10.6.1 Novas linhas correspondentes

Ao realizar uma pesquisa, às vezes é desejável que o termo ponto (.), Que corresponde a qualquer caractere, exceto para nova linha, inclua também caracteres de nova linha. Implementações de expressões regulares em outras linguagens frequentemente incluem um sinalizador para tornar isso possível, mas a implementação de JavaScript não.

Vejamos algumas maneiras de contornar essa omissão no JavaScript, conforme mostrado na próxima listagem.

Listagem 10.11 Correspondência ***todo*** personagens, incluindo novas linhas

```
const html = "<b> Olá </b> \ n <i> mundo! </i>"; assert (/.*/. exec (html) [0] === "<b>
Olá </b>",
                "Uma captura normal não trata das linhas finais.");
assert (/ [\ S \ s] * /. exec (html) [0] ===
                "<b> Olá </b> \ n <i> mundo! </i>",
                "Combinando tudo com um conjunto de caracteres.");
assert (/ (?:. | \ s) * /. exec (html) [0] ===
                "<b> Olá </b> \ n <i> mundo! </i>",
                "Usando um grupo sem captura para combinar tudo.");
```

Define um assunto de teste

Mostra essas novas linhas não combinam

Corresponde a todos usando correspondência de espaço em branco

Corresponde a todos usando alteração

Este exemplo define uma string de assunto de teste: "< b> Olá \ n mundo! ", contendo uma nova linha. Em seguida, tentamos várias maneiras de combinar todos os caracteres da string.

No primeiro teste, /.*/. exec (html) [0] === " Olá ", verificamos que as novas linhas não são correspondidas pelo. operador. Os ninjas não serão negados, portanto, no próximo teste, vamos usar uma regex alternativa, / [\ WL]*/, em que definimos uma classe de personagem que corresponde a qualquer coisa que seja *não* um caractere de espaço em branco e qualquer coisa que *é* um caractere de espaço em branco. Essa união é o conjunto de todos os personagens.

Outra abordagem é feita no próximo teste:

```
/[\S\s]*/.exec(html)[0] === "<b> Olá </b> \ n <i> mundo! </i>"
```

Aqui, usamos uma regex de alternância, / (?:. | \ s) * /, em que nós combinamos tudo com correspondência por., que é tudo menos nova linha, e tudo considerado espaço em branco, o que inclui nova linha. A união resultante é o conjunto de todos os caracteres, incluindo novas linhas. Observe o uso de uma subexpressão passiva para evitar qualquer captura não intencional. Por causa de sua simplicidade (e benefícios implícitos de velocidade), a solução fornecida por / [\ WL]*/ geralmente é considerado ótimo.

A seguir, vamos dar um passo para ampliar nossa visão para um âmbito mundial.

10.6.2 Unicode correspondente

Freqüentemente, no uso de expressões regulares, desejamos corresponder caracteres alfanuméricos, como um seletor de ID em uma implementação de mecanismo seletor CSS. Mas assumir que os caracteres alfabéticos virão apenas do conjunto de caracteres ASCII em inglês é uma visão curta.

Expandir o conjunto para incluir caracteres Unicode às vezes é desejável, suportando explicitamente vários idiomas não cobertos pelo conjunto de caracteres alfanuméricos tradicional (consulte a lista a seguir).

Listagem 10.12 Correspondência de caracteres Unicode

```
const text = "\ u5FCD \ u8005 \ u30D1 \ u30EF \ u30FC";
const matchAll = / [\ w \ u0080- \ uFFFF _-] + /;
assert (text.match (matchAll), "Nosso regexp corresponde a não ASCII!");
```

Corresponde a tudo, incluindo Unicode

Esta lista inclui toda a gama de caracteres Unicode na correspondência, criando uma classe de caracteres que inclui o \ C termo, para corresponder a todos os caracteres de palavra “normais”, mais um intervalo que abrange todo o conjunto de caracteres Unicode acima de U + 0080. Começar em 128 nos dá alguns caracteres ASCII altos junto com todos os caracteres Unicode no plano multilíngue básico.

Os astutos entre vocês podem notar que, adicionando toda a gama de caracteres Unicode acima \ u0080, combinamos não apenas caracteres alfabéticos, mas também todas as pontuações Unicode e outros caracteres especiais (setas, por exemplo). Mas tudo bem, porque o objetivo do exemplo é mostrar como combinar caracteres Unicode em geral. Se você tem um intervalo específico de caracteres que deseja combinar, você pode usar a lição deste exemplo para adicionar o intervalo que deseja à classe de caracteres.

Antes de prosseguirmos em nosso exame das expressões regulares, vamos abordar mais um problema comum.

10.6.3 *Caracteres de escape correspondentes*

É comum que os autores das páginas usem nomes que estejam em conformidade com os identificadores do programa ao atribuir eu ia valores para os elementos da página, mas isso é apenas uma convenção; eu ia os valores podem conter caracteres diferentes de caracteres de “palavra”, incluindo pontuação. Por exemplo, um desenvolvedor da web pode usar o eu ia valor formulário: atualização para um elemento.

Um desenvolvedor de biblioteca, ao escrever uma implementação para, digamos, um mecanismo seletor de CSS, gostaria de oferecer suporte a caracteres de escape. Isso permite que o usuário especifique nomes complexos que não estão em conformidade com as convenções de nomenclatura típicas. Portanto, vamos desenvolver um regex que permitirá a correspondência de caracteres de escape. Considere o seguinte código.

Listagem 10.13 Caracteres de escape correspondentes em um seletor CSS

```
const pattern = /^((\w+)|(\\.))+$/; testes const = [

    "formUpdate",
    "formulário \\. atualizar \\. seja o que for",
    "formulário \\: atualização",
    "\\ f \\ o \\ r \\ m \\ u \\ p \\ d \\ a \\ t \\ e",
    "formulário: atualização"
];
```

Esta expressão regular permite qualquer sequência composta de caracteres de palavras, uma barra invertida seguida por qualquer caractere (mesmo um barra invertida) ou ambos.

Configura vários assuntos de teste. Todos deveriam passar mas o último, que falha em escapar dela não palavra personagem (:).

```
para (deixe n = 0; n <tests.length; n ++) {
    assert (pattern.test (tests [n]),
                tests [n] + "é um identificador válido");
}
```

Passa por todos os assuntos de teste

Essa expressão em particular funciona permitindo a correspondência de uma sequência de caracteres de palavra ou de uma barra invertida seguida por qualquer caractere.

