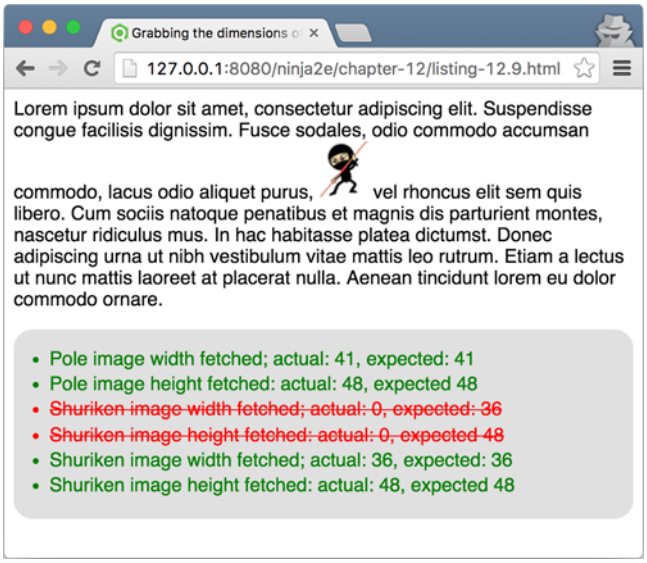


Figura 12.5 Temporariamente ajustando as propriedades de estilo dos elementos ocultos, podemos obter com sucesso suas dimensões.

12.4 Minimizando o thrashing do layout

Até agora neste capítulo, você aprendeu como modificar o DOM com relativa facilidade: criando e inserindo novos elementos, removendo elementos existentes ou modificando seus atributos. Modificar o DOM é uma das ferramentas fundamentais para obter aplicativos da web altamente dinâmicos.

Mas essa ferramenta também vem com asteriscos de uso, sendo um dos mais importantes, esteja ciente de *layout thrashing*. O thrashing de layout ocorre quando executamos uma série de leituras e gravações consecutivas no DOM, no processo, não permitindo que o navegador execute otimizações de layout.

Antes de nos aprofundarmos, considere que alterar os atributos de um elemento (ou modificar seu conteúdo) não afeta necessariamente apenas aquele elemento; em vez disso, pode causar uma cascata de mudanças. Por exemplo, definir a largura de um elemento pode levar a alterações nos filhos, irmãos e pais do elemento. Portanto, sempre que uma alteração é feita, o navegador deve calcular o impacto dessas alterações. Em certos casos, não há nada que possamos fazer a respeito; precisamos que essas mudanças ocorram. Mas, ao mesmo tempo, não há necessidade de colocar peso adicional sobre os ombros de nossos navegadores fracos, fazendo com que o desempenho de nosso aplicativo da web diminua.

Como recalcular o layout é caro, os navegadores tentam ser o mais preguiçosos possível, atrasando o trabalho com o layout o máximo que podem; eles tentam colocar em lote tantas operações de gravação quanto possível no DOM em uma fila para que essas operações possam ser executadas de uma vez. Então, quando surge uma operação que requer um layout atualizado, o navegador obedece a contragosto, executa todas as operações em lote e, finalmente, atualiza o layout. Mas às vezes, a maneira como vamos escrever nosso código não dá ao navegador espaço suficiente para realizar esses tipos de otimizações, e forçamos o

navegador para realizar muitos recálculos (possivelmente desnecessários). É disso que trata o layout; ocorre quando nosso código executa uma série de leituras e gravações consecutivas (muitas vezes desnecessárias) no DOM, não permitindo que o navegador otimize as operações de layout. O problema é que, sempre que modificamos o DOM, o navegador precisa recalcular o layout antes que qualquer informação de layout seja lida. Essa ação é cara, em termos de desempenho. Vamos dar uma olhada em um exemplo.

Listagem 12.10 Séries consecutivas de leituras e gravações causam problemas de layout

```
<div id = "ninja"> Sou um ninja </div> <div id = "samurai"> Sou um
samurai </div> <div id = "ronin"> Sou um ronin </ div > <script>

    const ninja = document.getElementById ("ninja"); const samurai = document.getElementById
    ("samurai"); const ronin = document.getElementById ("ronin");

    const ninjaWidth = ninja.clientWidth; ninja.style.width = ninjaWidth / 2 + "px";

    const samuraiWidth = samurai.clientWidth; samurai.style.width = samuraiWidth / 2 +
    "px";

    const roninWidth = ronin.clientWidth; ronin.style.width = roninWidth / 2 + "px";
</script>
```

Define alguns elementos HTML

Coleta os elementos do DOM

Executa uma série de leituras consecutivas e escreve. Modificações DOM invalidar o layout.

Lendo o valor do elemento clientWidth propriedade é uma daquelas ações que exige que o navegador tenha um layout atualizado. Ao realizar leituras e gravações consecutivas na propriedade width de diferentes elementos, não permitimos que o navegador seja preguiçoso de maneira inteligente. Em vez disso, como lemos as informações de layout após cada modificação de layout, o navegador precisa recalcular o layout todas as vezes, apenas para ter certeza de que ainda obteremos as informações corretas.

Uma forma de minimizar o thrashing de layout é criar um código de maneira que não cause recálculos de layout desnecessários. Por exemplo, podemos reescrever a listagem 12.10 no seguinte.

Listagem 12.11 Leituras e gravações Batch DOM para evitar sobrecarga de layout

```
<div id = "ninja"> Sou um ninja </div> <div id = "samurai"> Sou um
samurai </div> <div id = "ronin"> Sou um ronin </ div > <script>

    const ninja = document.getElementById ("ninja"); const samurai = document.getElementById
    ("samurai"); const ronin = document.getElementById ("ronin");

    const ninjaWidth = ninja.clientWidth; const samuraiWidth = samurai.clientWidth;
    const roninWidth = ronin.clientWidth;
```

Coloca em lote todas as leituras para propriedades de layout juntas

```
ninja.style.width = ninjaWidth / 2 + "px"; samurai.style.width = samuraiWidth / 2 +
"px"; ronin.style.width = roninWidth / 2 + "px"; </script>
```

Coloca em lote todas as gravações nas propriedades de layout juntas

Aqui, agrupamos todas as leituras e gravações, porque sabemos que não existem dependências entre as dimensões de nossos elementos; definindo a largura do ninja elemento não influencia a largura do samurai elemento. Isso permite que o navegador preguiçosamente em lote operações que modificam o DOM.

A alteração do layout não é algo que você notaria em páginas menores e mais simples, mas é algo para se manter em mente ao desenvolver aplicativos da web complexos, especialmente em dispositivos móveis. Por esse motivo, é sempre bom ter em mente os métodos e propriedades que requerem um layout atualizado, mostrados na tabela a seguir (obtida em http://ricostacruz.com/cheatsheets/layout-thrashing.html)

Tabela 12.2 APIs e propriedades que causam invalidação de layout

Interface	Nome da propriedade
Elemento	clientHeight, clientLeft, clientTop, clientWidth, focus, getBoundingClientRect, getClientRects, innerText, offsetHeight, offsetLeft, offsetParent, offsetTop, offsetWidth, outerText, scrollByLines, scrollByPages, scrollHeight, scrollIntoView, scrollIntoViewIfNeeded, scrollLeft, scrollTop, scrollWidth
MouseEvent	layerX, layerY, offsetX, offsetY
Janela	getComputedStyle, scrollBy, scrollTo, scroll, scrollY height, width
Quadro, Documento, Imagem	

Várias bibliotecas que tentam minimizar o thrashing de layout foram desenvolvidas. Um dos mais populares é o FastDom (https://github.com/wilsonpage/fastdom) O repositório da biblioteca inclui exemplos que mostram claramente os ganhos de desempenho que podem ser alcançados por lote de operações de leitura / gravação DOM (https://wilsonpage.github.io/ fastdom / examples / aspect-ratio.html)

DOM virtual do React

Uma das bibliotecas mais populares do lado do cliente é o React do Facebook (https: // facebook.github.io/react/) O React alcança um ótimo desempenho usando um DOM virtual, um conjunto de objetos JavaScript que imitam o DOM real. Quando desenvolvemos aplicações no React, realizamos todas as modificações no DOM virtual, sem qualquer preocupação com a alteração do layout. Então, em um momento apropriado, o React usa o DOM virtual para descobrir quais alterações devem ser feitas no DOM real, a fim de manter a IU em sincronia. Esse lote de atualizações aumenta o desempenho dos aplicativos.

12.5 Resumo

- A conversão de uma string HTML em elementos DOM inclui as seguintes etapas:
  - Certificando-se de que a string HTML é um código HTML válido
  - Envolvendo-o em marcação envolvente, exigido pelas regras do navegador
  - Inserir o HTML em um elemento DOM fictício por meio do innerHTML propriedade de um elemento DOM
  - Extrair os nós DOM criados de volta
- Para inserção rápida de nós DOM, use fragmentos DOM, porque um fragmento pode ser injetado em uma única operação, reduzindo drasticamente o número de operações.
- Os atributos e propriedades do elemento DOM, embora vinculados, nem sempre são idênticos! Podemos ler e gravar em atributos DOM usando o getAttribute e setAttribute métodos, enquanto gravamos nas propriedades do DOM usando a notação de propriedade do objeto.
- Ao trabalhar com atributos e propriedades, devemos estar cientes de *atributos personalizados*. Os atributos que decidimos colocar em elementos HTML para transportar informações úteis para nossos aplicativos não são apresentados automaticamente como propriedades de elementos.
- O estilo propriedade do elemento é um objeto que contém propriedades correspondentes aos valores de estilo especificados na marcação do elemento. Para obter os estilos calculados, que também levam em consideração os estilos definidos nas folhas de estilo, use o recurso integrado obter-ComputedStyle método.
- Para obter as dimensões dos elementos HTML, use offsetWidth e offsetHeight propriedades.
- O thrashing do layout ocorre quando o código executa uma série de leituras e gravações consecutivas no DOM, sempre forçando o navegador a recalcular as informações do layout. Isso leva a aplicativos da web mais lentos e menos responsivos.
- Reúna suas atualizações de DOM!

12.6 Exercícios

1 No código a seguir, qual das afirmações a seguir será aprovada?

```
<div id = "samurai"> </div>
<script>
    elemento const = document.querySelector ("# samurai");

    assert (element.id === "samurai", "id da propriedade é samurai"); assert (element.getAttribute ("id") === "samurai",

                "id do atributo é samurai");

    element.id = "novoSamurai";
```

```
    assert (element.id === "newSamurai", "id da propriedade é newSamurai"); assert (element.getAttribute ("id") ===
    "newSamurai",
                "id do atributo é newSamurai");
</script>
```

2 Dado o código a seguir, como podemos acessar o elemento largura da borda estilo propriedade?

```
<div id = "element" style = "border-width: 1px;
                                                estilo de borda: sólido; border-color: red ">
</div>
<script>
    elemento const = document.querySelector ("# elemento"); </script>
```

uma element.border-width

b element.getAttribute ("border-width");

c element.style ["border-width"];

d element.style.borderWidth;

3 Qual método embutido pode obter todos os estilos aplicados a um determinado elemento (estilos pro fornecidos pelo navegador, estilos aplicados por meio de folhas de estilo e propriedades definidas por meio do atributo style)?

uma getStyle

b getAllStyles

c getComputedStyle

4 Quando ocorre a alteração do layout?

13 Sobrevivendo eventos

Este capítulo cobre

- Compreender o ciclo de eventos
- Processando tarefas complexas com cronômetros
- Gerenciando animações com temporizadores
- Usando bolha de evento e delegação
- Usando eventos personalizados

O Capítulo 2 incluiu uma breve discussão sobre o modelo de execução de thread único JavaScript e introduziu o loop de eventos e a fila de eventos, nos quais os eventos aguardam sua vez de serem processados. Essa discussão foi particularmente útil ao apresentar as etapas do ciclo de vida de uma página da web, especialmente ao discutir a ordem em que certas partes do código JavaScript são executadas. Ao mesmo tempo, é uma simplificação; portanto, para obter uma imagem mais completa de como o navegador funciona, passaremos uma parte significativa deste capítulo explorando os cantos e recantos do loop de eventos. Isso nos ajudará a entender melhor algumas das limitações de desempenho inerentes ao JavaScript e ao navegador. Por sua vez, usaremos esse conhecimento para desenvolver aplicativos de execução mais suave.

Durante esta exploração, colocaremos um foco especial em cronômetros, um recurso JavaScript que nos permite atrasar a execução de um trecho de código de forma assíncrona por um determinado período de tempo. À primeira vista, pode não parecer muito, mas mostraremos como usar cronômetros para dividir tarefas de longa execução que tornam os aplicativos lentos e sem resposta em tarefas menores que não obstruem o navegador. Isso ajuda a desenvolver aplicativos com melhor desempenho.

Continuaremos esta exploração de eventos, mostrando como os eventos são propagados através da árvore DOM e como usar esse conhecimento para escrever código mais simples e com menos memória intensiva. Por fim, concluiremos o capítulo criando eventos personalizados, que podem ajudar a reduzir o acoplamento entre as diferentes partes do aplicativo. Sem mais delongas, vamos começar a percorrer o loop de eventos.

Você sabe?

Por que o tempo nos retornos de chamada do temporizador não é garantido? Se um setInterval **cronômetro dispara a cada 3 milissegundos enquanto outro manipulador de eventos está sendo executado por 16 ms, quantas vezes a função de retorno de chamada do temporizador será adicionada à fila de microtarefa?**

Por que o contexto da função para um manipulador de eventos é vezes diferente do alvo do evento?

13.1 Mergulhando no ciclo de eventos

Como você deve ter percebido, o loop de eventos é mais complicado do que sua apresentação no capítulo 2. Para começar, em vez de uma única fila de eventos, que contém apenas eventos, o loop de eventos tem pelo menos duas filas que, além dos eventos, retém outras ações realizadas pelo navegador. Essas ações são chamadas *tarefas* e são agrupados em duas categorias: *macrotarefas (* ou muitas vezes apenas chamados de tarefas) e *microtarefas.*

Exemplos de macrotarefas incluem a criação do objeto de documento principal, análise de HTML, execução de código JavaScript da linha principal (ou global), alteração da URL atual, bem como vários eventos, como carregamento de página, entrada, eventos de rede e eventos de cronômetro. Da perspectiva do navegador, uma macrotarefa representa uma unidade de trabalho independente e independente. Depois de executar uma tarefa, o navegador pode continuar com outras atribuições, como renderizar novamente a IU da página ou executar a coleta de lixo.

Microtarefas, por outro lado, são tarefas menores que atualizam o estado do aplicativo e devem ser executadas antes que o navegador continue com outras atribuições, como renderizar novamente a IU. Os exemplos incluem retornos de chamada de promessa e alterações de mutação DOM. As microtarefas devem ser executadas o mais rápido possível, de forma assíncrona, mas sem o custo de executar uma macrotarefa totalmente nova. Microtarefas nos permitem executar certas ações *antes* a IU é renderizada novamente, evitando assim a renderização desnecessária da IU que pode mostrar o estado inconsistente do aplicativo.

NOTA A especificação ECMAScript não menciona loops de eventos. Em vez disso, o loop de eventos é detalhado na especificação HTML (https: // html.spec.whatwg.org/#event-loops), que também discute o conceito de macrotarefas e microtarefas. A especificação ECMAScript menciona *empregos*

(que são análogos a microtarefas) em relação ao tratamento de retornos de chamada de promessa (http://mng.bz/fOlK) . Mesmo que o loop de eventos seja definido na especificação HTML, outros ambientes, como Node.js, também o usam.

A implementação de um loop de eventos *deve* use pelo menos uma fila para macrotarefas e pelo menos uma fila para microtarefas. Implementações de loop de evento geralmente vão além disso e têm *vários* filas para diferentes tipos de macro e microtarefas. Isso permite que o loop de eventos priorize tipos de tarefas; por exemplo, dar prioridade a tarefas sensíveis ao desempenho, como entrada do usuário. Por outro lado, como existem muitos navegadores e ambientes de execução de JavaScript na selva, você não deve se surpreender se encontrar loops de eventos com apenas uma única fila para os dois tipos de tarefas juntos.

O loop de eventos é baseado em dois princípios fundamentais:

- As tarefas são tratadas uma de cada vez.
- Uma tarefa é executada até a conclusão e não pode ser interrompida por outra tarefa.

Vamos dar uma olhada na figura 13.1, que descreve esses dois princípios.

