renderizados.
"children" são apenas propriedades e se comportam como qualquer outra propriedade quando
são passados ??através da sintaxe de aninhamento JSX:
`<Parent>
  <Child />
</Parent>

// o mesmo que:
`<Parent children={<Child />} />`

preocupações com elementos como propriedades
No capítulo anterior, exploramos como passar elementos como propriedades pode
melhorar o desempenho de nossos aplicativos. No entanto, as melhorias de desempenho não são o uso mais comum desse padrão. Na verdade, são mais um efeito colateral agradável e relativamente desconhecido. O principal caso de uso que este padrão resolve é realmente a flexibilidade e a configuração de componentes.

Vamos continuar nossa investigação sobre como o React funciona. Desta vez, vamos criar um componente simples de "botão com ícone". O que poderia ser complicado nisso, certo? Mas, ao construir, você descobrirá:
Como elementos como propriedades podem melhorar drasticamente as preocupações de configuração para esses componentes.
Como o renderização condicional de componentes influencia o desempenho.
Quando um componente passado como propriedade é renderizado exatamente.
Como definir propriedades padrão para componentes passados como propriedades usando a função `cloneElement`, e quais são os desvantagens disso.
Pronto? Vamos lá!
O problema
Imagine, por exemplo, que você precise implementar um componente de "botão". Um dos requisitos é que o botão deva ser capaz de

mostre o ícone de "carregamento" no lado direito quando for usado no contexto de "carregamento". Este é um padrão bastante comum para o envio de dados em formulários.
Sem problemas! Podemos simplesmente implementar o botão e adicionar a propriedade `isLoading`, com base na qual renderizaremos o ícone.

```javascript
const Button = ({ isLoading }) => {
  return (
    <button>Enviar {isLoading ? <Loading /> : null}</button>
  );
};
```

No dia seguinte, este botão precisa suportar todos os ícones disponíveis da sua biblioteca, não apenas o "Loading". Ok, podemos adicionar a propriedade `iconName` ao botão para isso. No dia seguinte, as pessoas querem poder controlar a cor desse ícone para que ele se alinhe melhor com a paleta de cores usada no site. A propriedade `iconColor` é adicionada. Em seguida, `iconSize`, para controlar o tamanho do ícone. E então, surge um caso de uso para que o botão também possa suportar ícones no lado esquerdo, além de avatares.

Eventualmente, metade das propriedades no botão existem apenas para controlar esses ícones; ninguém consegue entender o que está acontecendo internamente, e cada alteração resulta em alguma funcionalidade quebrada para os clientes.

```javascript
const Button = ({
  isLoading,
  iconLeftName,
  iconLeftColor,
  iconLeftSize,
  isIconLeftAvatar,
  ...
  }) => {
  // ninguém sabe o que está acontecendo aqui e como todas essas propriedades funcionam
  return ...
}
```

Parece familiar?
Elementos como propriedades
Felizmente, existe uma maneira fácil de melhorar significativamente essa situação. Tudo o que precisamos fazer é remover essas propriedades de configuração e passar o ícone como um Elemento:

```javascript
const Button = ({ icon }) => {
  return <button>Enviar {icon}</button>;
};
```

E então deixar que o consumidor configure esse ícone da maneira que desejar:

// Ícone de carregamento padrão
<Button icon={<Loading />} />

// Ícone de erro vermelho
<Button icon={<Error color="red" />} />

// Ícone de aviso amarelo grande
<Button icon={<Warning color="yellow" size="large" />} />

// Avatar em vez de ícone
<Button icon={<Avatar />} />

Exemplo interativo e código completo
https://advanced-react.com/examples/03/01

Se usar algo como isso para um Botão é uma boa ideia ou não, é algo que pode gerar discussões, é claro. Depende muito do quão rigoroso você precisa que seja.