Observe que é necessário mais trabalho para oferecer suporte total a todos os caracteres de escape. Para mais detalhes, visite https://mathiasbyne

10,7 Resumo

- Expressões regulares são uma ferramenta poderosa que permeia o desenvolvimento JavaScript moderno; praticamente todos os aspectos de qualquer tipo de combinação dependem de seu uso. Com um bom entendimento dos conceitos avançados de regex abordados neste capítulo, você deve se sentir confortável para lidar com qualquer parte desafiadora de código que possa se beneficiar das expressões regulares.

- Podemos criar expressões regulares com literais de expressão regular (/ teste/) e com o RegExp construtor (novo RegExp (" teste")). Os literais são preferidos quando a regex é conhecida no tempo de desenvolvimento e o construtor quando a regex é construída no tempo de execução.

- Com cada expressão regular, podemos associar cinco sinalizadores: eu torna a regex insensível a maiúsculas e minúsculas, g corresponde a todas as instâncias do padrão, m permite correspondências em várias linhas, y permite a correspondência pegajosa, enquanto você permite o uso de escapes Unicode. Sinalizadores são adicionados no final de um literal regex: / test / ig, ou como um segundo
  parâmetro para o RegExp construtor: novo RegExp ("teste", "i").
- Use [] (como em [abc]) para especificar um conjunto de caracteres que desejamos corresponder.
- Use ^ para significar que o padrão deve aparecer no início de uma string e $ para significar que o padrão deve aparecer no final de uma string.
- Usar ? para especificar que um termo é opcional, + que um termo deve aparecer uma ou várias vezes e * para especificar que um termo aparece zero, uma ou várias vezes. Usar . para combinar com qualquer personagem.
-
- Podemos usar barra invertida (\) para escapar de caracteres regex especiais (como. [$ ^). Use parênteses ()
- para agrupar vários termos e barra vertical (|) para especificar a alternância.

- As partes de uma string que são correspondidas com sucesso aos termos podem ser referenciadas de volta com uma barra invertida seguida pelo número da captura (\ 1, \ 2, e assim por diante).

- Cada string tem acesso ao Combine , que recebe uma expressão regular e retorna uma matriz contendo toda a string correspondida junto com todas as capturas correspondidas. Também podemos usar o substituir função, que causa uma substituição em correspondências de padrão em vez de em uma string fixa.

10.8 Exercícios

1. Em JavaScript, as expressões regulares podem ser criadas com qual das seguintes opções?
   - a. Literais de expressão regular
   - b. O embutido RegExp constructor
   - c. The built-in Expressão regular constructor
2. Qual das opções a seguir é um literal de expressão regular?
   - a. / teste/
   - b. \ teste\
   - c. novo RegExp ("teste");
3. Escolha os sinalizadores de expressão regular corretos:
   - a. / teste / g
   - b. g / test /
   - c. novo RegExp ("teste", "gi");
4. A expressão regular / def / corresponde a qual das seguintes strings?
   - a. Uma das cordas d, e, f
   - b. def
   - c. de
5. A expressão regular / [^ abc]/ corresponde a qual das seguintes?
   - a. Uma das cordas a, b, c
   - b. Uma das cordas d, e, f
   - c. Corresponde à string ab
6. Qual das seguintes expressões regulares corresponde à string Olá?
   - a. / Olá/
   - b. / Olá/
   - c. / Olá/
   - d. / [Olá]/
7. A expressão regular / (cd) + (de) * / corresponde a qual das seguintes strings?
   - a. CD
   - b. de
   - c. cdde
   - d. cdcd
   - e. ce
   - f. cdcddedede

8 Em expressões regulares, podemos expressar alternativas com qual das seguintes?

uma #

b E

c |

9 Na expressão regular / ([0-9]) 2 /, podemos fazer referência ao primeiro dígito correspondido com qual dos seguintes?

uma / 0

b / 1

c \ 0

d \ 1

10 A expressão regular / ([0-5]) 6 \ 1 / corresponderá a qual dos seguintes?

uma 060

b 16

c 261

d 565

11 A expressão regular / (?: ninja) - (truque)? - \ 1 / irá corresponder a qual dos seguintes mugindo?

uma ninja-

b ninja-truque-ninja

c truque-truque-ninja

12 Qual é o resultado de executar " 012675 ".substituir (/ 0-5 / g," a ")?

uma aaa67a

b a12675

c a1267a

11 Código técnicas de modularização

Este capítulo cobre

- Usando o padrão de módulo
- Usando os padrões atuais para escrever código modular: AMD e CommonJS
- Trabalhando com módulos ES6

Até agora, exploramos os primitivos básicos do JavaScript, como funções, objetos, coleções e expressões regulares. Temos mais do que algumas ferramentas em nosso cinto para resolver problemas específicos com nosso código JavaScript. Mas, à medida que nossos aplicativos começam a crescer, outro conjunto de problemas, relacionados a como estruturamos e gerenciamos nosso código, começa a surgir. Repetidamente, foi provado que as bases de código grandes e monolíticas são muito mais difíceis de entender e manter do que as menores e bem organizadas. Portanto, é natural que uma maneira de melhorar a estrutura e a organização de nossos programas seja dividi-los em segmentos menores, relativamente fracamente acoplados, chamados *módulos.*

Módulos são unidades maiores de organização de nosso código do que objetos e funções; eles nos permitem dividir programas em clusters que pertencem juntos. Ao criar

módulos, devemos nos esforçar para formar abstrações consistentes e encapsular detalhes de implementação. Isso torna mais fácil raciocinar sobre nosso aplicativo, porque não nos importamos com vários detalhes frívolos ao usar a funcionalidade de nosso módulo. Além disso, ter módulos significa que podemos facilmente reutilizar a funcionalidade do módulo em diferentes partes de nossos aplicativos e até mesmo em diferentes aplicativos, acelerando significativamente nosso processo de desenvolvimento.