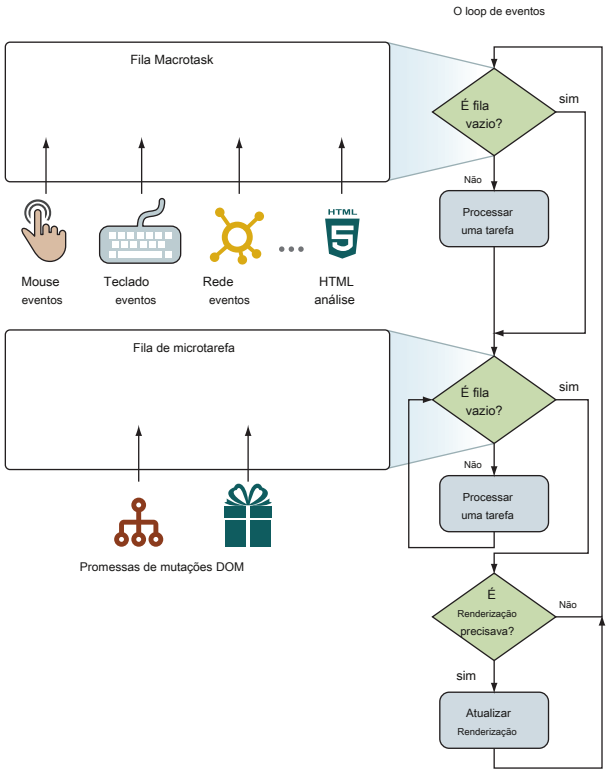


Figura 13.1 O loop de eventos geralmente tem acesso a pelo menos duas filas de tarefas: fila amicrotask e fila amacrotask. Ambos os tipos de tarefas são realizados um de cada vez.

Em um alto nível, a figura 13.1 mostra que em uma única iteração, o loop de eventos primeiro verifica a fila de macrotarefa e, se houver uma macrotarefa esperando para ser executada, inicia sua execução. Somente depois que a tarefa for totalmente processada (ou se não houver tarefas na fila), o loop de eventos passa para o processamento da fila de microtarefas. Se houver uma tarefa esperando nessa fila, o loop de eventos a pega e a executa até a conclusão. Isso é executado para todas as microtarefas na fila. Observe a diferença entre lidar com as filas de macrotarefa e microtarefa: em uma iteração de loop único, uma macrotarefa é processada no máximo (outras são deixadas esperando na fila), enquanto todas as microtarefas são processadas.

Quando a fila de microtarefa está finalmente vazia, o loop de eventos verifica se uma atualização de renderização da IU é necessária e, se for, a IU é renderizada novamente. Isso termina a iteração atual do loop de eventos, que volta ao início e verifica a fila de macrotarefa novamente.

Agora que temos um entendimento de alto nível do loop de eventos, vamos verificar alguns dos detalhes interessantes mostrados na figura 13.1:

- *Ambas as filas de tarefas são colocadas fora do loop de eventos,* para indicar que o ato de adicionar tarefas às filas correspondentes ocorre fora do loop de eventos. Se não fosse esse o caso, quaisquer eventos que ocorressem enquanto o código JavaScript estivesse sendo executado seriam ignorados. Como definitivamente não queremos fazer isso, os atos de detectar e adicionar tarefas são feitos separadamente do loop de eventos.

- *Ambos os tipos de tarefas são executados um de cada vez,* porque o JavaScript é baseado em um modelo de execução de thread único. Quando uma tarefa começa a ser executada, ela é executada até sua conclusão, sem ser interrompida por outra tarefa. Apenas o navegador pode interromper a execução de uma tarefa; por exemplo, se a tarefa começa a ser muito egoísta por ocupar muito tempo ou memória.

- *Todas as microtarefas devem ser executadas antes da próxima renderização,* porque seu objetivo é atualizar o estado do aplicativo antes de ocorrer a renderização.
- *O navegador geralmente tenta renderizar a página 60 vezes por segundo,* para atingir 60 quadros por segundo (60 fps), uma taxa de quadros geralmente considerada ideal para movimentos suaves, como animações— *ou seja, o navegador tenta renderizar um quadro a cada 16 ms.*
  Observe como a ação “Atualizar renderização”, mostrada na figura 13.1, acontece dentro do loop de eventos, porque o conteúdo da página não deve ser modificado por outra tarefa enquanto a página está sendo renderizada. Isso tudo significa que, se quisermos obter aplicativos de execução suave, não temos muito tempo para processar tarefas em uma única iteração de loop de evento. *Idealmente, uma única tarefa e todas as microtarefas geradas por ela devem ser concluídas em 16 ms.*

Agora, vamos considerar três situações que podem ocorrer na próxima iteração do loop de evento, depois que o navegador tiver concluído uma renderização de página:

- O loop de eventos atinge “É necessário renderizar?” ponto de decisão antes que outros 16 ms tenham decorrido. Como atualizar a IU é uma operação complexa, se não houver uma necessidade explícita de renderizar a página, o navegador pode optar por não realizar a renderização da IU nesta iteração de loop.

- O loop de eventos atinge “É necessário renderizar?” ponto de decisão cerca de 16 ms após a última renderização. Nesse caso, o navegador atualiza a IU e os usuários experimentam um aplicativo de execução suave.
- A execução da próxima tarefa (e todas as microtarefas relacionadas) leva muito mais do que 16 ms. Nesse caso, o navegador não será capaz de renderizar novamente a página na taxa de quadros desejada e a IU não será atualizada. Se a execução do código da tarefa não levar muito tempo (mais do que algumas centenas de milissegundos), esse atraso pode nem ser perceptível, especialmente se não houver muito movimento acontecendo na página. Por outro lado, se demorarmos muito ou se houver animações em execução na página, os usuários provavelmente perceberão a página da Web como lenta e sem resposta. Na pior das hipóteses, em que uma tarefa é executada por mais de alguns segundos, o navegador do usuário mostra a temida mensagem “Script sem resposta”. (Não se preocupe, mais adiante neste capítulo mostraremos a você uma técnica para dividir tarefas complexas em tarefas menores que não obstruem o loop de eventos.)

NOTA Tenha cuidado com os eventos que você decide tratar, com que frequência eles ocorrem e quanto tempo de processamento um tratador de eventos leva. Por exemplo, você deve ser extremamente cuidadoso ao lidar com eventos de movimento do mouse. Mover o mouse faz com que um grande número de eventos sejam enfileirados, portanto, realizar qualquer operação complexa nesse manipulador de movimento do mouse é um caminho certo para a construção de um aplicativo da Web lento e irregular.

Agora que descrevemos como funciona o loop de eventos, você está pronto para explorar alguns exemplos em detalhes.

13.1.1 *Um exemplo com apenas macrotarefas*

O resultado inevitável do modelo de execução de thread único do JavaScript é que apenas uma tarefa pode ser executada por vez. Isso, por sua vez, significa que todas as tarefas criadas precisam esperar em uma fila até que chegue sua vez de execução.

Vamos concentrar nossa atenção em uma página da web simples que contém o seguinte:

- Código JavaScript de linha principal não trivial (global)
- Dois botões e dois manipuladores de clique não triviais, um para cada botão

A lista a seguir mostra o código de amostra.

Listagem 13.1 Pseudocódigo para nossa demonstração de loop de evento com uma fila de tarefas

```
<button id = "firstButton"> </button>
<button id = "secondButton"> </button>
<script>
    const firstButton = document.getElementById ("firstButton"); const secondButton = document.getElementById
     ("secondButton"); firstButton.addEventListener ("click", function firstHandler () {/ * Algum código de identificador de
     clique executado por 8 ms * /

    });
```

Registra um manipulador de eventos para um evento de clique de botão no primeiro botão

```
secondButton.addEventListener ("clique", função secondHandler () {
    / * Clique no código do identificador que é executado por 5ms * /});

/ * Código que funciona por 15ms * / </script>
```

Registra outro manipulador de eventos de clique, desta vez para o segundo botão

Este exemplo requer um pouco de imaginação, então, em vez de bagunçar o fragmento de código com código desnecessário, pedimos que você imagine o seguinte:

- Nosso código JavaScript de linha principal leva 15 ms para ser executado.
- O primeiro manipulador de eventos de clique é executado por 8 ms.
- O segundo manipulador de eventos de clique é executado por 5 ms.

Agora, vamos continuar a ser criativos e dizer que temos um usuário super rápido que clica no primeiro botão 5 ms após o início da execução do nosso script e no segundo botão 12 ms depois. A Figura 13.2 mostra essa situação.

Há muitas informações para digerir aqui, mas entendê-las completamente dará uma ideia melhor de como o loop de eventos funciona. Na parte superior da figura, o tempo

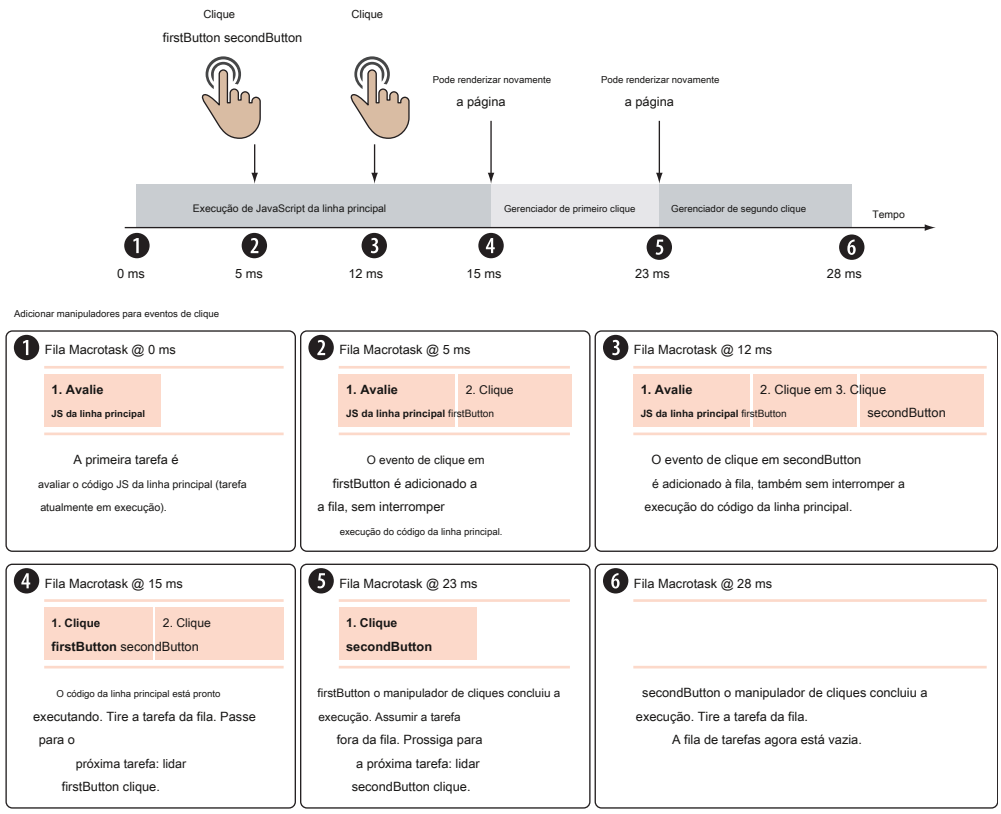


Figura 13.2 Este diagrama de tempo mostra como os eventos são adicionados à fila de tarefas à medida que ocorrem. Quando a execução de uma tarefa termina, o loop de eventos a tira da fila e continua executando a próxima tarefa.

(em milissegundos) é executado da esquerda para a direita ao longo do eixo x. Os retângulos abaixo dessa linha do tempo representam partes do código JavaScript em execução, estendendo-se pelo tempo de execução. Por exemplo, o primeiro bloco de código JavaScript de linha principal é executado por aproximadamente 15 ms, o primeiro manipulador de clique por aproximadamente 8 ms e o segundo manipulador de clique por aproximadamente 5 ms. O diagrama de tempo também mostra quando certos eventos ocorrem; por exemplo, o primeiro clique do botão ocorre 5 ms na execução do aplicativo e o segundo clique no botão 12 ms. A parte inferior da figura mostra o estado da fila de macrotarefa em vários pontos de execução do aplicativo.

O programa começa executando o código JavaScript da linha principal. Imediatamente, dois elementos, firstButton e secondButton, são buscados no DOM e duas funções, firstHandler e secondHandler, são registrados como manipuladores de eventos de clique:

```
firstButton.addEventListener ("clique", função firstHandler () {...}); secondButton.addEventListener ("clique", função secondHandler ()
{...});
```

Isso é seguido por um código que é executado por mais 15 ms. Durante esta execução, nosso usuário clica rapidamente firstButton 5 ms após o programa começar a ser executado e clicar secondButton 12 ms depois.

Como o JavaScript é baseado em um modelo de execução de thread único, clicar firstButton não significa que o manipulador de cliques seja executado imediatamente. (Lembre-se, se uma tarefa já estiver sendo executada, ela não pode ser interrompida por outra tarefa.) Em vez disso, o evento de clique relacionado a firstButton é colocado na fila de tarefas, onde espera pacientemente sua vez de ser executado. A mesma coisa acontece quando um clique de secondButton ocorre: um evento correspondente é colocado na fila de tarefas e aguarda a execução. Observe que é importante que a detecção de eventos e adição à fila de tarefas ocorram fora do loop de eventos; as tarefas são adicionado à fila de tarefas, mesmo enquanto o código JavaScript da linha principal está sendo executado.

Se tirarmos um instantâneo de nossa fila de tarefas 12 ms na execução de nosso script, veremos as três tarefas a seguir:

1. Avalie o código JavaScript da linha principal— a tarefa atualmente em execução. Clique firstButton
2. —O evento criado quando firstButton é clicado. Clique secondButton —O evento criado quando secondButton
3. é clicado.

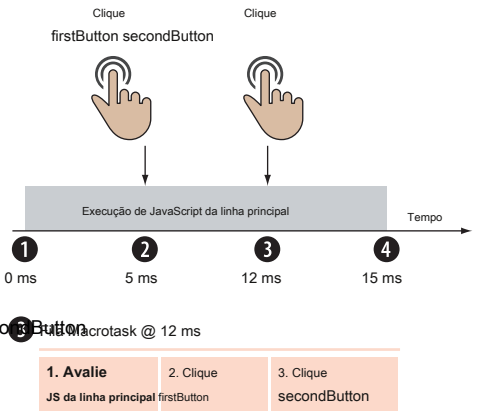


Figura 13.3 12 ms na execução do aplicativo, a fila de tarefas tem três tarefas: uma para avaliar o código JavaScript da linha principal (a tarefa atualmente em execução) e uma para cada evento de clique de botão.

Essas tarefas também são mostradas na figura 13.3.

O próximo ponto interessante na execução do aplicativo acontece aos 15 ms, quando o código JavaScript da linha principal termina seu

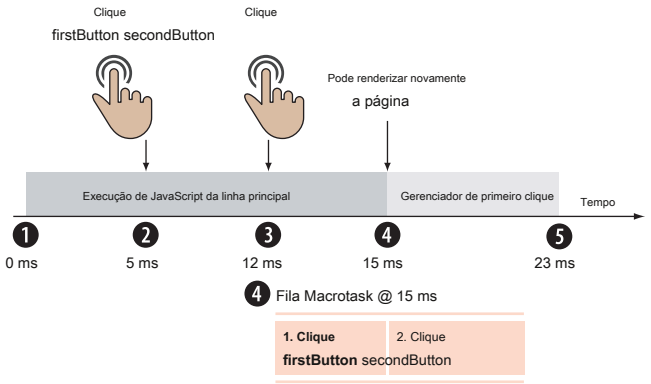


Figura 13.4 A fila de tarefas 15 ms na execução do aplicativo contém duas tarefas para eventos de clique. A primeira tarefa está sendo executada.

execução. Conforme mostrado na figura 13.1, após o término da execução de uma tarefa, o loop de eventos segue para o processamento da fila de microtarefa. Como nessa situação não temos nenhuma microtarefa (nem mesmo mostramos a fila de microtarefas no diagrama, porque ela está sempre vazia), pulamos esta etapa e passamos a atualizar a IU. Neste exemplo, embora a atualização aconteça e leve algum tempo, para simplificar, nós a mantemos fora de nossa discussão. Com isso, o loop de eventos termina a primeira iteração e inicia a segunda iteração, passando para a tarefa seguinte na fila.

A seguir, o firstButton a tarefa click inicia sua execução. A Figura 13.4 ilustra a fila de tarefas 15 ms na execução do aplicativo. A execução de firstHandler, associado com o firstButton clique, leva cerca de 8 ms, e o manipulador é executado até sua conclusão, sem interrupção, enquanto o evento de clique relacionado a secondButton está esperando na fila.

Em seguida, aos 23 ms, o firstButton O evento click é totalmente processado e a tarefa correspondente é removida da fila de tarefas. Novamente, o navegador verifica a fila de microtarefas, que ainda está vazia, e renderiza novamente a página, se necessário.

Finalmente, na terceira iteração do loop, o secondButton o evento click está sendo tratado, conforme mostrado na figura 13.5. O secondHandler leva cerca de 5 ms para ser executado e, depois de executado, a fila de tarefas está finalmente vazia, em 28 ms.

Este exemplo enfatiza que um evento deve aguardar sua vez para ser processado, se outras tarefas já estiverem sendo realizadas. Por exemplo, embora o secondButton o clique aconteceu 12 ms na execução do aplicativo, o manipulador correspondente é chamado em torno de 23 ms na execução do aplicativo.

Agora, vamos estender esse código para incluir microtarefas.