o design e a quantidade de variação que ele permite para aqueles que o implementam, especialmente em relação às funcionalidades do produto.
Mas imagine implementar algo como uma caixa de diálogo com uma barra de título, área de conteúdo e rodapé com alguns botões.
A menos que seus designers sejam muito rigorosos e experientes, é provável que você precise ter diferentes configurações desses botões em diferentes caixas de diálogo: um, dois, três botões, um botão é um link, um botão é "primário", diferentes textos, diferentes ícones, diferentes dicas de ferramenta, etc. Imagine passar tudo isso por meio de propriedades de configuração!
Mas com elementos como propriedades, fica muito mais fácil: basta criar uma propriedade de rodapé na caixa de diálogo

```
const ModalDialog = ({ content, footer }) => {
  return (
    <div className="modal-dialog">
      <div className="content">{content}</div>
      <div className="footer">{footer}</div>
    </div>
  );
};
```

e então passar o que for necessário:

```
// apenas um botão no rodapé
<ModalDialog content={<SomeFormHere />} footer={<SubmitButton />}
/>

// dois botões
<ModalDialog
  content={<SomeFormHere />}
  footer={<><SubmitButton /><CancelButton /></>}
/>
```

Exemplo interativo e código completo
https://advanced-react.com/examples/03/02
Ou algo como um componente ThreeColumnsLayout, que
exibe três colunas com algum conteúdo na tela. Neste caso, você
não consegue nem fazer nenhuma configuração: ele literalmente pode e deve exibir
qualquer coisa nessas colunas.

```
<ThreeColumnsLayout
  leftColumn={<Something />}
  middleColumn={<OtherThing />}
  rightColumn={<SomethingElse />}
/>
```

Exemplo interativo e código completo
https://advanced-react.com/examples/03/03
Essencialmente, um elemento como uma prop para um componente é uma maneira de dizer ao consumidor: me dê o que você quiser, não importa o que seja, eu não sei nem me importo, eu só sou responsável por colocá-lo no lugar certo. O resto é por sua conta.
E, claro, o padrão "children" como props, descrito no capítulo anterior, também é muito útil aqui. Se quisermos passar algo que consideramos uma "parte principal" desse componente, como a área de "conteúdo" no
diálogo modal, ou a coluna do meio no layout com três colunas, podemos
simplesmente usar a sintaxe aninhada para isso:

```
// antes
<ModalDialog
  content={<SomeFormHere />}
  footer={<SubmitButton />}
/>
```

```
// após
<ModalDialog
  footer={<SubmitButton />}
>
  <SomeFormHere />
</ModalDialog>
```
O que precisamos fazer, do ponto de vista do ModalDialog, é renomear a propriedade "content" para "children":
```
const ModalDialog = ({ children, footer }) => {
  return (
    <div className="dialog">
      <div className="content">{children}</div>
      <div className="footer">{footer}</div>
    </div>
  );
};
```
Exemplo interativo e código completo
https://advanced-react.com/examples/03/04
Lembre-se sempre: "children" neste contexto são apenas uma propriedade, e a sintaxe "aninhada" é apenas uma forma de simplificar a sintaxe!

Renderização condicional e desempenho

Uma das maiores preocupações que podem surgir com este padrão é o desempenho. Isso é irônico, considerando que, no capítulo anterior, discutimos como usá-lo para melhorar o desempenho. Então, o que está acontecendo?

Imagine que renderizamos o componente que aceita elementos como propriedades de forma condicional. Assim como nosso ModalDialog, que normalmente seria renderizado apenas quando a variável `isDialogOpen` for verdadeira:

```javascript
const App = () => {
  const [isDialogOpen, setIsDialogOpen] = useState(false);

  // Quando este componente será renderizado?
  const footer = <Footer />;

  return isDialogOpen ? (
    <ModalDialog footer={footer} />
  ) : null;
};
```

A questão aqui, com a qual até mesmo desenvolvedores experientes às vezes lutam, é a seguinte: declaramos nosso `Footer` antes do diálogo. Embora o diálogo ainda esteja fechado e não seja aberto por um tempo (ou talvez nunca). Isso significa que o `Footer` sempre será renderizado, mesmo que o diálogo não esteja na tela? E quais são as implicações de desempenho? Isso não irá tornar o componente `App` mais lento?