Como você viu anteriormente neste livro, o JavaScript é grande em variáveis globais: sempre que definimos uma variável no código da linha principal, essa variável é automaticamente considerada global e pode ser acessada de qualquer outra parte do nosso código. Isso pode não ser um problema para programas pequenos, mas à medida que nossos aplicativos começam a crescer e incluímos código de terceiros, a chance de ocorrer conflito de nomes começa a aumentar significativamente. Na maioria das outras linguagens de programação, esse problema é resolvido com namespaces (C ++ e C #) ou pacotes (Java), que envolvem todos os nomes incluídos em outro nome, reduzindo significativamente os conflitos potenciais.

Até o ES6, o JavaScript não oferecia um recurso integrado de nível superior que nos permitisse agrupar variáveis relacionadas em um módulo, namespace ou pacote. Portanto, para resolver esse problema, os programadores de JavaScript desenvolveram técnicas avançadas de modularização de código que tiram proveito de construções JavaScript existentes, como objetos, funções imediatas e encerramentos. Neste capítulo, exploraremos algumas dessas técnicas.

Felizmente, é apenas uma questão de tempo até que sejamos capazes de abandonar completamente essas técnicas alternativas, porque o ES6 finalmente introduz os módulos nativos. Infelizmente, os navegadores não pegaram, então vamos explorar como os módulos devem funcionar no ES6, embora não tenhamos uma implementação de navegador nativo específica para testá-los.

Vamos começar com técnicas de modularização que podemos usar hoje.

Você sabe?

Que mecanismo existente você usa para aproximar módulos em JavaScript pré-ES6?

Qual é a diferença entre o AMD e CommonJS especificações do módulo?

Usando ES6, quais duas instruções você precisaria para usar o tryThisOut () **função de um módulo chamado** teste **de dentro de outro módulo chamado** porquinho da índia ?

11.1 Modularizando o código em JavaScript pré-ES6

O JavaScript pré-ES6 tem apenas dois tipos de escopos: escopo global e escopo de função. Não tem algo intermediário, um namespace ou um módulo que nos permitiria agrupar certas funcionalidades. Para escrever código modular, os desenvolvedores de JavaScript são forçados a ser criativos com os recursos existentes da linguagem JavaScript.

Ao decidir quais recursos usar, devemos ter em mente que, no mínimo, cada sistema de módulo deve ser capaz de fazer o seguinte:

- *Defina uma interface* através do qual podemos acessar a funcionalidade oferecida pelo módulo.

- *Ocultar componentes internos do módulo* para que os usuários de nossos módulos não sejam sobrecarregados com uma série de detalhes de implementação sem importância. Além disso, ao ocultar os componentes internos do módulo, protegemos esses componentes internos do mundo externo, evitando, assim, modificações indesejadas que podem levar a todos os tipos de efeitos colaterais e bugs.

Nesta seção, veremos primeiro como criar módulos usando recursos JavaScript padrão que exploramos até agora no livro, recursos como objetos, fechamentos e funções imediatas. Continuaremos com essa veia de modularização explorando a Asynchronous Module Definition (AMD) e o CommonJS, os dois padrões de especificação de módulo mais populares, construídos em bases ligeiramente diferentes. Você aprenderá como definir módulos usando esses padrões, bem como seus prós e contras.

Mas vamos começar com algo para o qual já definimos o cenário nos capítulos anteriores.

11.1.1 Usando objetos, fechamentos e funções imediatas para especificar módulos

Vamos voltar aos nossos requisitos mínimos de sistema de módulo, *escondendo detalhes de implementação* e *definindo interfaces de módulo.* Agora pense em quais recursos da linguagem JavaScript podemos aproveitar para implementar esses requisitos mínimos:

- *Escondendo o interior do módulo* —Como já sabemos, chamar uma função JavaScript cria um novo escopo no qual podemos definir variáveis que são visíveis apenas na função atual. Portanto, uma opção para ocultar as partes internas do módulo é usar funções como módulos. Desta forma, todas as variáveis de função se tornam variáveis de módulo interno que estão ocultas do mundo externo.

- *Definindo interfaces de módulo* —Implementar os componentes internos do módulo por meio de variáveis de função significa que essas variáveis só podem ser acessadas dentro do módulo. Mas se nossos módulos devem ser usados a partir de outro código, temos que ser capazes de definir uma interface limpa por meio da qual podemos expor a funcionalidade oferecida pelo módulo. Uma maneira de conseguir isso é aproveitando objetos e fechos. A ideia é que, a partir de nosso módulo de função, retornemos um objeto que representa a interface pública de nosso módulo. Esse objeto deve conter métodos oferecidos pelo módulo, métodos que irão, por meio de fechamentos, manter vivas nossas variáveis de módulo internas, mesmo depois que nossa função de módulo terminar sua execução.

Agora que fornecemos uma descrição de alto nível de como implementar módulos em JavaScript, vamos examiná-lo lentamente, passo a passo, começando com o uso de funções para ocultar o interior do módulo.

FUNÇÕES COMO MÓDULOS

Chamar uma função cria um novo escopo que podemos usar para definir variáveis que não serão visíveis de fora da função atual. Vamos dar uma olhada no seguinte snippet de código que conta o número de cliques em uma página da web:

```
(function countClicks () {
    deixe numClicks = 0;
    document.addEventListener ("click", () => {
        alerta (++ numClicks);
    });
}) ();
```

Define uma variável local que armazenará contagens de cliques

Sempre que um usuário clica, o contador é incrementado e o valor atual é relatado.

Neste exemplo, criamos uma função chamada countClicks isso cria uma variável numClicks e registra um manipulador de eventos de clique em todo o documento. Sempre que um clique é feito, o numClicks variável é incrementada e o resultado é exibido ao usuário por meio de uma caixa de alerta. Há duas coisas importantes a serem observadas aqui:

- O numClicks variável, interna ao countClicks função, é mantida ativa através do encerramento da função de tratamento de clique. A variável pode ser referenciada apenas dentro do manipulador, e *em nenhum outro lugar!* Nós protegemos o numClicks
  variável do código fora do countClicks função. Ao mesmo tempo, não poluímos o namespace global do nosso programa com uma variável que provavelmente não é tão importante para o resto do nosso código.

- Nosso countClicks função é chamada apenas neste lugar, portanto, em vez de definir uma função e, em seguida, chamá-la em uma instrução separada, usamos uma função imediata, ou um IIFE (apresentado no capítulo 3), para definir e invocar imediatamente o countClicks função.