13.1.2 Um exemplo com macro e microtarefas

Agora que você viu como o loop de eventos funciona em uma fila de tarefas, estenderemos nosso exemplo para incluir também uma fila de microtarefa. A maneira mais limpa de fazer isso é incluir uma promessa no primeiro manipulador de clique de botão e o código que trata a promessa depois que ela é resolvida. Como você deve se lembrar do capítulo 6, *uma promessa é um espaço reservado para um valor que ainda não temos, mas que teremos mais tarde;* é uma garantia de que eventualmente saberemos

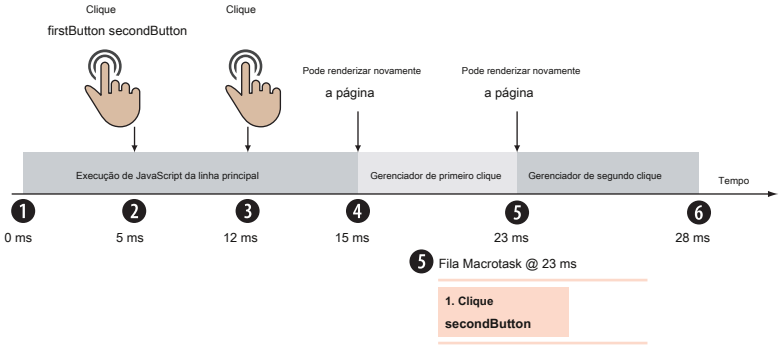


Figura 13.5 23 ms após o início da execução do aplicativo, apenas uma tarefa, tratando do **secondButton** clique no evento, continua a ser executado.

o resultado de uma computação assíncrona. Por esse motivo, manipuladores de promessa, os retornos de chamada que anexamos por meio do então método, são sempre chamados de forma assíncrona, mesmo se os anexarmos a promessas já resolvidas.

A lista a seguir mostra o código modificado para este exemplo de duas filas.

Listagem 13.2 Pseudocódigo para nossa demonstração de loop de evento com duas filas

```
<button id = "firstButton"> </button>
<button id = "secondButton"> </button>
<script>
    const firstButton = document.getElementById ("firstButton"); const secondButton = document.getElementById
    ("secondButton"); firstButton.addEventListener ("click", function firstHandler () {

        Promise.resolve (). Then (() => {
            / * Algum código de tratamento de promessa executado por 4 ms * /});

        / * Alguns códigos de identificador de clique que funcionam por 8 ms * /});

    secondButton.addEventListener ("clique", função secondHandler () {
        / * Clique no código do identificador que é executado por 5ms * /});

/ * Código que funciona por 15ms * / </script>
```

Resolve imediatamente um promessa e passa um retorno de chamada para o método then

Neste exemplo, presumimos que as mesmas ações ocorram como no primeiro exemplo:

- firstButton é clicado após 5 ms.
- secondButton é clicado após 12 ms.
- firstHandler lida com o evento de clique de firstButton e funciona por 8 ms.
- secondHandler lida com o evento de clique de secondButton e funciona por 5 ms.

A única diferença é que desta vez, dentro do firstHandler código, também criamos uma promessa resolvida imediatamente para a qual passamos um retorno de chamada que será executado por 4 ms. Como uma promessa representa um valor futuro que geralmente não sabemos imediatamente, os manipuladores de promessa sempre são tratados de forma assíncrona.

Para ser honesto, neste caso, onde criamos uma promessa resolvida imediatamente, o mecanismo JavaScript poderia invocar imediatamente o retorno de chamada, porque já sabemos que a promessa foi resolvida com sucesso. Mas, para fins de consistência, o mecanismo JavaScript não faz isso e, em vez disso, chama todos os retornos de chamada de promessa de forma assíncrona, após o resto do firstHandler o código (que funciona por 8 ms) termina a execução. Ele faz isso criando uma nova microtarefa e empurrando-a para a fila de microtarefas. Vamos explorar o diagrama de tempo dessa execução na Figura 13.6.

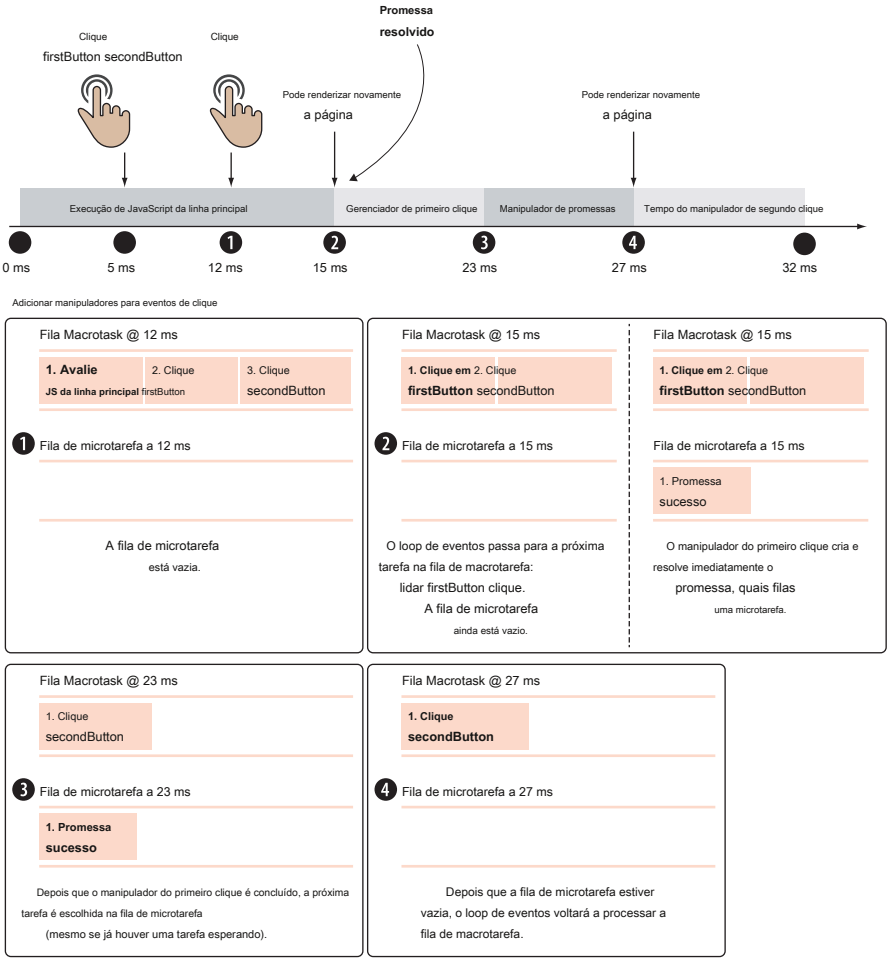


Figura 13.6 Se uma microtarefa estiver enfileirada na fila de microtarefas, ela terá prioridade e será processada mesmo se uma tarefa mais antiga já estiver esperando na fila. Neste caso, a microtarefa de sucesso da promessa tem prioridade sobre o **secondButton** clique na tarefa de evento.

Este diagrama de tempo é semelhante ao diagrama do exemplo anterior. Se tirarmos um instantâneo da fila de tarefas 12 ms na execução do aplicativo, veremos exatamente as mesmas tarefas na fila: O código JavaScript da linha principal está sendo processado enquanto as tarefas para lidar com o firstButton clique e o secondButton click estão aguardando sua vez (como na figura 13.3). Mas, além da fila de tarefas, neste exemplo também vamos nos concentrar na fila de microtarefas, que ainda está vazia 12 ms na execução do aplicativo.

O próximo ponto interessante na execução do aplicativo acontece aos 15 ms, quando a execução do JavaScript da linha principal termina. Como a execução de uma tarefa foi concluída, o loop de eventos verifica a fila de microtarefas, que está vazia, e segue para a renderização da página, se necessário. Novamente, para simplificar, não incluímos um fragmento de renderização em nosso diagrama de tempo.

Na próxima iteração do loop de eventos, a tarefa associada ao firstButton

o clique está sendo processado:

```
firstButton.addEventListener ("click", function firstHandler () {
    Promise.resolve (). Then (() => {
        / * Alguns códigos de manipulação de promessa que são executados por 4ms * /
    });
    / * Algum código de identificador de clique que roda por 8ms * /});
```

O firstHandler função cria uma promessa já resolvida, chamando Promessa

. resolver() com uma função de callback que com certeza será invocada, porque a promessa já foi resolvida. Isso cria uma nova microtarefa para executar o código de retorno de chamada. A microtarefa é colocada na fila de microtarefa e o manipulador de cliques continua a ser executado por mais 8 ms. O estado atual das filas de tarefas é mostrado na figura 13.7.

Revisitamos as filas de tarefas novamente 23 ms na execução do aplicativo, após o

firstButton click foi completamente administrado e sua tarefa retirada da fila de tarefas.

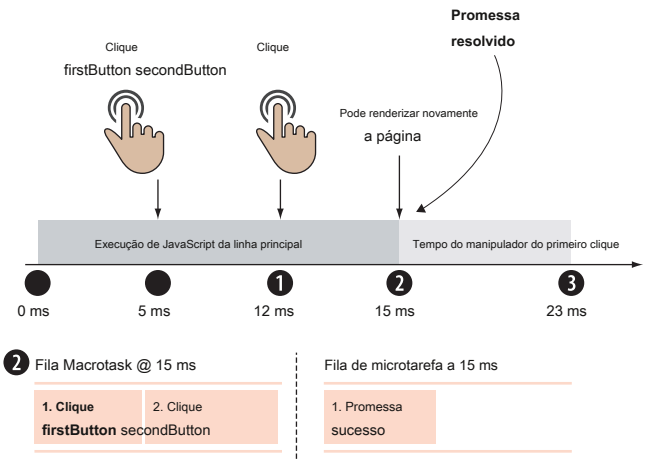


Figura 13.7 Durante a execução
ção do manipulador do primeiro clique, uma promessa resolvida é criada. Isso coloca na fila uma promessa de sucesso de microtarefa nas microtarefas
fila que será executada o mais rápido possível, mas sem interromper a tarefa em execução no momento.

Nesse ponto, o loop de eventos deve escolher qual tarefa processar em seguida. Temos uma macrotarefa para lidar com o secondButton clique que foi colocado na fila de tarefas 12 ms na execução do aplicativo e a tarefa onêmica para lidar com o sucesso da promessa que foi colocada na fila de tarefas em algum lugar em torno de 15 ms na execução do aplicativo.

Se considerarmos coisas assim, seria justo que o secondButton a tarefa de clique é tratada primeiro, mas como já mencionamos, microtarefas são tarefas menores que devem ser executadas o mais rápido possível. As microtarefas têm prioridade e se você olhar para trás na figura 13.1, verá que cada vez que uma tarefa é processada, o loop de eventos verifica primeiro a fila de microtarefas, com o objetivo de processar todas as microtarefas antes de continuar com a renderização ou outras tarefas .

Por este motivo, a tarefa de sucesso da promessa é executada *imediatamente* depois de firstButton clique, mesmo com o "mais antigo" secondButton clique na tarefa ainda aguardando na fila de tarefas, conforme mostrado na figura 13.8.

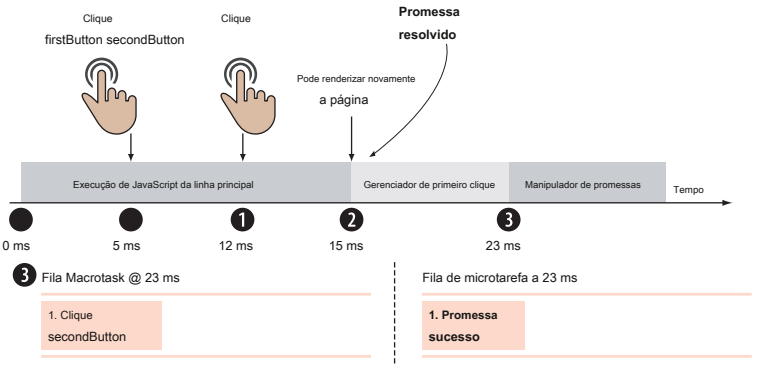


Figura 13.8 Depois que uma tarefa é executada, o loop de eventos processa todas as tarefas na fila de microtarefa. Neste caso, antes de passar para o **secondButton** clique na tarefa, a tarefa de sucesso da promessa é tratada.

Há um ponto importante que precisamos enfatizar. Depois que uma macrotarefa é executada, o loop de eventos imediatamente passa a lidar com a fila de microtarefas, sem permitir a renderização até que a fila de microtarefas esteja vazia. Basta dar uma olhada no diagrama de tempo na figura 13.9.

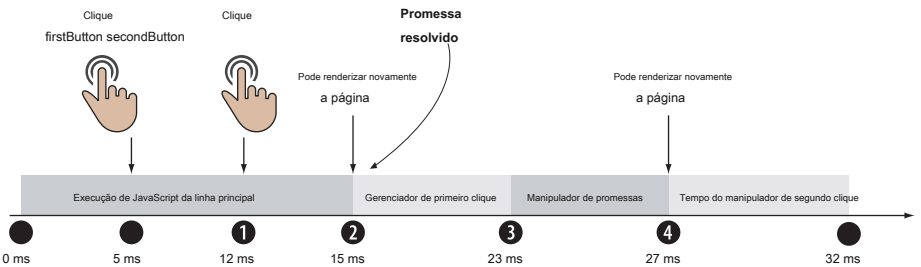


Figura 13.9 Uma página pode ser renderizada novamente entre duas macrotarefas (execução de JavaScript da linha principal e manipulador de primeiro clique), enquanto ***não pode*** ser renderizado antes de uma microtarefa ser executada (antes do manipulador de promessa).

A Figura 13.9 mostra que uma nova renderização pode ocorrer entre duas macrotarefas, apenas se não houver microtarefas entre elas. Em nosso caso, a página pode ser renderizada entre a execução do JavaScript de linha principal e o manipulador do primeiro clique, mas não pode ser renderizada imediatamente após o manipulador do primeiro clique, porque microtarefas, como manipuladores de promessa, têm prioridade sobre a renderização.

Uma renderização também pode ocorrer após uma microtarefa, mas apenas se nenhuma outra microtarefa estiver esperando na fila de microtarefa. Em nosso exemplo, depois que o manipulador de promessa ocorre, mas antes que o loop de eventos passe para o manipulador de segundo clique, o navegador pode renderizar novamente a página.

Observe que não há nada que impeça a promessa de microtarefa de sucesso de enfileirar outras microtarefas, e que todas essas microtarefas terão prioridade sobre o secondButton clique na tarefa. O loop de eventos irá renderizar novamente a página e mover para o secondButton tarefa apenas quando a fila de microtarefa estiver vazia, então tome cuidado!

Agora que você entende como o loop de eventos funciona, vamos dar uma olhada em um grupo especial de eventos: temporizadores.

13.2 Cronômetros de domesticação: tempos limite e intervalos

Timers, um recurso frequentemente mal utilizado e mal compreendido em JavaScript, podem aprimorar o desenvolvimento de aplicativos complexos se usados corretamente. Os cronômetros nos permitem atrasar a execução de um trecho de código por *pelo menos* um certo número de milissegundos. Usaremos esse recurso para dividir tarefas de longa execução em tarefas menores que não obstruirão o loop de eventos, interrompendo a renderização do navegador e, no processo, tornando nossos aplicativos da web lentos e sem resposta.

Mas primeiro, começaremos examinando as funções que podemos usar para construir e manipular temporizadores. O navegador oferece dois métodos para a criação de cronômetros: setTimeout e setInterval. O navegador também fornece dois métodos correspondentes para limpá-los (ou removê-los): clearTimeout e clearInterval. Todos são métodos do janela (contexto global) objeto. De maneira semelhante ao loop de eventos, os temporizadores não são definidos no próprio JavaScript; em vez disso, eles são fornecidos pelo ambiente de host (como o navegador no cliente ou Node.js no servidor). A Tabela 13.1 lista os métodos para criar e limpar temporizadores.

Tabela 13.1 Métodos de manipulação de temporizador do JavaScript (métodos do global **janela** objeto)

Método	Formato	Descrição
setTimeout	id = setTimeout (fn, atraso)	Inicia um cronômetro que executará o retorno de chamada passado uma vez após o atraso especificado ter decorrido. Um valor que identifica exclusivamente o cronômetro é retornado.
clearTimeout	clearTimeout (id)	Cancela (limpa) o cronômetro identificado pelo valor passado se o cronômetro ainda não tiver disparado.

Tabela 13.1 Métodos de manipulação de temporizador do JavaScript (métodos do global **janela** objeto) (***contínuo)***

Método	Formato	Descrição
setInterval	id = setInterval (fn, atraso)	Inicia um temporizador que tentará continuamente executar o retorno de chamada passado no intervalo de atraso especificado, até ser cancelado. Um valor que identifica exclusivamente o cronômetro é retornado.
clearInterval clearInterval	(id)	Cancela (limpa) o temporizador de intervalo identificado pelo valor passado.

Esses métodos nos permitem definir e limpar temporizadores que disparam uma única vez ou disparam periodicamente em um intervalo especificado. Na prática, a maioria dos navegadores permite que você use ambos clearTimeout e clearInterval para cancelar os dois tipos de temporizadores, mas é recomendado que os métodos sejam usados em pares correspondentes, pelo menos para maior clareza.

NOTA É importante entender que *o atraso de um temporizador não é garantido.* Isso tem muito a ver com o loop de eventos, como veremos na próxima seção.