Felizmente, não há nada para se preocupar aqui. Lembre-se, no Capítulo 2, Elementos, filhos como propriedades e re-renderizações, discutimos o que é um Elemento? Tudo o que fizemos ao declarar a variável `footer` (`footer = <Footer />`) foi criar um Elemento, nada mais.

Do ponto de vista do React e do código, é apenas um objeto que permanece em memória sem fazer nada. E criar objetos é barato (pelo menos, em comparação com a renderização de componentes).

O `Footer` será realmente renderizado apenas quando estiver presente no objeto de retorno de um dos componentes, e não antes. No nosso caso, será o componente `ModalDialog`. Não importa que o elemento `<Footer />` tenha sido criado no `App`. É o `ModalDialog` que irá pegá-lo e renderizá-lo:

```
const ModalDialog = ({ children, footer }) => {
  return (
    <div className="dialog">
      <div className="content">{children}</div>
      {/* Qualquer coisa que venha da propriedade footer será renderizada
apenas quando este componente inteiro for renderizado */}
      {/* não antes */}
      <div className="footer">{footer}</div>
    </div>
  );
};
```

É isso que torna os padrões de roteamento, como nos diferentes versões do React Router, completamente seguros:

```
const App = () => {
  return (
    <>
      <Route path="/algum/caminho" element={<Page />} />
      <Route path="/outro/caminho" element={<OtherPage />} />
      ...
    </>
  );
};
```

Não há nenhuma condição aqui, então parece que o App possui e renderiza tanto o `<Page />` quanto o `<OtherPage />` ao mesmo tempo. Mas não. Ele apenas cria pequenos objetos que descrevem essas páginas. A renderização real só acontecerá quando o caminho em uma das rotas corresponder à URL e o elemento da propriedade `element` for realmente retornado do componente `Route`.

Valores padrão para os elementos provenientes de propriedades

Vamos falar um pouco mais sobre nossos botões e seus ícones.
Uma das objeções contra o uso desses ícones como propriedades é que esse padrão é muito flexível. Está tudo bem se o componente ThreeColumnsLayout aceitar qualquer coisa na propriedade leftColumn. Mas, no caso do Button, realmente não queremos passar tudo lá. No mundo real, o Button precisaria ter algum grau de controle sobre os ícones. Se o botão tem a propriedade isDisabled, provavelmente você deseja que o ícone também apareça "desativado". Botões maiores desejariam ícones maiores por padrão, botões azuis desejariam ícones brancos por padrão, e botões brancos desejariam ícones pretos.
No entanto, se deixarmos a implementação como está, isso será problemático: caberá aos consumidores do Button lembrar de tudo.

```
// O botão principal deve ter ícones brancos
<Button aparência="primary" icon={<Loading cor="white" />} />

// O botão secundário deve ter ícones pretos
<Button aparência="secondary" icon={<Loading cor="black" />} />

// O botão grande deve ter ícones grandes
<Button tamanho="large" icon={<Loading tamanho="large" />} />
```

Na maioria das vezes, isso será esquecido, e na outra metade, mal interpretado.
O que precisamos aqui é atribuir alguns valores padrão a esses ícones que o Button pode controlar, mantendo a flexibilidade do padrão.
Felizmente, podemos fazer exatamente isso. Lembre-se de que esses ícones nas propriedades são apenas objetos com formas conhecidas e previsíveis. E o React tem APIs que nos permitem trabalhar com eles facilmente. No nosso caso, podemos clonar o ícone no Button com a ajuda da função React.cloneElement[3], e atribuir qualquer propriedade a esse novo elemento que desejamos. Assim, nada nos impede de criar algumas propriedades de ícone padrão, combinando-as