Também podemos dar uma olhada no estado atual do aplicativo, com relação a como nossa variável de função interna (ou módulo) é mantida viva por meio de fechamentos, conforme mostrado na Figura 11.1.

Agora que entendemos como ocultar os detalhes do módulo interno e como os fechamentos podem manter esses detalhes internos vivos pelo tempo que for necessário, vamos passar ao nosso segundo requisito mínimo para módulos: definir interfaces de módulo.

O PADRÃO DO MÓDULO: FUNÇÕES DE AUMENTAÇÃO COMO MÓDULOS COM OBJETOS COMO INTERFACES

A interface do módulo é geralmente composta de um conjunto de variáveis e funções que nosso módulo fornece para o mundo exterior. A maneira mais fácil de criar tal interface é usar o humilde objeto JavaScript.

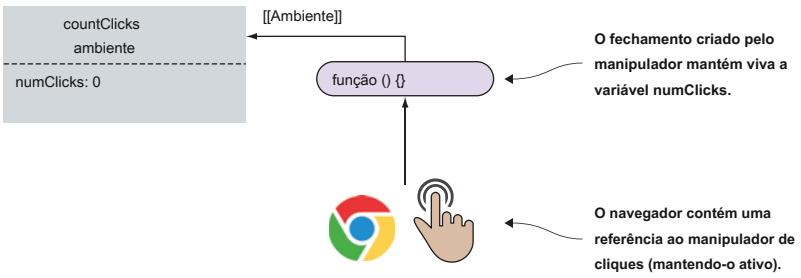
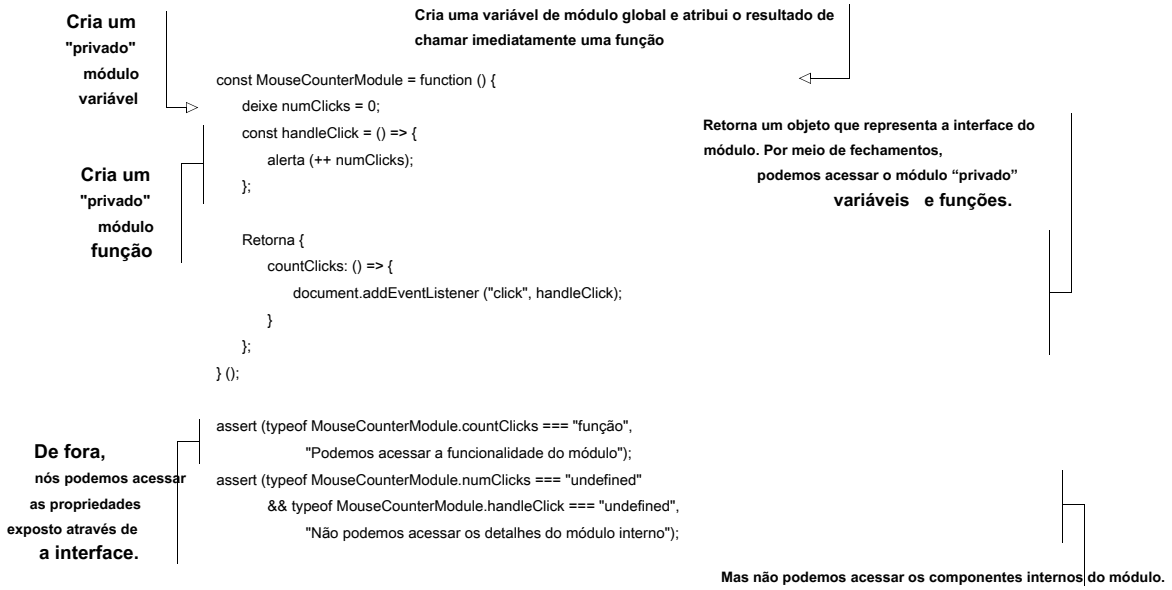


Figura 11.1 O manipulador de eventos click, por meio de fechamentos, mantém vivo o local **numClicks** variável.

Por exemplo, vamos criar uma interface para nosso módulo que conta os cliques em nossa página web, conforme mostrado na listagem a seguir.

Listagem 11.1 O padrão do módulo



Aqui, usamos uma função imediata para implementar um módulo. Dentro da função imediata, definimos nossos detalhes de implementação do módulo interno: uma variável local, numClicks, e uma função local, handleClick, que são acessíveis apenas dentro do módulo. Em seguida, criamos e retornamos imediatamente um objeto que servirá como a "interface pública" do módulo. Esta interface contém um countClicks método que podemos usar de fora do módulo para acessar a funcionalidade do módulo.

Ao mesmo tempo, como expusemos uma interface de módulo, os detalhes de nosso módulo interno são mantidos vivos por meio de fechamentos criados pela interface. Por exemplo, neste caso, o countClicks método da interface mantém variáveis do módulo interno vivas numClicks e handleClick, conforme mostrado na figura 11.2.

Por fim, armazenamos o objeto que representa a interface do módulo, retornado pela função imediata, em uma variável chamada MouseCounterModule, por meio do qual podemos consumir facilmente a funcionalidade do módulo, escrevendo o seguinte código:

```
MouseCounterModule.countClicks ()
```

E é basicamente isso.

Aproveitando as funções imediatas, podemos ocultar certos detalhes de implementação do módulo. Então, adicionando objetos e fechamentos à mistura, podemos especificar uma interface de módulo que expõe a funcionalidade fornecida por nosso módulo para o mundo externo.

```
const MouseCounterModule = função () {
    deixei numClicks = 0;
    const handleClick = () => {
        alerta (++ numClicks);
    };

    Retorna {
        countClicks: () => {
            document.addEventListener ("click", handleClick);
        }
    };
} ();
```

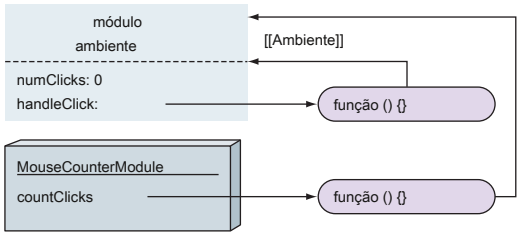


Figura 11.2 Exponha o interface pública de um módulo por meio de um objeto retornado. A implementação do módulo interno ("privado" variables e funções) é mantida ativa por meio de fechamentos criados por métodos de interface pública.