13.2.1 Execução do cronômetro dentro do loop de evento

Você já examinou exatamente o que acontece quando um evento ocorre. Mas os cronômetros são diferentes dos eventos padrão, então vamos explorar um exemplo semelhante aos que você viu até agora. A lista a seguir mostra o código usado para este exemplo.

Listagem 13.3 Pseudocódigo para nosso tempo limite e demonstração de intervalo

```
<button id = "myButton"> </button>
<script>
    setTimeout (function timeoutHandler () {
        / * Algum código de controle de tempo limite que é executado por 6 ms * /}, 10);

    setInterval (function intervalHandler () {
        / * Algum código de manipulação de intervalo que é executado por 8 ms * /}, 10);

    const myButton = document.getElementById ("myButton"); myButton.addEventListener ("click", function
    clickHandler () {
            / * Alguns códigos de identificador de clique que são executados por 10 ms * /
    });

    / * Código que funciona por 18 ms * / </script>
```

Registra um tempo limite que expira após 10 ms

Registra um intervalo que expira a cada 10 ms

Registra um evento manipulador para um evento de clique de botão

Desta vez, temos apenas um botão, mas também registramos dois temporizadores. Primeiro, registramos um tempo limite que expira após 10 ms:

```
setTimeout (function timeoutHandler () {
    / * Algum código de manipulador de tempo limite que é executado por 6 ms * /}, 10);
```

Como manipulador, esse tempo limite tem uma função que leva 6 ms para ser executada. Em seguida, também registramos um intervalo que expira a cada 10 ms:

```
setInterval (function intervalHandler () {
    / * Algum código de manipulador de intervalo que é executado por 8ms * /}, 10);
```

O intervalo possui um manipulador que leva 8 ms para ser executado. Continuamos registrando um manipulador de eventos de clique de botão que leva 10 ms para ser executado:

```
const myButton = document.getElementById ("myButton"); myButton.addEventListener ("click", function
clickHandler () {
      / * Algum código de manipulador de cliques que é executado por 10 ms * /
});
```

Este exemplo termina com um bloco de código que é executado por cerca de 18 ms (novamente, faça um pouco de humor e imagine um código complexo aqui).

Agora, digamos que temos novamente um usuário rápido e impaciente que clica no botão 6 ms na execução do aplicativo. A Figura 13.10 mostra um diagrama de tempo dos primeiros 18 ms de execução.

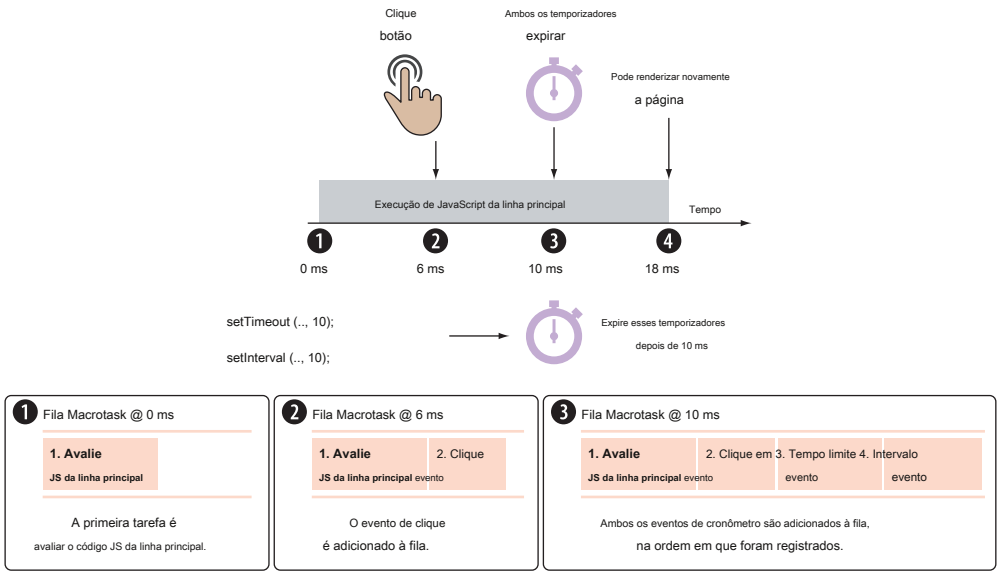


Figura 13.10 Um diagrama de tempo que mostra 18ms de execução no programa de exemplo. A primeira tarefa atualmente em execução é avaliar o código JavaScript da linha principal. Demora 18 ms para executar. Durante essa execução, três eventos ocorrem: um clique do mouse, uma expiração do cronômetro e um disparo de intervalo.

Como nos exemplos anteriores, a primeira tarefa na fila é executar o código JavaScript da linha principal. Durante essa execução, que leva cerca de 18 ms para ser concluída, três coisas importantes ocorrem:

1 A 0 ms, um temporizador de tempo limite é iniciado com um atraso de 10 ms e um temporizador de intervalo também é iniciado com um atraso de 10 ms. Suas referências são mantidas pelo navegador. Em 6 ms, o mouse é clicado.

2

3 Em 10 ms, o temporizador de tempo limite expira e o primeiro intervalo é disparado.

Como já sabemos por nossa exploração do loop de eventos, uma tarefa sempre é executada até a conclusão e não pode ser interrompida por outra tarefa. Em vez disso, todas as tarefas recém-criadas são colocadas em uma fila, onde pacientemente aguardam sua vez de serem processadas. Quando o usuário clica no botão 6 ms na execução do aplicativo, essa tarefa é adicionada à fila de tarefas. Algo semelhante acontece por volta dos 10 ms, quando o cronômetro expira e o intervalo é disparado. Eventos de cronômetro, assim como eventos de entrada (como eventos de mouse), são colocados na fila de tarefas. Observe que tanto o cronômetro quanto o intervalo são iniciados com um atraso de 10 ms, e que após este período, suas tarefas correspondentes são

colocado na fila de tarefas. Voltaremos a isso mais tarde, mas por agora é suficiente que você observe que as tarefas são adicionadas à fila na ordem em que os manipuladores são registrados: primeiro o manipulador de tempo limite e, em seguida, o manipulador de intervalo.

O bloco inicial de código conclui a execução após 18 ms, e como não há microtarefas nesta execução, o navegador pode renderizar novamente a página (novamente, deixada de fora de nossas discussões de tempo, devido à simplicidade) e passar para a segunda iteração de o loop de eventos. O estado da fila de tarefas neste momento é mostrado na figura 13.11.

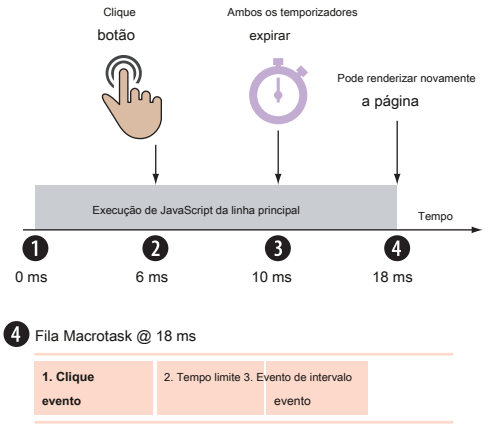


Figura 13.11 Os eventos de cronômetro são colocados na fila de tarefas à medida que expiram.

Quando o bloco inicial de código termina a execução em 18 ms, três blocos de código são enfileirados para execução: o manipulador de clique, o manipulador de tempo limite e a primeira chamada do manipulador de intervalo. Isso significa que o manipulador de cliques em espera (que presumimos que leva 10 ms para ser executado) começa a execução. A Figura 13.12 mostra outro diagrama de tempo.

Ao contrário do setTimeout função, que expira apenas uma vez, o setInterval A função é disparada até que a limpemos explicitamente. Então, por volta de 20 ms, outro intervalo é disparado. Normalmente, isso criaria uma nova tarefa e a adicionaria à fila de tarefas. Mas desta vez, como uma instância de uma tarefa de intervalo já está enfileirada e aguardando execução, essa chamada é descartada. *O navegador não enfileira mais de uma instância de um manipulador de intervalo específico.*

O manipulador de cliques é concluído em 28 ms, e o navegador pode novamente renderizar a página antes que o loop de eventos entre em outra iteração. Na próxima iteração do loop de evento, a 28 ms, a tarefa de tempo limite é processada. Mas pense no início de

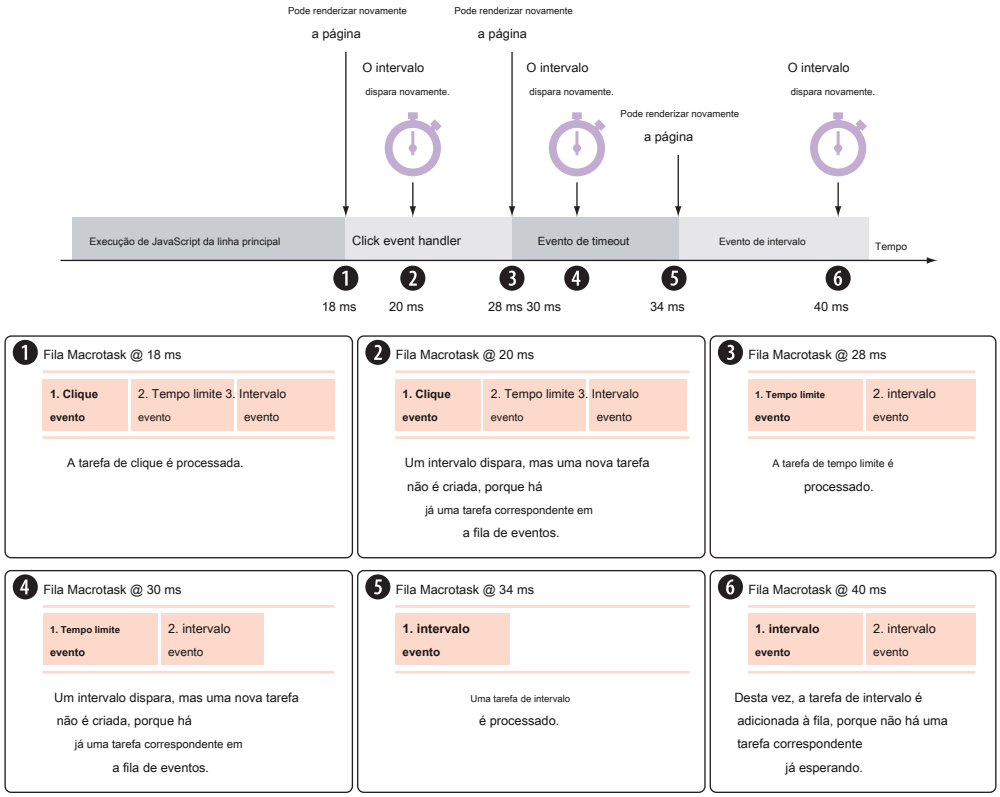


Figura 13.12 Se um evento de intervalo for disparado e uma tarefa já estiver associada a esse intervalo de espera na fila, uma nova tarefa não será adicionada. Em vez disso, nada acontece, como é mostrado para as filas de 20 ms e 30 ms.

este exemplo. Usamos a seguinte chamada de função para definir um tempo limite que deve expirar após 10 ms:

```
setTimeout (function timeoutHandler () {
    / * Algum código de controle de tempo limite que é executado por 6 ms * /}, 10);
```

Como esta é a primeira tarefa em nosso aplicativo, não é estranho esperar que o manipulador de tempo limite seja executado exatamente após 10 ms. Mas, como você pode ver na figura 13.11, o tempo limite começa na marca de 28 ms!

É por isso que fomos extremamente cuidadosos ao dizer que um cronômetro fornece a capacidade de atrasar de forma assíncrona a execução de um trecho de código por *pelo menos* um certo número de milissegundos. Devido à natureza de thread único do JavaScript, podemos controlar apenas quando a tarefa do cronômetro é adicionada à fila, e não quando é finalmente executada! Agora que esclarecemos esse pequeno enigma, vamos continuar com o restante da execução do aplicativo.

A tarefa de time-out leva 6 ms para ser executada, portanto, deve terminar 34 ms na execução do aplicativo. Durante esse período, a 30 ms, outro intervalo é disparado, porque agendamos sua execução a cada 10 ms. Mais uma vez, nenhuma tarefa adicional é enfileirada, porque uma tarefa correspondente para a execução do manipulador de intervalo já está esperando na fila. Em 34 ms, o manipulador de tempo limite termina e o navegador novamente tem a chance de renderizar novamente a página e entrar em outra iteração do loop de eventos.

Finalmente, o manipulador de intervalo começa sua execução em 34 ms, 24 ms *depois de* a marca de 10 ms em que foi adicionado à fila de eventos. Isso novamente enfatiza que o atraso que passamos
em como um argumento para as funções setTimeout (fn, atraso) e setInterval (fn,
atraso) especifica apenas o atraso após o qual a tarefa correspondente é adicionada à fila, e não o tempo exato de execução.

O manipulador de intervalo leva 8 ms para ser executado, portanto, enquanto está em execução, outro intervalo expira na marca de 40 ms. Desta vez, como o gerenciador de intervalo está sendo executado (e não esperando na fila), uma nova tarefa de intervalo é finalmente adicionada à fila de tarefas e a execução de nosso aplicativo continua, conforme mostrado na figura 13.13. Configurando um
setInterval o atraso de 10 ms não significa que terminaremos com nosso manipulador executando a cada 10 ms. Por exemplo, como as tarefas são enfileiradas e a duração da execução de uma única tarefa pode variar, os intervalos podem ser executados um após o outro, como é o caso com intervalos nas marcas de 42 e 50 ms.

Finalmente, após 50 ms, nossos intervalos se estabilizam e são executados a cada 10 ms. O conceito importante a ser aprendido é que o loop de eventos pode processar apenas uma tarefa por vez e que nunca podemos ter certeza de que os manipuladores de cronômetro serão executados exatamente quando esperamos. Isso é especialmente verdadeiro para manipuladores de intervalo. Vimos neste exemplo que, embora tenhamos programado um intervalo esperado para disparar em marcas de 10, 20, 30, 40, 50, 60 e 70 ms, os retornos de chamada foram executados em marcas de 34, 42, 50, 60 e 70 ms . Neste caso, perdemos completamente dois deles ao longo do caminho, e alguns não foram executados no tempo esperado.

Como podemos ver, os intervalos têm considerações especiais que não se aplicam a tempos limite. Vamos examiná-los mais de perto.

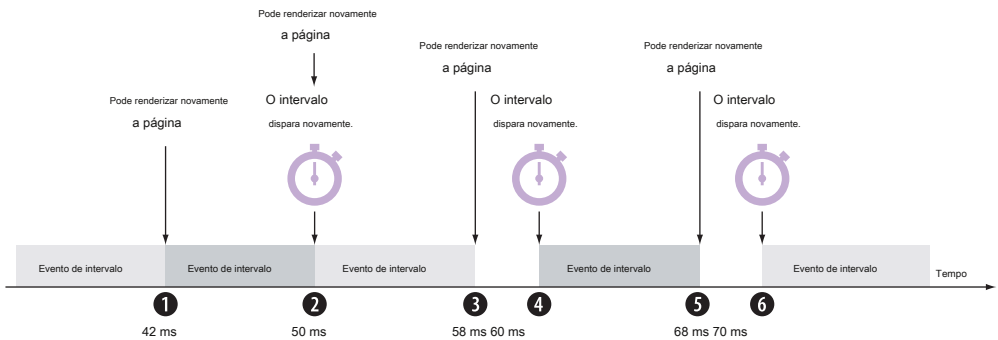


Figura 13.13 Por causa dos contratempos causados pelo clique do mouse e pelo manipulador de tempo limite, leva algum tempo para que os manipuladores de intervalo comecem a ser executados a cada 10 ms.

DIFERENÇAS ENTRE TEMPO LIMITADO E INTERVALOS

À primeira vista, um intervalo pode parecer um tempo limite que se repete periodicamente. Mas as diferenças são um pouco mais profundas. Vejamos um exemplo para ilustrar melhor a diferença
ferências entre setTimeout e setInterval:

```
setTimeout (function repeatMe () {
    / * Algum bloco longo de código ... * / setTimeout (repeatMe,
    10);
}, 10);
setInterval (() => {
    / * Algum bloco longo de código ... * /}, 10);
```

Define um tempo limite que se reprograma a cada 10 milissegundos

Define um intervalo que dispara a cada 10 milissegundos

As duas partes do código podem *aparecer* para ser funcionalmente equivalente, mas não são. Notavelmente, o setTimeout variante do código sempre terá um atraso de pelo menos 10 ms após a execução do retorno de chamada anterior (dependendo do estado da fila de eventos, pode acabar sendo mais, mas nunca menos), enquanto setInterval tentará executar um retorno de chamada a cada 10 ms, independentemente de quando o último retorno de chamada foi executado. E, como você viu no exemplo da seção anterior, os intervalos podem ser disparados imediatamente um após o outro, independentemente do atraso.

Sabemos que o retorno de chamada de tempo limite nunca tem garantia de execução exata quando disparado. Em vez de ser disparado a cada 10 ms, conforme o intervalo, ele se reprogramará por 10 ms depois de começar a ser executado.