Este padrão de usar funções imediatas, objetos e encerramentos para criar módulos em JavaScript é chamado de *padrão de módulo*. Foi popularizado por Douglas Crockford e foi uma das primeiras formas massivamente populares de modularizar o código JavaScript.

Assim que tivermos a capacidade de definir módulos, é sempre bom poder dividi-los em vários arquivos (para gerenciá-los mais facilmente), ou ser capaz de definir funcionalidades adicionais em módulos existentes, sem modificar seu código-fonte.

Vamos ver como isso pode ser feito.

MÓDULOS DE AUMENTAÇÃO

Vamos aumentar nosso MouseCounterModule do exemplo anterior com um recurso adicional de contagem do número de rolagens do mouse, mas sem modificar o original MouseCounterModule código. Veja a lista a seguir.

Listagem 11.2 Módulos de aumento

```
const MouseCounterModule = function () {
    deixe numClicks = 0;
    const handleClick = () => {
        alerta (++ numClicks);
    };

    Retorna {
        countClicks: () => {
            document.addEventListener ("click", handleClick);
        }
    };
} ();

(função (módulo) {
```

O original MouseCounterModule

Invoca imediatamente uma função que aceita o módulo que queremos estender como argumento

```
    deixe numScrolls = 0;
    const handleScroll = () => {
        alerta (++ numScrolls);
    }

    module.countScrolls = () => {
        document.addEventListener ("wheel", handleScroll);
    };
}) (MouseCounterModule);

assert (typeof MouseCounterModule.countClicks === "função",
            "Podemos acessar a funcionalidade do módulo inicial");

assert (typeof MouseCounterModule.countScrolls === "função",
            "Podemos acessar a funcionalidade do módulo aumentado");
```

Define novo variáveis privadas e funções

Passa no módulo como um argumento

Estende o interface do módulo

Ao aumentar um módulo, geralmente seguimos um procedimento semelhante à criação de um novo módulo. Chamamos imediatamente uma função, mas desta vez, passamos a ela o módulo que queremos estender como argumento:

```
(função (módulo) {
    . . .
    módulo de retorno;
}) (MouseCounterModule);
```

Dentro da função, realizamos nosso trabalho e criamos todas as variáveis e funções privadas necessárias. Neste caso, definimos uma variável privada e uma função privada para contar e relatar o número de pergaminhos:

```
deixe numScrolls = 0;
const handleScroll = () => {
    alerta (++ numScrolls);
}
```

Finalmente, estendemos nosso módulo, disponível através da função imediata módulo parâmetro, da mesma forma que estenderíamos qualquer outro objeto:

```
module.countScrolls = () => {
    document.addEventListener ("wheel", handleScroll);
};
```

Depois de realizar esta operação simples, nosso MouseCounterModule também pode countScrolls.

Nossa interface de módulo público agora tem dois métodos e podemos usar o módulo das seguintes maneiras:

```
MouseCounterModule.countClicks ();
MouseCounterModule.countScrolls ();
```

Um método que faz parte da interface do módulo desde o início

Um novo método de módulo que adicionamos ao estender o módulo

Como já mencionamos, estendemos o módulo de uma maneira semelhante à criação de um novo módulo, por meio de uma função imediatamente chamada que estende o módulo. Isso tem alguns efeitos colaterais interessantes em termos de encerramentos, então vamos dar uma olhada mais de perto no estado do aplicativo depois de aumentar o módulo, conforme mostrado na Figura 11.3.

Se você olhar com atenção, a figura 11.3 também mostra uma das deficiências do padrão de módulo: a incapacidade de compartilhar variáveis de módulo privadas entre extensões de módulo. Por exemplo, o countClicks função mantém um fechamento em torno do numClicks e handleClick variáveis, e poderíamos acessar essas partes internas do módulo privado através a countClicks método.

```
const MouseCounterModule = função () {
    deixei numClicks = 0;
    const handleClick = () => {
        alerta (++ numClicks);
    };
    Retorna {
        countClicks: () => {
            document.addEventListener ("click", handleClick);
        }
    };
} ();

( função (módulo){
    deixei numScrolls = 0;
    const handleScroll = () => {
        alerta (++ numScrolls);
    }

    module.countScrolls = () => {
        document.addEventListener ("roda", handleClick);
    };
}) (MouseCounterModule);
```

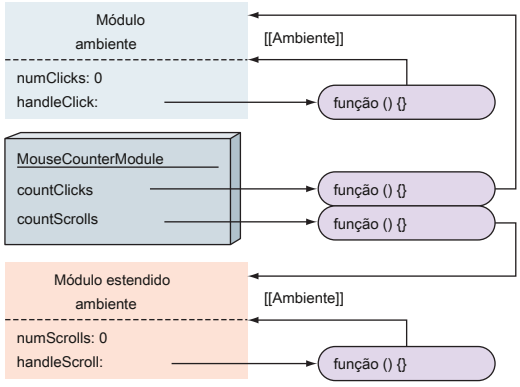


Figura 11.3 Ao aumentar um módulo, estendemos sua interface externa com novas funcionalidades, geralmente passando o módulo para outra função imediata. Neste exemplo, adicionamos a capacidade de **countScrolls** para nosso **MouseCounterModule.** Observe que duas funções separadas são definidas em ambientes diferentes e não podem acessar as variáveis internas uma da outra.

Infelizmente, nossa extensão, o countScrolls função, é criado em um escopo completamente separado, com um conjunto completamente novo de variáveis privadas: numScrolls e handleScroll. O countScrolls função cria um fechamento apenas em torno numScrolls e handleScroll variáveis e, portanto, não pode acessar o numClicks e handleClick variáveis.

> NOTA As extensões de módulo, quando executadas por meio de funções imediatas separadas, não podem compartilhar internos de módulo privados, porque cada chamada de função cria um novo escopo. Embora isso seja uma deficiência, não é um obstáculo e ainda podemos usar o padrão de módulo para manter nossos aplicativos JavaScript modulares.