Tudo isso é um conhecimento extremamente importante. Saber como um mecanismo JavaScript lida com o código assíncrono, especialmente com o grande número de eventos assíncronos que normalmente ocorrem na página média, cria uma grande base para a construção de partes avançadas de código de aplicativo.

Com tudo isso sob controle, vamos ver como nossa compreensão dos temporizadores e do loop de eventos pode ajudar a evitar algumas armadilhas de desempenho.

13.2.2 Lidar com processamento computacionalmente caro

A natureza de thread único do JavaScript é provavelmente a maior pegadinha no desenvolvimento de aplicativos JavaScript complexos. Enquanto o JavaScript está ocupado em execução, a interação do usuário no navegador pode se tornar, na melhor das hipóteses, lenta e, na pior, sem resposta. O navegador pode engasgar ou parecer travar, porque todas as atualizações na renderização de uma página são suspensas enquanto o JavaScript está sendo executado.

Reduzir todas as operações complexas que levam mais do que algumas centenas de milissegundos em porções gerenciáveis se torna uma necessidade se quisermos manter a interface responsiva. Além disso, a maioria dos navegadores produzirá uma caixa de diálogo avisando ao usuário que um script ficou “sem resposta” se tiver sido executado sem interrupção por pelo menos 5 segundos, enquanto alguns outros navegadores irão até mesmo matar silenciosamente qualquer script em execução por mais de 5 segundos.

Você pode ter participado de uma reunião de família em que um tio tagarela não para de falar e insiste em contar as mesmas histórias indefinidamente. Se ninguém mais tiver a chance de interromper e dizer uma palavra, a conversa não será agradável para

ninguém (exceto o tio Bruce). Da mesma forma, o código que consome todo o tempo de processamento resulta em um resultado inferior ao desejável; produzir uma interface de usuário que não responde nunca é bom. Mas quase certamente surgirão situações que exigem que processemos uma quantidade significativa de dados, situações como a manipulação de alguns milhares de elementos DOM, por exemplo.

Nessas ocasiões, os cronômetros podem ajudar e se tornar especialmente úteis. Como os temporizadores são capazes de suspender efetivamente a execução de um trecho de JavaScript até um momento posterior, eles também podem quebrar trechos individuais de código em fragmentos que não são longos o suficiente para fazer o navegador travar. Levando isso em consideração, podemos converter loops e operações intensivas em operações não bloqueadoras.

Vejamos o seguinte exemplo de uma tarefa que provavelmente levará muito tempo.

Listagem 13.4 Uma tarefa de longa duração

Rende 20.000 linhas, que deve se qualificar como uma "carga de barco"

Encontra o corpo elemento que estava indo para criar um barco carregado de linhas para

```
<table><tbody></tbody> </table>
<script>
    const tbody = document.querySelector ("tbody"); para (seja i = 0; i <20000; i ++) {
        const tr = document.createElement ("tr"); para (deixe t = 0; t <6; t ++) {
            const td = document.createElement ("td");
            td.appendChild (document.createTextNode (i + "," + t)); tr.appendChild (td);
        }
        tbody.appendChild (tr);
    }
</script>
```

Cria um linha individual

Para cada linha, cria 6 células, cada um com um nó de texto

Anexa o novo linha para seu pai

Neste exemplo, estamos criando um total de 240.000 nós DOM, preenchendo uma tabela com 20.000 linhas de 6 células, cada uma contendo um nó de texto. Isso é incrivelmente caro e provavelmente irá travar o navegador por um período perceptível durante a execução, impedindo o usuário de realizar interações normais (da mesma forma que o tio Bruce domina a conversa na reunião familiar).

O que precisamos fazer é calar o tio Bruce em intervalos regulares para que outras pessoas possam ter a chance de entrar na conversa. No código, podemos introduzir temporizadores para criar apenas essas "interrupções na conversa", conforme mostrado na próxima listagem.

Listagem 13.5 Usando um cronômetro para interromper uma tarefa de longa execução

```
const rowCount = 20000; const divideInto =
4;
const chunkSize = rowCount / divideInto; deixe iteração = 0;

const table = document.getElementsByTagName ("tbody") [0]; setTimeout (function generateRows () {

    const base = chunkSize * iteração;
```

Configura os dados

Calcula onde paramos da última vez

```
    para (deixe i = 0; i <chunkSize; i ++) {
        const tr = document.createElement ("tr"); para (deixe t = 0; t <6; t ++) {

            const td = document.createElement ("td"); td.appendChild (

                document.createTextNode ((i + base) + "," + t +
                                                    "," + iteração));
            tr.appendChild (td);
        }
        table.appendChild (tr);
    }
    iteração ++;
    if (iteração <divideInto)
        setTimeout (generateRows, 0);
}, 0);
```

Horários nas próximas fase

Define o atraso de tempo limite para 0 para indicar que a próxima iteração deve ser executada "assim que possível", mas depois que a IU foi atualizada

Nessa modificação do exemplo, dividimos a longa operação em quatro operações menores, cada uma criando seu próprio compartilhamento de nós DOM. Essas operações menores têm muito menos probabilidade de interromper o fluxo do navegador, conforme mostrado na figura 13.14. Observe que nós o configuramos para que os valores dos dados que controlam a operação sejam coletados em variáveis facilmente ajustáveis (rowCount, divideInto, e tamanho do pedaço), se precisarmos dividir as operações em, digamos, dez partes em vez de quatro.

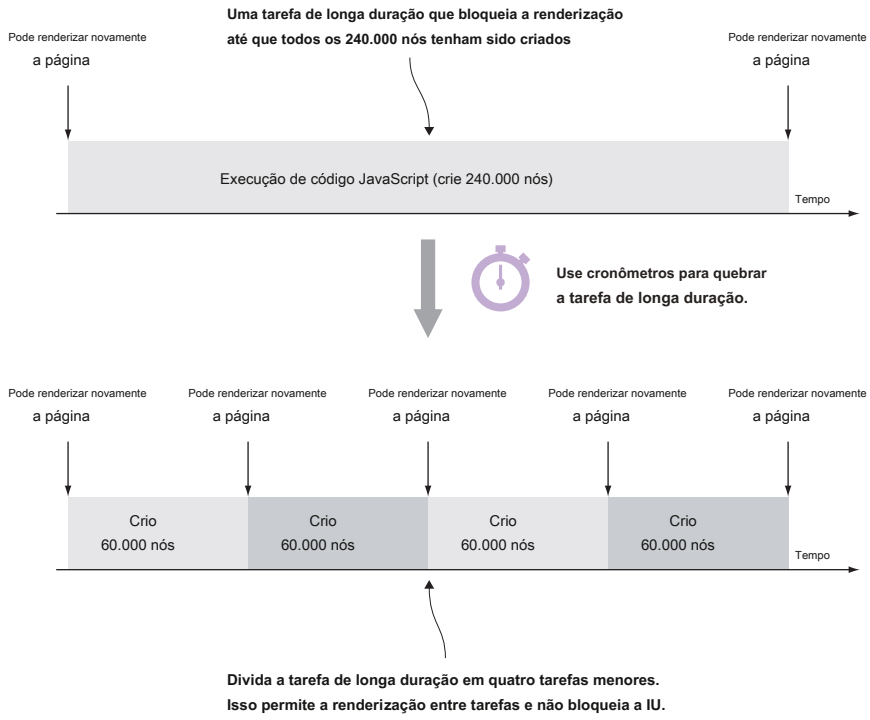


Figura 13.14 Use temporizadores para dividir tarefas de longa duração em tarefas menores que não obstruam o loop de eventos.

Também é importante observar o pouco de matemática necessária para manter o controle de onde paramos na iteração anterior, base = chunkSize * iteration, e como agendamos automaticamente as próximas iterações até determinarmos que terminamos:

```
if (iteração <divideInto)
        setTimeout (generateRows, 0);
```

O que é impressionante é quão pouco o código precisa mudar para acomodar essa nova abordagem assíncrona. Temos que fazer um *pouco* mais trabalho para acompanhar o que está acontecendo, para garantir que a operação seja conduzida corretamente e para programar as peças de execução. Mas, além disso, o núcleo do código é semelhante ao que começamos.

NOTA Neste caso, usamos 0 para o nosso atraso de tempo limite. Se você prestou muita atenção em como funciona o loop de eventos, sabe que isso não significa que o retorno de chamada será executado em 0 ms. Em vez disso, é uma forma de dizer ao navegador, execute este retorno de chamada o mais rápido possível; mas, ao contrário das microtarefas, você tem permissão para fazer a renderização da página no meio. Isso permite que o navegador atualize a IU e torne nossos aplicativos da web mais responsivos.

A mudança mais perceptível resultante dessa técnica, da perspectiva do usuário, é que um longo travamento do navegador é substituído por quatro (ou quantas quisermos) atualizações visuais da página. Embora o navegador tente executar os segmentos de código o mais rápido possível, ele também renderizará as alterações do DOM após cada etapa do cronômetro. Na versão original do código, ele precisava esperar por uma grande atualização em massa.

Na maior parte do tempo, esses tipos de atualizações são imperceptíveis para o usuário, mas é importante lembrar que eles ocorrem. Devemos nos esforçar para garantir que qualquer código que introduzirmos na página não interrompa perceptivelmente a operação normal do navegador.

Muitas vezes é surpreendente como essa técnica pode ser útil. Ao compreender como o loop de eventos funciona, podemos contornar as restrições do ambiente do navegador de thread único, ao mesmo tempo que oferecemos uma experiência agradável ao usuário.

Agora que entendemos o loop de eventos e as funções que os temporizadores podem desempenhar ao lidar com operações complexas, vamos dar uma olhada mais de perto em como os próprios eventos funcionam.

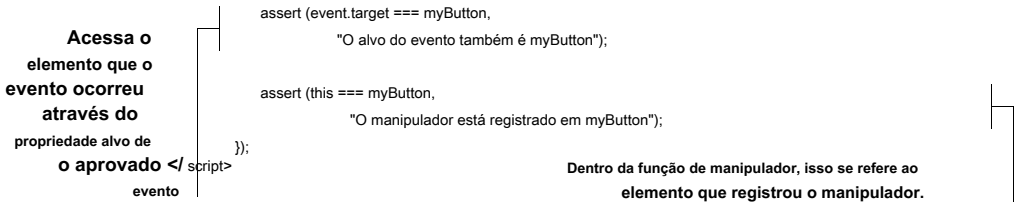
13.3 Trabalho com eventos

Quando um determinado evento ocorre, podemos tratá-lo em nosso código. Como você viu muitas vezes ao longo deste livro, uma maneira comum de registrar manipuladores de eventos é usando o sistema integrado addEventListener método, conforme mostrado na lista a seguir.

Listagem 13.6 Registrando manipuladores de eventos

```
<button id = "myButton"> Clique </button>
<script>
    const myButton = document.getElementById ("myButton"); myButton.addEventListener ("click", function myHandler
    (event) {
```

Registra um manipulador de eventos usando o método addEventListener



Neste snippet, definimos um botão chamado myButton e registrar um manipulador de eventos de clique usando o built-in addEventListener método que é acessível a partir de todos os elementos.

Depois que um evento de clique ocorre, o navegador chama o manipulador associado, neste caso o myHandler função. Para esse manipulador, o navegador passa um objeto de evento que contém propriedades que podemos usar para descobrir mais informações sobre o evento, como a posição do mouse ou o botão do mouse que foi clicado, caso se trate de um evento de clique do mouse ou a tecla pressionada se estivermos lidando com um evento de teclado.

Uma das propriedades do objeto de evento passado é o alvo propriedade, que faz referência ao elemento no qual o evento ocorreu.

NOTA Como na maioria das outras funções, no manipulador de eventos, podemos usar o esta palavra-chave. As pessoas costumam dizer coloquialmente que, em um manipulador de eventos, o esta palavra-chave se refere ao objeto no qual o evento ocorreu, mas como logo descobriremos, isso não é exatamente verdade. Em vez disso, o esta palavra-chave refere-se ao elemento no qual o manipulador de eventos foi *registrado.* Para ser honesto, na maioria dos casos, o elemento no qual o manipulador de eventos foi registrado *é* o elemento no qual o evento ocorreu, mas há exceções. Exploraremos essas situações em breve.

Antes de explorar mais esse conceito, vamos definir o cenário para que você possa ver como os eventos podem ser propagados por meio do DOM.

13.3.1 Propagar eventos através do DOM

Como já sabemos no capítulo 2, em documentos HTML, os elementos são organizados em uma árvore. Um elemento pode ter zero ou mais filhos, e todos os elementos (exceto a raiz html elemento) tem exatamente um pai. Agora, suponha que estejamos trabalhando com uma página que possui um elemento dentro de outro elemento, e ambos os elementos possuem um manipulador de cliques, como na listagem a seguir.

Listagem 13.7 Elementos aninhados e manipuladores de clique

```
<html>
    <head>
        <style>
            # outerContainer {width: 100px; altura: 100px; cor de fundo: azul;}
            # innerContainer {width: 50px; altura: 50px; cor de fundo: vermelho;} </style>

    </head>
    <body>
```

```
        <div id = "outerContainer">                                          Cria dois
            <div id = "innerContainer"> </div>                               elementos aninhados
        </div>
        <script>
            const outerContainer = document.getElementById ("outerContainer"); const innerContainer = document.getElementById
            ("innerContainer");

            outerContainer.addEventListener ("click", () => {                Registra um manipulador de
                relatório ("clique do recipiente externo");                  cliques para o contêiner externo
            });

            innerContainer.addEventListener ("click", () => {                Registra um manipulador de
                relatório ("Clique no recipiente interno");                  cliques para o contêiner interno
            });

            document.addEventListener ("click", () => {                      Registra um manipulador de
                relatório ("clique no documento");                           cliques para todo o documento
            });
        </script>
    </body>
</html>
```

Aqui temos dois elementos HTML, outerContainer e innerContainer, que são, como todos os outros elementos HTML, contidos em nosso global documento. E em todos os três objetos, registramos um manipulador de cliques.

Agora vamos supor que um usuário clique no innerContainer elemento. Porque innerContainer está contido no outerContainer elemento, e ambos os elementos estão contidos no documento, é óbvio que isso deve acionar a execução de todos os três manipuladores de eventos, gerando três mensagens. O que não está aparente é a ordem em que os manipuladores de eventos devem ser executados.

Devemos seguir a ordem em que os eventos foram registrados? Devemos começar com o elemento no qual o evento ocorre e mover para cima? Ou devemos começar do topo e descer em direção ao elemento-alvo? No passado, quando os navegadores estavam tomando essas decisões pela primeira vez, os dois principais concorrentes, Netscape e Microsoft, fizeram escolhas opostas.

No modelo de evento do Netscape, a manipulação de eventos começa com o elemento superior e desce até o elemento de destino do evento. Em nosso caso, os manipuladores de eventos seriam executados na seguinte ordem: documento manipulador de cliques, outerContainer manipulador de clique e finalmente innerContainer clique no manipulador. Isso é chamado *captura de eventos.*

A Microsoft optou por fazer o contrário: comece a partir do elemento-alvo e borbulhe na árvore DOM. Em nosso caso, os eventos seriam executados na seguinte

ordem: innerContainer manipulador de cliques, outerContainer manipulador de clique e documento

clique no manipulador. Isso é chamado *borbulhamento do evento.*

O padrão definido pelo Consórcio W3 (www.w3.org/TR/DOM-Level-3-Events/) ,

que é implementado por todos os navegadores modernos, abraça ambas as abordagens. Um evento é tratado em duas fases:

1. *Fase de captura* —Um evento é primeiro capturado no elemento superior e arrastado para o elemento de destino.
2. *Fase de bolhas* —Depois que o elemento de destino é alcançado na fase de captura, o tratamento de eventos muda para bubbling e o evento borbulha novamente do elemento de destino para o elemento superior.

Essas duas fases são mostradas na figura 13.15.

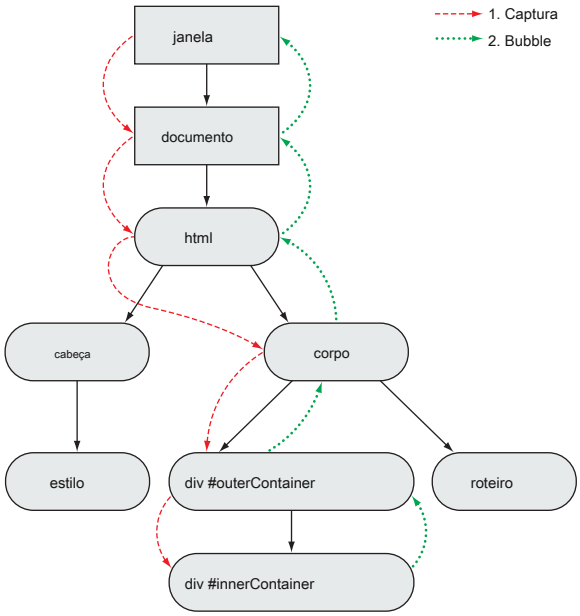


Figura 13.15 Com a captura, o
o evento desce até o elemento de destino. Com o borbulhamento, o evento borbulha do elemento de destino.

Podemos facilmente decidir qual ordem de manipulação de eventos queremos usar, adicionando outro argumento booleano ao addEventListener método. Se usarmos verdade como o terceiro argumento, o evento será capturado, enquanto se usarmos falso (ou omita o valor), o evento irá borbulhar. Então, de certa forma, o padrão W3C prefere o bubbling de evento um pouco mais do que a captura de evento, porque o bubbling se tornou a opção padrão.

Agora, vamos voltar à listagem 13.7 e olhar atentamente como registramos os eventos:

```
outerContainer.addEventListener ("click", () => {
    relatório ("clique do recipiente externo");
});

innerContainer.addEventListener ("click", () => {
    relatório ("Clique no recipiente interno");
});

document.addEventListener ("click", () => {
    relatório ("clique no documento");
});
```

Como você pode ver, em todos os três casos, chamamos o addEventListener método com apenas dois argumentos, o que significa que o método padrão, *borbulhando,* é escolhido. Então, neste caso, se clicarmos no innerContainer , os manipuladores de eventos seriam executados nesta ordem: innerContainer manipulador de cliques, outerContainer manipulador de cliques,

documento clique no manipulador.

Vamos modificar o código na Listagem 13.7 da seguinte maneira.

Listagem 13.8 Captura versus borbulhamento

```
const outerContainer = document.getElementById ("outerContainer"); const innerContainer = document.getElementById
("innerContainer");

document.addEventListener ("click", () => {
    relatório ("clique no documento");
});

outerContainer.addEventListener ("click", () => {
    relatório ("clique do recipiente externo");
}, verdade);

innerContainer.addEventListener ("click", () => {
    relatório ("Clique no recipiente interno");
}, falso);
```

Ao não especificar o terceiro argumento, o modo padrão, bubbling, é ativado.

Passar verdadeiro como terceiro argumento permite a captura.

Passando em falso permite borbulhar.

Desta vez, definimos o manipulador de eventos do outerContainer para o modo de captura (passando verdade como o terceiro argumento), e os manipuladores de eventos de innerContainer
(passando em falso como o terceiro argumento) e documento para o modo bubbling (deixando de fora o terceiro argumento escolhe o modo padrão, bubbling).

Como você sabe, um único evento pode acionar a execução de vários manipuladores de eventos, onde cada manipulador pode estar no modo de captura ou bolha. Por esse motivo, o evento primeiro passa pela captura, começando do elemento superior e descendo até o elemento de destino do evento. Quando o elemento de destino é alcançado, o modo de bolhas é ativado e o evento borbulha do elemento de destino de volta ao topo.

Em nosso caso, a captura começa do topo, janela objeto e goteja até o
innerContainer , com o objetivo de encontrar todos os elementos que possuem um manipulador de eventos para este evento de clique no modo de captura. Apenas um elemento, outerContainer, é encontrado e seu manipulador de cliques correspondente é executado como o primeiro manipulador de eventos.

O evento continua viajando pelo caminho de captura, mas nenhum manipulador de eventos com captura é encontrado. Depois que o evento atinge o elemento de destino do evento, o
innerContainer elemento, o evento segue para a fase de bolhas, onde vai do elemento de destino de volta ao topo, executando todos os manipuladores de eventos de bolhas nesse caminho.

No nosso caso, o innerContainer o manipulador de cliques será executado como o segundo manipulador de eventos, e o documento clique no manipulador como o terceiro. A saída gerada clicando no innerContainer elemento, bem como o caminho percorrido, é mostrado na figura 13.16.

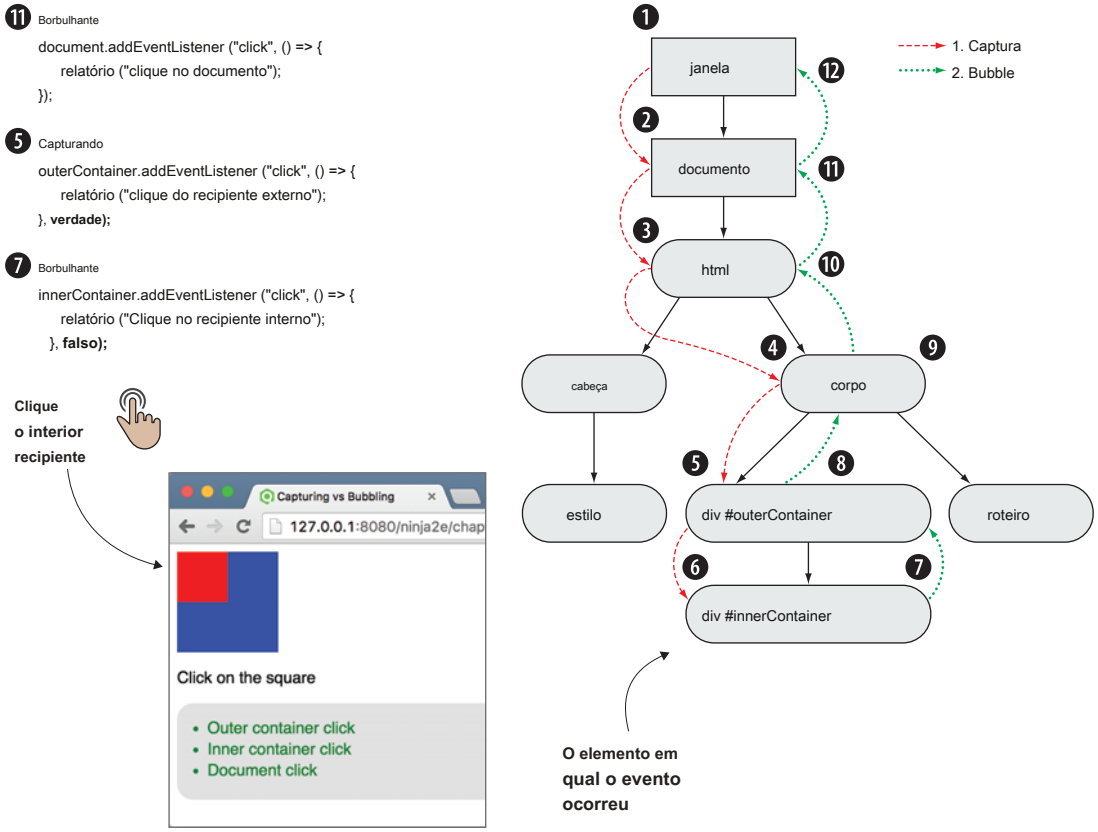


Figura 13.16 Primeiro, o evento desce do topo, executando todos os manipuladores de eventos no modo de captura. Quando o elemento de destino é alcançado, o evento borbulha no topo, executando todos os manipuladores de eventos no modo de bolhas.

Uma das coisas que este exemplo mostra é que o elemento no qual o evento é tratado não precisa ser o elemento no qual o evento ocorre. Por exemplo, em nosso caso, o evento ocorre no innerContainer elemento, mas podemos tratá-lo em elementos superiores na hierarquia DOM, como no outerContainer ou o documento elemento.

Isso nos leva de volta ao esta palavra-chave em manipuladores de eventos e por que afirmamos explicitamente que o esta palavra-chave refere-se ao elemento no qual o manipulador de eventos está registrado, e não necessariamente ao elemento no qual o evento ocorre.

Novamente, vamos modificar nosso exemplo de execução, conforme mostrado na lista a seguir.

Listagem 13.9 A diferença entre esta e event.target em manipuladores de eventos

```
const outerContainer = document.getElementById ("outerContainer"); const innerContainer = document.getElementById
("innerContainer");

innerContainer.addEventListener ("click", function (event) {
```

```
    report ("manipulador innerContainer");
    assert (this === innerContainer,
                "Isso se refere ao innerContainer"); assert (event.target ===
    innerContainer,
                "event.target refere-se ao innerContainer");
});

outerContainer.addEventListener ("click", function (event) {
    report ("manipulador innerContainer");
    assert (this === outerContainer,
                "Isso se refere ao outerContainer"); assert (event.target ===
    innerContainer,
                "event.target refere-se ao innerContainer");
});
```

Dentro do innerContainer handler, this e event.target apontam para o elemento innerContainer.

No manipulador outerContainer, se estivermos manipulando o evento originado no innerContainer, isso fará referência ao outerContainer e event.target ao innerContainer.

Novamente, vamos olhar para a execução do aplicativo quando ocorre um clique em innerContainer.

Porque ambos os manipuladores de eventos usam bolhas de eventos (não há um terceiro argumento definido para verdade no addEventListener métodos), primeiro o innerContainer o manipulador de cliques é invocado. Dentro do corpo do manipulador, verificamos se o esta palavra-chave e o event.target propriedade refere-se ao innerContainer elemento:

```
    assert (this === innerContainer,
                "Isso se refere ao innerContainer"); assert (event.target ===
    innerContainer,
                "event.target refere-se ao innerContainer");
```

O esta a palavra-chave aponta para o innerContainer elemento porque esse é o elemento no qual o manipulador atual foi *registrado,* Considerando que a event.target propriedade aponta para o innerContainer elemento porque é o elemento no qual o evento tem *ocorreu.*

Em seguida, o evento borbulha até o outerContainer manipulador. Desta vez, o esta palavra-chave e o event.target apontar para diferentes elementos:

```
    afirmar (este === outerContainer,
                "Isso se refere ao outerContainer"); assert (event.target === innerContainer,

                "event.target refere-se ao innerContainer");
```

Como esperado, o esta palavra-chave refere-se ao outerContainer , porque este é o elemento no qual o manipulador atual foi registrado. Por outro lado, o event.target propriedade aponta para o innerContainer elemento, porque este é o elemento no qual o evento ocorreu.

Agora que entendemos como um evento é propagado por meio da árvore DOM e como acessar o elemento no qual o evento ocorreu originalmente, vamos ver como aplicar esse conhecimento para escrever código com menos memória intensiva.

DELEGANDO EVENTOS PARA UM ANCESTOR

Digamos que desejamos indicar visualmente se uma célula de uma tabela foi clicada pelo usuário, inicialmente exibindo um fundo branco para cada célula e, em seguida, alterando a cor de fundo para amarelo após o clique na célula. Parece muito fácil. Podemos iterar por todas as células e estabelecer um manipulador em cada uma que altera a propriedade da cor de fundo:

```
células const = document.querySelectorAll ('td'); para (seja n = 0; n <comprimento das
células; n ++) {
    cells [n] .addEventListener ('click', function () {
        this.style.backgroundColor = 'amarelo';
    });
}
```

Claro que funciona, mas é elegante? Não. Estamos estabelecendo exatamente o mesmo manipulador de eventos em potencialmente centenas de elementos, e todos eles *exatamente a mesma coisa.*

Uma abordagem muito mais elegante é estabelecer um único manipulador em um nível mais alto do que as células que podem manipular todos os eventos usando o bubbling de eventos. Sabemos que todas as células serão descendentes de sua tabela envolvente e sabemos que podemos obter uma referência para o elemento que foi clicado por meio de event.target. É muito mais suave para

delegar o tratamento de eventos para a mesa, da seguinte maneira:

```
const table = document.getElementById ('someTable'); table.addEventListener ('click', function
(event) {
    if (event.target.tagName.toLowerCase () === 'td')
        event.target.style.backgroundColor = 'amarelo';
});
```

Executa uma ação apenas se o clique acontecer em um elemento de célula (e não em um descendente aleatório)

Aqui, estabelecemos um manipulador que facilmente manipula o trabalho de alterar a cor de fundo de todas as células da tabela que foram clicadas. Isso é muito mais eficiente e elegante.

Com a delegação de eventos, temos que nos certificar de que ela é aplicada apenas a elementos que são ancestrais dos elementos que são os alvos do evento. Dessa forma, temos certeza de que os eventos irão eventualmente borbulhar até o elemento ao qual o manipulador foi delegado.

Até agora, lidamos com eventos fornecidos pelo navegador, mas você nunca desejou fervorosamente a capacidade de acionar o seu próprio *personalizadas* eventos?

13.3.2 Eventos personalizados

Imagine um cenário no qual você deseja executar uma ação, mas deseja acioná-la sob uma variedade de condições a partir de diferentes partes do código, talvez até mesmo de um código que esteja em arquivos de script compartilhados. Um novato repetiria o código em todos os lugares necessários. Um jornaleiro criaria uma função global e a chamaria de qualquer lugar que fosse necessário. Um ninja usaria eventos personalizados. Mas por que?

ACOPLAMENTO SOLTO

Digamos que estejamos fazendo operações a partir do código compartilhado e queremos que o código da página saiba quando é hora de reagir a uma condição específica. Se usarmos a abordagem de função global do jornaleiro, apresentaremos a desvantagem de que nosso código compartilhado precisa definir um nome fixo para a função e todas as páginas que usam o código compartilhado precisam usar tal função.

Além disso, e se houver várias coisas a serem feitas quando a condição de acionamento ocorrer? Fazer concessões para várias notificações seria árduo e necessariamente confuso. Essas desvantagens são resultado de *acoplamento próximo,* em que o código que detecta as condições deve conhecer os detalhes do código que reagirá a essa condição.

Acoplamento solto, por outro lado, ocorre quando o código que dispara a condição não sabe nada sobre o código que vai reagir à condição, ou mesmo se há algo que vai reagir a ela. Uma das vantagens dos manipuladores de eventos é que podemos estabelecer quantos quisermos, e esses manipuladores são completamente independentes uns dos outros. Portanto, o tratamento de eventos é um bom exemplo de acoplamento fraco. Quando um evento de clique de botão é disparado, o código que dispara o evento não tem conhecimento de quais manipuladores estabelecemos na página, ou mesmo se houver algum. Em vez disso, o evento click é enviado para a fila de tarefas pelo navegador, e o que quer que tenha causado o disparo do evento não se importa com o que acontece depois disso. Se os manipuladores tiverem sido estabelecidos para o evento de clique, eles serão chamados individualmente de maneira completamente independente.

Há muito a ser dito sobre o acoplamento fraco. Em nosso cenário, o código compartilhado, quando detecta uma condição interessante, dispara um sinal de algum tipo que diz: “Esta coisa interessante aconteceu; qualquer pessoa interessada pode lidar com isso ”, e não dá a mínima se alguém está interessado. Vamos examinar um exemplo concreto.

UM EXEMPLO AJAX-Y

Vamos fingir que escrevemos algum código compartilhado que realizará uma solicitação Ajax. As páginas nas quais esse código será usado desejam ser notificadas quando uma solicitação Ajax começa e quando termina; cada página tem suas próprias coisas que precisa fazer quando esses “eventos” ocorrem.

Por exemplo, em uma página usando este pacote, queremos exibir um catavento giratório quando uma solicitação Ajax começa e queremos ocultá-lo quando a solicitação for concluída, a fim de fornecer ao usuário algum feedback visual de que uma solicitação está sendo processada. Se imaginarmos a condição inicial como um evento denominado ajax-start, e a condição de parada como
ajax-complete, não seria ótimo se pudéssemos estabelecer manipuladores de eventos na página para esses eventos que mostram e ocultam a imagem conforme apropriado?

Considere isto:

```
document.addEventListener ('ajax-start', e => {
    document.getElementById ('whirlyThing'). style.display = 'bloco embutido'; });

document.addEventListener ('ajax-complete', e => {
    document.getElementById ('whirlyThing'). style.display = 'none'; });
```

Infelizmente, esses eventos não existem, mas nada nos impede de trazê-los à existência.

CRIANDO EVENTOS PERSONALIZADOS

Os eventos personalizados são uma forma de simular (para o usuário do nosso código compartilhado) a experiência de um evento real, mas um evento que tem sentido comercial dentro do contexto de nossa aplicação. A lista a seguir mostra um exemplo de como disparar um evento personalizado.

Listagem 13.10 Usando eventos personalizados

```
<style>
    # whirlyThing {display: none; } </style>

<button type = "button" id = "clickMe"> Iniciar </button>
<img id = "whirlyThing" src = "whirly-thing.gif" />

<script>
    function triggerEvent (target, eventType, eventDetail) {
        const event = new CustomEvent (eventType, {
            detalhe: eventDetail
        });
        target.dispatchEvent (evento);

function performAjaxOperation () {
    triggerEvent (documento, 'ajax-start', {url: 'meu-url'}); setTimeout (() => {

        triggerEvent (documento, 'ajax-complete');
    }, 5000);
}

botão const = document.getElementById ('clickMe'); button.addEventListener ('click', () => {

    performAjaxOperation ();
});

document.addEventListener ('ajax-start', e => {
    document.getElementById ('whirlyThing'). style.display = 'bloco embutido'; assert (e.detail.url === 'my-url', 'Podemos passar dados do
  evento'); });

document.addEventListener ('ajax-complete', e => {
        document.getElementById ('whirlyThing'). style.display = 'none';
});
</script>
```

Um botão no qual clicaremos para simular uma solicitação Ajax

Nossa imagem giratória, O que indica carregando, se mostrado

Usa o CustomEvent construtor para criar um novo evento

Passes em em formação para o evento objeto Através dos o detalhe } propriedade

Usa o método interno dispatchEvent para despachar o evento para o elemento especificado

Imita nossa solicitação Ajax com um cronômetro. No início da execução, dispara o evento ajax-start. Depois de decorrido um tempo suficiente, aciona o evento ajax-complete. Passa em um URL como dados adicionais do evento

Quando um botão é clicado, o Ajax Operação começou.