Observe que, no padrão de módulo, os módulos são objetos como qualquer outro e podemos estendê-los da maneira que acharmos apropriada. Por exemplo, podemos adicionar novas funcionalidades estendendo o objeto de módulo com novas propriedades:

```
MouseCounterModule.newMethod = () => {...}
```

Também podemos usar o mesmo princípio para criar submódulos facilmente:

```
MouseCounterModule.newSubmodule = () => {
    Retorna {...};
} ();
```

Observe que todas essas abordagens sofrem da mesma deficiência fundamental do padrão do módulo: as extensões subsequentes do módulo não podem acessar os detalhes internos do módulo definidos anteriormente.

Infelizmente, existem mais problemas com o padrão do módulo. Quando começamos a construir aplicativos modulares, os próprios módulos muitas vezes dependem de outros módulos para sua funcionalidade. Infelizmente, o padrão do módulo não cobre o gerenciamento dessas dependências. Nós, como desenvolvedores, temos que cuidar da ordem de dependência certa para que nosso código de módulo tenha tudo de que precisa para ser executado. Embora isso não seja um problema em aplicativos pequenos e médios, pode apresentar problemas sérios em aplicativos grandes que usam muitos módulos interdependentes.

Para lidar com esses problemas, surgiram alguns padrões concorrentes, a saber, Asynchronous Module Definition (AMD) e CommonJS.

11.1.2 Modularizando aplicativos JavaScript com AMD e CommonJS

AMD e CommonJS são padrões concorrentes de especificação de módulo que nos permitem especificar módulos JavaScript. Além de algumas diferenças na sintaxe e filosofia, a principal diferença é que o AMD foi projetado explicitamente com o navegador em mente, enquanto o CommonJS foi projetado para um ambiente JavaScript de uso geral (como servidores, com Node.js), sem estar vinculado às limitações de o navegador. Esta seção fornece uma visão geral relativamente curta das especificações do módulo; configurando-os e incluindo

em seus projetos está além do escopo deste livro. Para mais informações, recomendamos *Design de aplicativo JavaScript* por Nicolas G. Bevacqua (Manning, 2015).

AMD

AMD cresceu a partir do kit de ferramentas Dojo (https://dojotoolkit.org/), um dos kits de ferramentas JavaScript populares para construir aplicativos da Web do lado do cliente. A AMD nos permite especificar facilmente os módulos e suas dependências. Ao mesmo tempo, foi desenvolvido do zero para o navegador. Atualmente, a implementação AMD mais popular é RequireJS (http://requirejs.org/)

Vamos ver um exemplo de definição de um pequeno módulo que tem uma dependência do jQuery.

Listagem 11.3 Usando AMD para especificar um módulo dependente de jQuery

```
definir(' MouseCounterModule ', [' jQuery '], $ => {
    deixe numClicks = 0;
    const handleClick = () => {
        alerta (++ numClicks);
    };

    Retorna {
        countClicks: () => {
            $ (document) .on ("click", handleClick);
        }
    };
});
```

Usa a função define para especificar um módulo, suas dependências e a função de fábrica do módulo que criará o módulo

A interface pública do nosso módulo

AMD fornece uma função chamada definir que aceita o seguinte:

- O ID do módulo recém-criado. Este ID pode ser usado posteriormente para solicitar o módulo de outras partes do nosso sistema.
- Uma lista de IDs de módulo dos quais nosso módulo atual depende (os módulos necessários).
- Uma função de fábrica que irá inicializar o módulo e que aceita os módulos necessários como argumentos.

Neste exemplo, usamos definir função para criar um módulo com um ID MouseCounterModule isso depende do jQuery. Por causa dessa dependência, a AMD primeiro solicita o módulo jQuery, o que pode levar algum tempo se o arquivo tiver que ser solicitado de um servidor. Esta ação é realizada de forma assíncrona, a fim de evitar o bloqueio. Depois que todas as dependências foram baixadas e avaliadas, a função de fábrica do módulo é chamada com um argumento para cada módulo solicitado. Nesse caso, haverá um argumento, porque nosso novo módulo requer apenas jQuery. Dentro da função de fábrica, criamos nosso módulo da mesma forma que faríamos com o padrão de módulo padrão: retornando um objeto que expõe a interface pública do módulo.

Como você pode ver, a AMD oferece vários benefícios interessantes, como estes:

- Resolução automática de dependências, para que não tenhamos que pensar na ordem em que incluímos nossos módulos.
- Os módulos podem ser carregados de forma assíncrona, evitando assim o bloqueio. Vários
- módulos podem ser definidos em um arquivo.

Agora que você tem uma ideia básica de como o AMD funciona, vamos dar uma olhada em outro padrão de definição de módulo muito popular.

COMMONJS

Enquanto o AMD foi desenvolvido explicitamente para o navegador, CommonJS é uma especificação de módulo projetada para um ambiente JavaScript de propósito geral. Atualmente, ele tem o maior número de seguidores na comunidade Node.js.

CommonJS usa módulos baseados em arquivo, portanto, podemos especificar um módulo por arquivo. Para cada módulo, CommonJS expõe uma variável, módulo, com uma propriedade, exportações, que podemos facilmente estender com propriedades adicionais. No final, o conteúdo de module.exports é exposta como a interface pública do módulo.

Se quisermos usar um módulo de outras partes do aplicativo, podemos solicitá-lo. O arquivo será carregado de forma síncrona e teremos acesso à sua interface pública. Esta é a razão pela qual CommonJS é muito mais popular no servidor, onde a busca do módulo é relativamente rápida porque requer apenas uma leitura do sistema de arquivos, do que no cliente, onde o módulo deve ser baixado de um servidor remoto e onde sincronizado carregar geralmente significa bloqueio.

Vejamos um exemplo que define nossa recorrência MouseCounterModule, desta vez em CommonJS.

Listagem 11.4 Usando CommonJS para definir um módulo

```
//MouseCounterModule.js
const $ = require ("jQuery"); deixe numClicks = 0;

const handleClick = () => {
    alerta (++ numClicks);
};

module.exports = {
    countClicks: () => {
        $ (document) .on ("click", handleClick);
    }
};
```

Requer sincronicamente um módulo jQuery

Modifica a propriedade module.exports para especificar a interface pública de um módulo

Para incluir nosso módulo em um arquivo diferente, podemos escrever isto:

```
const MouseCounterModule = require ("MouseCounterModule.js"); MouseCounterModule.countClicks ();
```

Veja como isso é simples?