Verifica que podemos acessar dados adicionais do evento

Alças o ajax- iniciar evento mostrando nosso whirly coisa

Lida com o evento Ajax-complete escondendo nossa coisa whirly

Neste exemplo, exploramos eventos personalizados estabelecendo o cenário descrito na seção anterior: Uma imagem de cata-vento animada é exibida enquanto uma operação Ajax está em andamento. A operação é acionada com o clique de um botão.

De uma forma completamente dissociada, um manipulador para um evento personalizado chamado ajaxstart é estabelecido, como é o do ajax-complete evento personalizado. Os manipuladores para esses eventos mostram e ocultam a imagem do cata-vento, respectivamente:

```
    button.addEventListener ('click', () => {
        performAjaxOperation ();
    });

    document.addEventListener ('ajax-start', e => {
        document.getElementById ('whirlyThing'). style.display = 'bloco embutido'; assert (e.detail.url === 'my-url', 'Podemos passar dados do
      evento'); });


    document.addEventListener ('ajax-complete', e => {
            document.getElementById ('whirlyThing'). style.display = 'none';
    });
```

Observe que os três manipuladores nada sabem da existência um do outro. Em particular, o manipulador de clique de botão não tem responsabilidades com relação a mostrar e ocultar a imagem.

A própria operação Ajax é simulada com o seguinte código:

```
function performAjaxOperation () {
    triggerEvent (documento, 'ajax-start', {url: 'meu-url'}); setTimeout (() => {

        triggerEvent (documento, 'ajax-complete');
    }, 5000);
}
```

A função aciona o ajax-start evento e envia dados sobre o evento (o url
), fingindo que uma solicitação Ajax está prestes a ser feita. A função então emite um tempo limite de 5 segundos, simulando uma solicitação Ajax que se estende por 5 segundos. Quando o cronômetro expira, fingimos que a resposta foi retornada e acionamos um ajaxcomplete evento para indicar que a operação Ajax foi concluída.

Observe o alto grau de desacoplamento em todo este exemplo. O código de operação Ajax compartilhado não tem conhecimento do que o código da página fará quando os eventos forem disparados, ou mesmo se houver código da página para disparar. O código da página é modularizado em pequenos manipuladores que não conhecem uns aos outros. Além disso, o código da página não tem ideia de como o código compartilhado está funcionando; ele apenas reage a eventos que podem ou não ser acionados.

Esse nível de desacoplamento ajuda a manter o código modular, mais fácil de escrever e muito mais fácil de depurar quando algo dá errado. Também torna mais fácil compartilhar partes do código e movê-las sem medo de violar uma dependência acoplada entre os fragmentos de código. O desacoplamento é uma vantagem fundamental ao usar eventos customizados no código e nos permite desenvolver aplicativos de uma maneira muito mais expressiva e flexível.

13,4 Resumo

- Uma tarefa de loop de evento representa uma ação executada pelo navegador. As tarefas são agrupadas em duas categorias:

  - As macrotarefas são ações discretas e independentes do navegador, como criar o objeto do documento principal, manipular vários eventos e fazer alterações de URL.

  - Microtarefas são tarefas menores que devem ser executadas o mais rápido possível. Os exemplos incluem retornos de chamada de promessa e alterações de mutação DOM.

- Por causa do modelo de execução de thread único, as tarefas são processadas uma por vez e, depois que uma tarefa começa a ser executada, ela não pode ser interrompida por outra tarefa. O loop de eventos geralmente tem pelo menos duas filas de eventos: uma fila de macrotarefa e uma fila de microtarefa.

- Timers fornecem a capacidade de atrasar de forma assíncrona a execução de um trecho de código por *pelo menos* algum número de milissegundos.

  - Use o setTimeout função para executar um retorno de chamada uma vez após o atraso especificado ter decorrido.

  - Use o setInterval para iniciar um cronômetro que tentará executar o retorno de chamada no intervalo de atraso especificado, até ser cancelado.

  - Ambas as funções retornam um ID do temporizador que podemos usar para cancelar um temporizador através de clearTimeout e clearInterval funções.

  - Use temporizadores para dividir códigos caros computacionalmente em pedaços gerenciáveis que não obstruam o navegador.

- O DOM é uma árvore hierárquica de elementos e um evento que ocorre em um elemento (o destino) geralmente é propagado por meio do DOM. Existem dois mecanismos de propagação:

  - Na captura de evento, o evento desce do elemento superior até o elemento de destino.

  - No evento borbulhando, o evento borbulha do elemento de destino até o elemento superior.

- Ao chamar manipuladores de eventos, o navegador também passa um objeto de evento. Acesse o elemento no qual o evento ocorreu por meio do alvo propriedade. Dentro do manipulador, use o esta palavra-chave para se referir ao elemento no qual o manipulador foi registrado.

- Use eventos personalizados, criados por meio do CustomEvent construtor, e despachado com o dispatchEvent método, para reduzir o acoplamento entre as diferentes partes de seu aplicativo.

13.5 Exercícios

1 Por que é importante que a adição de tarefas à fila de tarefas aconteça fora do loop de eventos?

2 Por que é importante que cada iteração do loop de eventos não leve muito mais de cerca de 16 ms?

3 Qual é o resultado da execução do código a seguir por 2 segundos?

```
setTimeout (function () {
    console.log ("Tempo limite");
}, 1000);

setInterval (function () {
    console.log ("Intervalo");
}, 500);
```

uma Timeout Interval Interval Interval Interval

b Intervalo Tempo Limite Intervalo Intervalo Intervalo

c Tempo Limite do Intervalo

4 Qual é o resultado da execução do código a seguir por 2 segundos?

```
const timeoutId = setTimeout (function () {
    console.log ("Tempo limite");
}, 1000);

setInterval (function () {
    console.log ("Intervalo");
}, 500);

clearTimeout (timeoutId);
```

uma Intervalo Tempo Limite Intervalo Intervalo Intervalo

b Intervalo

c Intervalo Intervalo Intervalo Intervalo

5 Qual é o resultado de executar o código a seguir e clicar no elemento com o ID interno?

```
<body>
    <div id = "outer">
        <div id = "inner"> </div>
    </div>
    <script>
        const innerElement = document.querySelector ("# inner"); const outerElement = document.querySelector
        ("# outer"); const bodyElement = document.querySelector ("body");


        innerElement.addEventListener ("click", function () {
                console.log ("Interno");
        });

        outerElement.addEventListener ("click", function () {
                console.log ("Externo");
        }, verdade);

        bodyElement.addEventListener ("click", function () {
                console.log ("Corpo");
```

```
        })
    </script>
</body>
```

uma Corpo Externo Interno

b Corpo Externo Interno

c Corpo Interno Externo

14 Em desenvolvimento estratégias para vários navegadores

Este capítulo cobre

- Desenvolvimento de código JavaScript reutilizável em vários navegadores
- Analisando problemas entre navegadores que precisam ser resolvidos
- Lidando com esses problemas de maneira inteligente

Qualquer pessoa que tenha passado um tempo desenvolvendo código JavaScript na página sabe que existe uma ampla gama de pontos problemáticos quando se trata de garantir que o código funcione perfeitamente em um conjunto de navegadores compatíveis. Essas considerações vão desde o fornecimento de desenvolvimento básico para necessidades imediatas até o planejamento de futuras versões do navegador, até a reutilização de código em páginas da web que ainda não foram criadas.

Codificar para vários navegadores é uma tarefa não trivial que deve ser balanceada de acordo com as metodologias de desenvolvimento que você possui, bem como os recursos disponíveis para o seu projeto. Por mais que adoraríamos que nossas páginas funcionassem perfeitamente em todos os navegadores que já existiram ou que existirão, a realidade vai mostrar sua cara feia e devemos perceber que temos recursos de desenvolvimento finitos. Devemos planejar para aplicar esses recursos de forma adequada e cuidadosa, obtendo o maior retorno para nosso investimento.

Por isso, começamos este capítulo com conselhos sobre como escolher quais navegadores oferecer suporte. Isso é seguido por uma discussão das principais preocupações com relação ao desenvolvimento de crossbrowser, bem como estratégias eficazes para lidar com esses problemas. Vamos começar a escolher cuidadosamente os navegadores compatíveis.

Você sabe?

Quais são algumas maneiras comuns de lidar com inconsistências cies no comportamento com navegadores diferentes?

Qual é a melhor maneira de tornar seu código utilizável em outros as páginas das pessoas?

Por que polyfills são úteis em scripts entre navegadores?

14.1 Considerações entre navegadores

Aperfeiçoar nossas habilidades de programação em JavaScript nos levará muito longe, especialmente agora que o JavaScript escapou dos limites do navegador e está sendo usado no servidor com Node.js. Mas ao desenvolver aplicativos JavaScript baseados em navegador (que é o foco deste livro), mais cedo ou mais tarde, vamos enfrentar primeiro *Os navegadores* e seus vários problemas e inconsistências.

Em um mundo perfeito, todos os navegadores estariam livres de erros e ofereceriam suporte aos padrões da web de maneira consistente, mas como todos sabemos, não vivemos nesse mundo. Embora a qualidade dos navegadores tenha melhorado muito nos últimos tempos, todos ainda têm alguns bugs, APIs ausentes e peculiaridades específicas do navegador com os quais precisaremos lidar. O desenvolvimento de uma estratégia abrangente para lidar com esses problemas do navegador e se familiarizar intimamente com suas diferenças e peculiaridades não é menos importante do que a proficiência no próprio JavaScript.

Ao escrever aplicativos de navegador, é importante escolher quais navegadores dar suporte. Provavelmente gostaríamos de oferecer suporte aos pequenos, mas as limitações de desenvolvimento e recursos de teste ditam o contrário. Então, como decidimos quais apoiar e em que nível?

Uma abordagem que podemos empregar foi livremente emprestada de um antigo Yahoo! aproximação, *suporte de navegador classificado.* Nessa técnica, criamos uma matriz de suporte ao navegador que serve como um instantâneo da importância de um navegador e sua plataforma para nossas necessidades. Nesta tabela, listamos as plataformas de destino em um eixo e os navegadores no outro. Então, nas células da tabela, damos uma "nota" (A a F, ou qualquer outro sistema de notas que atenda às nossas necessidades) para cada combinação de navegador / plataforma. A Tabela 14.1 mostra um exemplo hipotético.

Observe que não preenchemos nenhuma nota. As notas que você atribui a uma combinação específica de plataforma e navegador dependem inteiramente das necessidades e requisitos de seu projeto, bem como de outros fatores importantes, como a composição do público-alvo. Podemos usar essa abordagem para chegar a notas que medem a importância do suporte para essa plataforma / navegador e combinar essas informações com o custo desse suporte para tentar chegar ao conjunto ideal de navegadores suportados.

Tabela 14.1 Uma matriz hipotética de suporte ao navegador

	janelas	OS X	Linux	iOS	Android
IE 9		N / D	N / D	N / D	N / D
IE10		N / D	N / D	N / D	N / D
IE11		N / D	N / D	N / D	N / D
Beira		N / D	N / D	N / D	N / D
Raposa de fogo				N / D	
cromada					
Ópera					
Safári			N / D		N / D

Quando optamos por oferecer suporte a um navegador, normalmente fazemos as seguintes promessas:

- Testaremos ativamente esse navegador com nosso conjunto de testes.
- Vamos consertar bugs e regressões associados a esse navegador.
- O navegador executará nosso código com um nível de desempenho razoável.

Como é impraticável desenvolver em várias combinações de plataforma / navegador, devemos pesar os custos e os benefícios de oferecer suporte aos vários navegadores. Esta análise deve levar em consideração várias considerações, e as principais são as seguintes:

- As expectativas e necessidades do público-alvo
- A participação de mercado do navegador
- A quantidade de esforço necessária para suportar o navegador

O primeiro ponto é subjetivo que somente o seu projeto pode determinar. A participação de mercado, por outro lado, pode frequentemente ser medida usando as informações disponíveis. E uma estimativa aproximada do esforço envolvido no suporte de cada navegador pode ser determinada considerando os recursos dos navegadores e sua adesão aos padrões modernos.

A Figura 14.1 mostra um gráfico de amostra que representa informações sobre o uso do navegador (obtido a partir de http://gs.statcounter.c para abril de 2016). Qualquer pedaço de código JavaScript reutilizável, seja uma biblioteca JavaScript de consumo em massa ou nosso próprio código na página, deve ser desenvolvido para funcionar em tantos ambientes quanto possível, concentrando-se nos navegadores e plataformas que são importantes para o usuário final. Para bibliotecas de consumo em massa, esse é um grande conjunto; para aplicativos mais direcionados, o conjunto necessário pode ser mais estreito.

Mas é de vital importância não morder mais do que você pode mastigar, e a qualidade nunca deve ser sacrificada pela cobertura. Isso é importante o suficiente para repetir; na verdade, recomendamos que você leia em voz alta:

A qualidade nunca deve ser sacrificada pela cobertura.

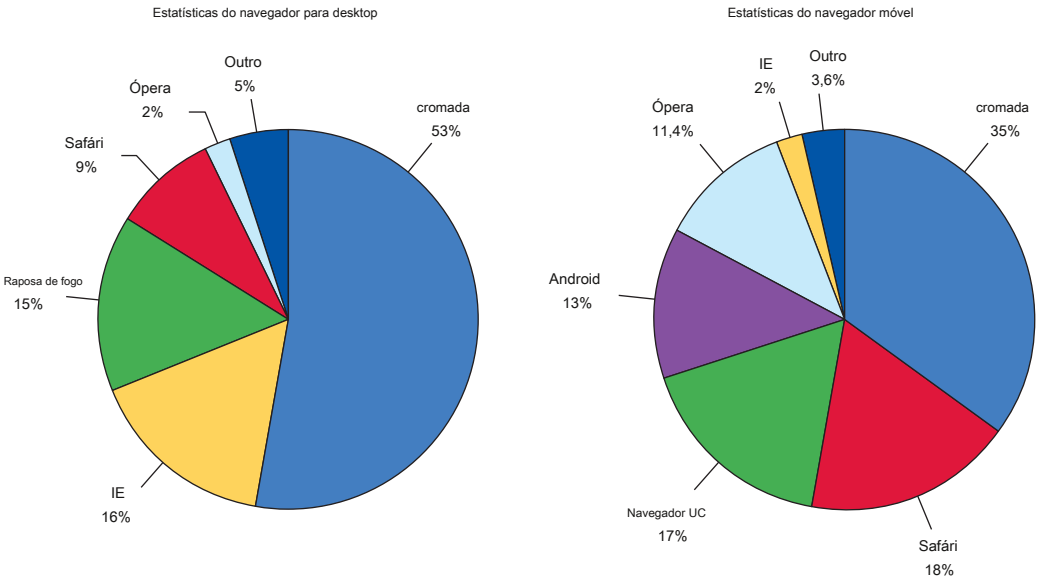


Figura 14.1 Observar as estatísticas de uso de navegadores em desktops e dispositivos móveis nos dá uma ideia de quais navegadores devemos concentrar nossa atenção.

Neste capítulo, examinaremos as situações em que o código JavaScript se encontrará em relação ao suporte para navegadores cruzados. Em seguida, examinaremos algumas das melhores maneiras de escrever esse código com o objetivo de aliviar quaisquer problemas potenciais que essas situações representam. Isso deve ajudar muito a decidir quais dessas técnicas vale a pena adotar e deve ajudá-lo a preencher seu próprio gráfico de suporte do navegador.

14.2 As cinco principais preocupações de desenvolvimento

Qualquer pedaço de código não trivial carrega uma miríade de preocupações de desenvolvimento com que se preocupar. Mas cinco pontos principais representam os maiores desafios para nosso código JavaScript reutilizável, conforme ilustrado na Figura 14.2.

Estes são os cinco pontos:

- Bugs do navegador
- Correções de bug do navegador
- Código externo
- Regressões de navegador
- Recursos ausentes nos navegadores

Bem Tempo queremos equilibrar o valor que gastamos em cada ponto com os benefícios resultantes. Em última análise, estes

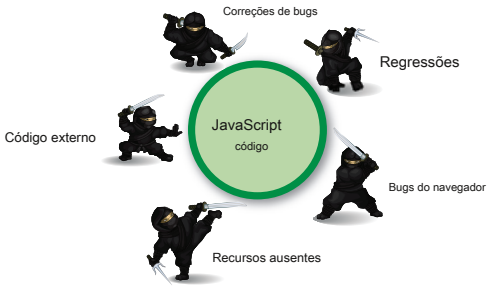


Figura 14.2 Os cinco principais pontos de preocupação para o desenvolvimento de JavaScript reutilizável

são perguntas que você terá que responder, aplicando-as à sua situação. Uma análise de seu público-alvo, recursos de desenvolvimento e cronograma são fatores que influenciam sua decisão.