Como a filosofia do CommonJS determina um módulo por arquivo, qualquer código que colocarmos em um módulo de arquivo fará parte desse módulo. Portanto, não há necessidade de agrupar variáveis em funções imediatas. Todas as variáveis definidas em um módulo estão seguramente contidas no escopo do módulo atual e não vazam para o escopo global. Por exemplo, todas as três variáveis do nosso módulo ($, numClicks, e

handleClick) têm escopo de módulo, embora sejam definidos no código de nível superior (fora de todas as funções e blocos), o que os tornaria tecnicamente variáveis globais em arquivos JavaScript padrão.

Mais uma vez, é importante observar que apenas as variáveis e funções expostas por meio do module.exports objeto estão disponíveis fora do módulo. O procedimento é semelhante ao padrão de módulo, só que em vez de retornar um objeto completamente novo, o ambiente já fornece um que podemos estender com nossos métodos e propriedades de interface.

O CommonJS tem algumas vantagens:

- Possui sintaxe simples. Precisamos especificar apenas o module.exports propriedades, enquanto o resto do código do módulo permanece praticamente o mesmo, como se estivéssemos escrevendo JavaScript padrão. Requerer módulos também é simples; nós apenas usamos o
  exigir função.
- CommonJS é o formato de módulo padrão para Node.js, portanto, temos acesso a milhares de pacotes que estão disponíveis por meio do npm, o gerenciador de pacotes do nó.

A maior desvantagem do CommonJS é que ele não foi criado explicitamente com o navegador em mente. Em JavaScript no navegador, não há suporte para o módulo variável e o exportar propriedade; temos que empacotar nossos módulos CommonJS em um formato legível por navegador. Podemos conseguir isso com o Browserify (http://browserify.org/) ou RequireJS (http://requirejs.org/docs/commonjs.html)

Ter dois padrões concorrentes para especificar módulos, AMD e CommonJS, levou a uma daquelas situações em que as pessoas tendem a se dividir em dois, às vezes até mesmo campos opostos. Se você trabalha em projetos relativamente fechados, isso pode não ser um problema; você escolhe o padrão que melhor se adapta a você. Podem surgir problemas, no entanto, quando precisamos reutilizar o código do campo oposto e somos forçados a saltar por todos os tipos de obstáculos. Uma solução é usar a definição de módulo universal, ou UMD (https://github.com/u), um padrão com uma sintaxe um tanto complicada que permite que o mesmo arquivo seja usado por AMD e CommonJS. Isso está além do escopo deste livro, mas se você estiver interessado, muitos recursos de qualidade estão disponíveis online.

Felizmente, o comitê ECMAScript reconheceu a necessidade de uma sintaxe de módulo unificada com suporte em todos os ambientes JavaScript, então o ES6 define um novo padrão de módulo que deve finalmente colocar essas diferenças de lado.

11.2 Módulos ES6

Os módulos ES6 são projetados para combinar as vantagens de CommonJS e AMD:

- Semelhante ao CommonJS, os módulos ES6 têm uma sintaxe relativamente simples e os módulos ES6 são baseados em arquivo (um módulo por arquivo).
- Semelhante ao AMD, os módulos ES6 fornecem suporte para carregamento de módulo assíncrono.

NOTA Os módulos integrados fazem parte do padrão ES6. Como você verá em breve, a sintaxe do módulo ES6 inclui semântica e palavras-chave adicionais (como o exportar e importar palavras-chave) que não são compatíveis com os navegadores atuais. Se quisermos usar módulos hoje, temos que transpilar nosso código de módulo com Traceur (https://github.com/google/traceur-compiler), Babel (http: //babeljs.io/) ou TypeScript (www.typescriptlang.org/) Também podemos usar a biblioteca SystemJS (https://github.com/systemjs/systemjs), que fornece suporte para carregar todos os padrões de módulo disponíveis atualmente: AMD, CommonJS e até módulos ES6. Você pode encontrar instruções sobre como usar SystemJS no repositório do projeto (https://github.com/systemjs/systemjs)

A ideia principal por trás dos módulos ES6 é que apenas os identificadores explicitamente exportados de um módulo são acessíveis de fora desse módulo. Todos os outros identificadores, mesmo aqueles definidos no escopo de nível superior (o que seria o escopo global em JavaScript padrão), são acessíveis apenas de dentro do módulo. Isso foi inspirado por CommonJS.

Para fornecer essa funcionalidade, o ES6 apresenta duas novas palavras-chave:

- exportar —Para tornar certos identificadores disponíveis de fora do módulo
- importar —Para importar identificadores de módulo exportados

A sintaxe para exportar e importar a funcionalidade do módulo é simples, mas tem muitas nuances sutis que exploraremos lentamente, passo a passo.

11.2.1 Funcionalidade de exportação e importação

Vamos começar com um exemplo simples que mostra como exportar funcionalidade de um módulo e importá-lo para outro.

Listagem 11.5 Exportando de um módulo Ninja.js

```
const ninja = "Yoshi";
export const message = "Olá";

função de exportação sayHiToNinja () {
    mensagem de retorno + "" + ninja;
}
```

Define uma variável de nível superior em um módulo

Define uma variável e uma função e as exporta do módulo com a palavra-chave export

Acessa uma variável do módulo interno da API pública do módulo

Primeiro definimos uma variável, ninja, uma variável de módulo que estará acessível apenas dentro deste módulo, embora seja colocada no código de nível superior (o que a tornaria uma variável global no código pré-ES6).

Em seguida, definimos outra variável de nível superior, mensagem, que tornamos acessível de fora do módulo usando o novo exportar palavra-chave. Por fim, também criamos e exportamos o diga HiToNinja função.

E é isso! Esta é a sintaxe mínima que precisamos saber para definir nossos próprios módulos. Não precisamos usar funções imediatas ou lembrar de qualquer sintaxe esotérica para exportar a funcionalidade de um módulo. Escrevemos nosso código como escreveríamos código JavaScript padrão, com a única diferença de que prefixamos alguns dos identificadores (como variáveis, funções ou classes) com um exportar palavra-chave.

Antes de aprender a importar essa funcionalidade exportada, vamos dar uma olhada em uma forma alternativa de exportar identificadores: Listamos tudo o que queremos exportar no final do módulo, conforme mostrado na lista a seguir.