Ao nos esforçarmos para desenvolver código JavaScript reutilizável, devemos levar todos os pontos em consideração, mas prestar mais atenção aos navegadores mais populares que existem no momento, porque eles são mais prováveis de serem usados por nosso público-alvo. Com outros navegadores menos populares, devemos pelo menos nos certificar de que nosso código seja degradado normalmente. Por exemplo, se um navegador não oferece suporte a uma determinada API, no mínimo, devemos ter cuidado para que nosso código não lance nenhuma exceção para que o resto do código possa ser executado.

Nas seções a seguir, analisaremos essas várias preocupações para entender melhor os desafios que enfrentamos e como combatê-los.

14.2.1 Bugs e diferenças do navegador

Uma das preocupações com a qual precisaremos lidar ao desenvolver o código JavaScript reutilizável é lidar com os vários bugs e diferenças de API associadas ao conjunto de navegadores que decidimos oferecer suporte. Embora os navegadores sejam muito mais uniformes atualmente, quaisquer recursos que fornecemos em nosso código devem ser completamente e *verificável* correto em *todo* navegadores que escolhemos oferecer suporte.

A maneira de conseguir isso é direta: precisamos de um conjunto abrangente de testes para cobrir os casos de uso comuns e marginais do código. Com uma boa cobertura de teste, podemos nos sentir seguros em saber que o código que desenvolvemos funcionará no conjunto de navegadores com suporte. E presumindo que nenhuma alteração subsequente do navegador rompa a compatibilidade com versões anteriores, teremos uma sensação calorosa e confusa de que nosso código funcionará até mesmo em versões futuras desses navegadores. Veremos estratégias específicas para lidar com bugs e diferenças do navegador na seção 14.3.

Um ponto complicado em tudo isso é implementar correções para bugs atuais do navegador de forma que sejam resistentes a quaisquer correções para esses bugs que sejam implementados em versões futuras do navegador.

14.2.2 Correções de bug do navegador

Assumir que um navegador apresentará para sempre um bug específico é temerário - a maioria dos bugs do navegador eventualmente são corrigidos e *contando* na presença do bug é uma estratégia de desenvolvimento perigosa. É melhor usar as técnicas da seção 14.3 para certificar-se de que quaisquer soluções alternativas de bug sejam preparadas para o futuro, tanto quanto possível.

Ao escrever um trecho de código JavaScript reutilizável, queremos ter certeza de que ele pode durar muito tempo. Assim como acontece com a escrita de qualquer aspecto de um site (CSS, HTML e assim por diante), não queremos voltar e corrigir o código que está quebrado por um novo lançamento do navegador.

Fazer suposições sobre bugs do navegador causa uma forma comum de quebra de site: hacks específicos colocados em prática para contornar bugs apresentados por um navegador que quebram quando o navegador corrige os bugs em versões futuras.

O problema de lidar com bugs do navegador é duplo:

- Nosso código pode quebrar quando a correção do bug for instituída.
- Podemos acabar treinando fornecedores de navegadores para *não* consertar bugs por medo de causar falhas em sites.

Um exemplo interessante da segunda situação ocorreu recentemente, com o scrollTop erro (https://dev.opera.com/articles/fixing-the-scrolltop-bug/) Ao lidar com elementos no HTML DOM, podemos usar o scrollTop e scrollLeft propriedades para acessar e modificar a posição de rolagem atual do elemento. Mas se usarmos essas propriedades na raiz, html elemento, essas propriedades devem, de acordo com a especificação, em vez disso relatar (e influenciar) a posição de rolagem da janela de visualização. O IE 11 e o Firefox seguem de perto essa especificação. Infelizmente, Safari, Chrome e Opera não. Em vez disso, se você tentar modificar essas propriedades da raiz, html elemento, nada acontece. Para obter o mesmo efeito nesses navegadores, temos que usar o scrollTop e scrollLeft propriedades no corpo elemento.

Quando confrontados com essa inconsistência, os desenvolvedores da web frequentemente recorrem à detecção do nome atual do navegador (por meio da string do agente do usuário, mais sobre isso posteriormente) e, em seguida, modificando o scrollTop e scrollLeft do html elemento se o nosso código JavaScript estiver sendo executado no IE ou Firefox, e do corpo elemento se o código estiver sendo executado no Safari, Chrome ou Opera. Infelizmente, essa maneira de contornar esse bug provou ser desastrosa. Como muitas páginas agora codificam explicitamente "se este for Safari, Chrome ou Opera", modifique o corpo , esses navegadores não podem realmente corrigir esse bug, porque a correção do bug, ironicamente, causaria falhas em muitas páginas da web.

Isso traz outro ponto importante sobre os bugs: ao determinar se uma parte da funcionalidade é potencialmente um bug, sempre verifique-o com a especificação!

Um bug do navegador também é diferente de uma API não especificada. É importante consultar as especificações do navegador, porque elas fornecem os padrões exatos que os navegadores usam para desenvolver e melhorar seu código. Em contraste, a implementação de uma API não especificada pode mudar a qualquer momento (especialmente se a implementação tentar se tornar padronizada). No caso de inconsistências em APIs não especificadas, você deve sempre testar a saída esperada. Esteja sempre ciente de que mudanças futuras podem ocorrer nessas APIs à medida que se solidificam.

Além disso, há uma distinção entre correções de bugs e alterações de API. Considerando que correções de bugs são facilmente previstas - um navegador eventualmente corrigirá os bugs em sua implementação, mesmo que leve muito tempo - as alterações de API são muito mais difíceis de detectar. É improvável que as APIs padrão mudem (embora não seja totalmente inédito); as alterações são muito mais prováveis de ocorrer com APIs não especificadas.

Felizmente, isso raramente acontece de uma forma que prejudique massivamente a maioria dos aplicativos da web. Mas se isso acontecer, é efetivamente indetectável com antecedência (a menos, é claro, que testemos cada API que já tocamos - mas a sobrecarga incorrida em tal processo seria ridícula). Mudanças de API desse tipo devem ser tratadas como qualquer outra regressão.

Para nosso próximo ponto de preocupação, sabemos que nenhum homem é uma ilha e nem o nosso código. Vamos explorar as ramificações.

14.2.3 Código externo e marcação

Qualquer código reutilizável deve coexistir com o código que o cerca. Quer estejamos esperando que nosso código funcione em páginas que escrevemos ou em sites desenvolvidos por terceiros, precisamos garantir que ele possa existir na página com qualquer outro código aleatório.

Esta é uma faca de dois gumes: nosso código não apenas deve ser capaz de suportar a convivência com código externo potencialmente mal escrito, mas também deve tomar cuidado para não ter efeitos adversos no código com o qual vive.

Exatamente quanto precisamos estar vigilantes sobre esse ponto de preocupação depende muito do ambiente no qual esperamos que o código seja usado. Por exemplo, se estamos escrevendo um código reutilizável para um único ou um número limitado de sites sobre os quais temos algum nível de controle, pode ser seguro nos preocupar menos com os efeitos do código externo porque sabemos onde o código irá operar, e nós podemos, até certo ponto, corrigir quaisquer problemas por nós mesmos.

GORJETA Esta é uma preocupação importante o suficiente para justificar um livro inteiro sobre o assunto. Se você gostaria de se aprofundar mais, recomendamos altamente *JavaScript de terceiros* por Ben Vinegar e Anton Kovalyov (Manning, 2013, https: // www.manning.com/books/third-party-javascript)

Se estivermos desenvolvendo código que terá um amplo nível de aplicabilidade em ambientes desconhecidos (e incontroláveis), precisaremos ter a certeza de que nosso código é robusto. Vamos discutir algumas estratégias para conseguir isso.

ENCAPSULANDO NOSSO CÓDIGO

Para evitar que nosso código afete outras partes do código nas páginas onde é carregado, é melhor praticar *encapsulamento*. Em geral, refere-se ao ato de colocar algo dentro ou como se estivesse dentro de uma cápsula. Uma definição mais focada no domínio é “um mecanismo de linguagem para restringir o acesso a alguns dos componentes do objeto”. Sua tia Mathilda pode resumir de forma mais sucinta como “Mantenha o seu nariz no seu próprio negócio!”

Manter uma pegada global incrivelmente pequena ao introduzir nosso código em uma página pode ajudar muito a deixar tia Mathilda feliz. Na verdade, manter nossa pegada global para um punhado de variáveis globais, ou melhor ainda, *1,* é bastante fácil.

Como você viu no capítulo 12, jQuery, a biblioteca JavaScript mais popular do lado do cliente, é um bom exemplo disso. Ele apresenta uma variável global (uma função) chamada jQuery,
e um alias para essa variável global, $. Ele ainda possui um meio suportado de devolver o $ alias a qualquer outro código na página ou outra biblioteca que queira usá-lo.

Quase todas as operações em jQuery são feitas por meio do jQuery função. E quaisquer outras funções que ele forneça (chamadas *funções utilitárias)* são definidos como propriedades de jQuery
(lembre-se do capítulo 3 como é fácil definir funções que são propriedades de outras funções), usando assim o nome jQuery como um *namespace* para todas as suas definições.

Podemos usar a mesma estratégia. Digamos que estejamos definindo um conjunto de funções para nosso próprio uso, ou para uso de outros, que agruparemos em um namespace de nossa própria escolha - digamos, ninja.

Poderíamos, como jQuery, definir uma função global chamada ninja () que executa várias operações com base no que passamos para a função. Por exemplo:

```
var ninja = function () {/ * o código de implementação vai aqui * /}
```

Definir nossas próprias funções de utilitário que usam esta função como seu namespace é fácil:

```
ninja.hitsuke = function () {/ * código para distrair os guardas com fogo aqui * /}
```

Se não quisermos ou precisarmos ninja para ser uma função, mas apenas para servir como um namespace, poderíamos defini-la da seguinte maneira:

```
var ninja = {};
```

Isso cria um objeto vazio no qual podemos definir propriedades e funções para evitar a adição desses nomes ao namespace global.

Outras práticas que queremos evitar, a fim de manter nosso código encapsulado, são a modificação de quaisquer variáveis existentes, protótipos de função ou mesmo elementos DOM. Qualquer aspecto da página que nosso código modifica, fora dele, é uma área potencial de colisão e confusão.

O outro lado da rua de mão dupla é que, mesmo que *nós* seguir as práticas recomendadas e encapsular cuidadosamente nosso código, não podemos ter certeza de que o código que não escrevemos será tão bem comportado.

LIDANDO COM CÓDIGO MENOS DO QUE EXEMPLAR

Há uma velha piada que circula desde que Grace Hopper removeu aquela mariposa de um retransmissor no período Cretáceo: “O único código que não é uma merda é o código que você mesmo escreve”. Isso pode parecer cínico, mas quando nosso código coexiste com código que não podemos controlar, devemos presumir o pior, apenas por segurança.

Algum código, mesmo se bem escrito, pode *intencionalmente* estar fazendo coisas como modificar protótipos de funções, propriedades de objetos e métodos de elementos DOM. Essa prática, bem-intencionada ou não, pode nos colocar em armadilhas.

Nessas circunstâncias, nosso código poderia estar fazendo algo inócuo, como usar arrays JavaScript, e ninguém poderia nos culpar por fazer a simples suposição de que os arrays JavaScript vão agir como arrays JavaScript. Mas se algum outro código na página modificar a maneira como os arrays funcionam, nosso código pode acabar não funcionando como planejado, sem absolutamente nenhuma falha nossa.

Infelizmente, não existem muitas regras constantes ao lidar com situações dessa natureza, mas podemos tomar algumas medidas atenuantes. As próximas seções apresentam essas etapas defensivas.

LIDANDO COM IDS GOSTOSOS

A maioria dos navegadores exibe um *anti-recurso (* não podemos chamá-lo de *erro* porque o comportamento é absolutamente intencional) que pode fazer com que nosso código tropece e caia inesperadamente. Este recurso faz com que referências de elemento sejam adicionadas a outros elementos usando o eu ia ou nome
atributos do elemento original. E quando isso eu ia ou nome conflita com propriedades que já fazem parte do elemento, coisas ruins podem acontecer.

Dê uma olhada no seguinte trecho de HTML para observar a maldade que pode resultar desses *IDs gananciosos:*

```
<form id = "form" action = "/ ocultar">
    <input type = "text" id = "action" />
    <input type = "submit" id = "submit" />
</form>
```

Agora, nos navegadores, vamos chamar isso de:

```
var what = document.getElementById ('form'). action;
```

Corretamente, esperamos que este seja o valor do formulário açao atributo. E na maioria dos casos, seria. Mas se inspecionarmos o valor da variável o que, descobrimos que, em vez disso, é uma referência ao input # action elemento! Huh?

Vamos tentar outra coisa:

```
document.getElementById ('formulário'). submit ();
```

Essa declaração deve fazer com que o formulário seja enviado, mas em vez disso, obtemos um erro de script:

```
TypeError não capturado: a propriedade 'submit' do objeto # <HTMLFormElement> não é uma função
```

O que está acontecendo?

Os navegadores adicionaram propriedades ao < formulário> elemento para cada um dos elementos de entrada dentro do formulário que faz referência ao elemento. Isso pode parecer útil no início, até percebermos que o nome da propriedade adicionada é tirado do eu ia ou nome valores dos elementos de entrada. E se esse valor for apenas uma propriedade já usada do elemento do formulário, como açao ou enviar, essas propriedades originais são substituídas pela nova propriedade. Isso geralmente é conhecido como *DOM clobbering*.

Então, antes do input # submit elemento é criado, a referência form.action aponta para o valor do açao atributo para o < formulário>. Depois, ele aponta para o
input # submit elemento. A mesma coisa acontece com form.submit. Sim!

Isso é um resquício de muito tempo atrás, de uma época em que os navegadores não tinham um rico conjunto de métodos API para buscar elementos do DOM. Os fornecedores de navegadores adicionaram esse recurso para fornecer acesso fácil aos elementos do formulário. Hoje em dia, podemos acessar facilmente qualquer elemento no DOM, portanto, ficamos apenas com os efeitos colaterais infelizes do recurso.

Em qualquer caso, esse “recurso” específico dos navegadores pode causar inúmeros e confusos problemas em nosso código, e devemos mantê-lo em mente ao depurar. Quando encontramos propriedades que aparentemente foram inexplicavelmente transformadas em algo diferente do que esperamos, a destruição do DOM é um provável culpado.

Felizmente, podemos evitar esse problema em nossa própria marcação, evitando eu ia e
nome valores que podem entrar em conflito com nomes de propriedade padrão e podemos incentivar outros a fazer o mesmo. O valor que enviar deve ser evitado principalmente, pois é uma fonte comum de comportamento problemático e frustrante.

CARREGANDO A ORDEM DE FOLHAS DE ESTILO E SCRIPTS

Freqüentemente, esperamos que as regras CSS já estejam disponíveis quando nosso código for executado. Uma das melhores maneiras de garantir que as regras CSS fornecidas pelas folhas de estilo sejam definidas quando nosso código JavaScript é executado é incluir as folhas de estilo externas *prévio* para incluir os arquivos de script externos.

Não fazer isso pode causar resultados inesperados, porque o script tenta acessar as informações de estilo ainda indefinidas. Infelizmente, esse problema não pode ser corrigido facilmente com JavaScript puro e, em vez disso, deve ser tratado com a documentação do usuário.

Essas últimas seções cobriram alguns exemplos básicos de como as externalidades podem afetar o funcionamento do nosso código, frequentemente de maneiras não intencionais e confusas. Problemas com nosso código costumam aparecer quando outros usuários tentam integrá-lo *deles* locais, ponto em que devemos ser capazes de diagnosticar os problemas e construir testes apropriados para lidar com eles. Em outras ocasiões, descobriremos esses problemas ao integrar o código de outras pessoas em nossas páginas e, esperançosamente, as dicas nessas seções ajudarão a identificar as causas.

É uma pena que não haja soluções melhores e deterministas para lidar com esses problemas de integração, a não ser dar alguns primeiros passos inteligentes e escrever nosso código de forma defensiva. Vamos agora passar para o próximo ponto de preocupação.

14.2.4 Regressões

Regressões são um dos problemas mais difíceis que encontraremos na criação de código JavaScript reutilizável e sustentável. Esses são bugs, ou alterações de API não compatíveis com versões anteriores (principalmente para APIs não especificadas), que os navegadores introduziram e que fazem com que o código seja interrompido de maneiras imprevisíveis.

NOTA Aqui estamos usando o termo *regressão* em sua definição clássica: um recurso que costumava funcionar, mas não funciona mais como esperado. Isso geralmente não é intencional, mas às vezes é causado por alterações deliberadas que quebram o código existente.

ANTECIPANDO MUDANÇAS

Lá *estamos* algumas mudanças de API que, com alguma previsão, podemos detectar e tratar proativamente, conforme mostrado na Listagem 14.1. Por exemplo, com o Internet Explorer 9, a Microsoft introduziu suporte para manipuladores de eventos DOM nível 2 (vinculados usando o addEventListener
método), enquanto as versões anteriores do IE usavam o método interno específico do IE attachEvent
método. Para código escrito antes do IE 9, a detecção de recurso simples foi capaz de lidar com essa mudança.

Listagem 14.1 Antecipando uma mudança de API futura

```
function bindEvent (element, type, handle) {
    if (element.addEventListener) {
        element.addEventListener (tipo, identificador, falso);
    }
    else if (element.attachEvent) {
```

Vincula-se usando o API padrão