Listagem 11.6 Exportando no final de um módulo

```
const ninja = "Yoshi"; const mensagem =
"Olá";

function sayHiToNinja () {
    mensagem de retorno + "" + ninja;
}

exportar {mensagem, digaHiToNinja};
```

Especifica todos os identificadores de módulo

Exporta alguns dos identificadores de módulo

Esta forma de exportar identificadores de módulo tem alguma semelhança com o padrão do módulo, pois uma função imediata retorna um objeto que representa a interface pública do nosso módulo, e especialmente para CommonJS, conforme expandimos o module.exports objeto com a interface do módulo público.

Independentemente de como exportamos os identificadores de um determinado módulo, se precisarmos importá-los para outro módulo, temos que usar o importar palavra-chave, como no exemplo a seguir.

Listagem 11.7 Importando do módulo Ninja.js

```
importar {mensagem, digaHiToNinja} de "Ninja.js";

assert (mensagem === "Olá",
        "Podemos acessar a variável importada"); assert (sayHiToNinja () === "Olá
Yoshi",
        "Podemos dizer olá ao Yoshi de fora do módulo");

assert (typeof ninja === "undefined",
        "Mas não podemos acessar Yoshi diretamente");
```

Usa a palavra-chave import para importar uma ligação de identificador de um módulo

Agora podemos acessar a variável importada e chamar a função importada.

Não podemos acessar variáveis de módulo não exportadas diretamente.

Usamos o novo importar palavra-chave para importar uma variável, mensagem e uma função, diga HiToNinja de ninja módulo:

```
importar {mensagem, digaHiToNinja} de "Ninja.js";
```

Fazendo isso, ganhamos acesso a esses dois identificadores definidos no ninja módulo. Finalmente, podemos testar se podemos acessar o mensagem variável e chame o diga HiToNinja função:

```
assert (mensagem === "Olá",
        “Podemos acessar a variável importada”); assert (sayHiToNinja () === "Olá
Yoshi",
        "Podemos dizer olá ao Yoshi de fora do módulo");
```

O que não podemos fazer é acessar as variáveis não exportadas e não importadas. Por exemplo, não podemos acessar o ninja variável porque não está marcada com exportar:

```
assert (typeof ninja === "undefined",
        "Mas não podemos acessar Yoshi diretamente");
```

Com os módulos, estamos finalmente um pouco mais seguros do uso indevido de variáveis globais. Tudo o que não marcamos explicitamente para exportação fica bem isolado dentro de um módulo.

Neste exemplo, usamos um *exportação nomeada,* o que nos permite exportar vários identificadores de um módulo (como fizemos com mensagem e digaHiToNinja). Como podemos exportar um grande número de identificadores, listar todos eles em uma instrução de importação pode ser entediante. Portanto, uma notação abreviada nos permite trazer todos os identificadores exportados de um módulo, conforme mostrado na lista a seguir.

Listagem 11.8 Importando todas as exportações nomeadas do módulo Ninja.js

```
importar * como ninjaModule de "Ninja.js";

assert (ninjaModule.message === "Olá",
        “Podemos acessar a variável importada”);
assert (ninjaModule.sayHiToNinja () === "Olá Yoshi",
        "Podemos dizer olá ao Yoshi de fora do módulo");

assert (typeof ninjaModule.ninja === "undefined",
        "Mas não podemos acessar Yoshi diretamente");
```

Usa * notação para importar todos os identificadores exportados

Refere-se às exportações nomeadas por meio de notação de propriedade

Ainda não conseguimos acessar identificadores não exportados.

Como mostra a listagem 11.8, para importar todos os identificadores exportados de um módulo, usamos o importar * notação em combinação com um identificador que usaremos para se referir a todo o módulo (neste caso, o ninjaModule identificador). Depois de fazer isso, podemos acessar os identificadores exportados por meio da notação de propriedade; por exemplo, ninjaModule.message, ninjaModule.sayHiToNinja. Observe que ainda não podemos acessar variáveis de nível superior que não foram exportadas, como é o caso com o ninja variável.

EXPORTAÇÕES PADRÃO

Freqüentemente, não queremos exportar um conjunto de identificadores relacionados de um módulo, mas sim representar o módulo inteiro por meio de uma única exportação. Uma situação bastante comum em que isso ocorre é quando nossos módulos contêm uma única classe, como na listagem a seguir.

Listagem 11.9 Uma exportação padrão de Ninja.js

```
exportação padrão class Ninja {
    construtor (nome) {
        this.name = nome;
    }
}

função de exportação compareNinjas (ninja1, ninja2) {
    return ninja1.name === ninja2.name;
}
```

Usa as palavras-chave padrão de exportação para especificar a ligação do módulo padrão

Ainda podemos usar exportações nomeadas junto com a exportação padrão.

Aqui nós adicionamos o predefinição palavra-chave após o exportar palavra-chave, que especifica a ligação padrão para este módulo. Neste caso, a ligação padrão para este módulo é a classe chamada Ninja. Mesmo que tenhamos especificado uma ligação padrão, ainda podemos usar exportações nomeadas para exportar identificadores adicionais, como fizemos com o compareNinjas função.

Agora, podemos usar uma sintaxe simplificada para importar funcionalidades do Ninja.js, conforme mostrado na lista a seguir.

Listagem 11.10 Importando uma exportação padrão

Ao importar uma exportação padrão, não há precisamos de colchetes, e podemos usar qualquer nome que queremos.

```
importar ImportedNinja de "Ninja.js";
importar {compareNinjas} de "Ninja.js";

const ninja1 = novo ImportedNinja ("Yoshi"); const ninja2 = novo ImportedNinja
("Hattori");

afirmar (ninja1! == indefinido
        && ninja2! == undefined, "Podemos criar alguns Ninjas");

assert (! compareNinjas (ninja1, ninja2),
            "Podemos comparar ninjas");
```

Ainda podemos importar exportações nomeadas.

Cria um casal de ninjas, e testa que eles existir

Também podemos acessar as exportações nomeadas.

Começamos este exemplo importando uma exportação padrão. Para isso, usamos uma sintaxe de importação menos confusa, eliminando as chaves que são obrigatórias para importar exportações nomeadas. Além disso, observe que podemos escolher um nome arbitrário para se referir à exportação padrão; não estamos obrigados a usar o que usamos na exportação. Neste exemplo,
ImportedNinja refere-se a Ninja classe definida no arquivo Ninja.js.

Continuamos o exemplo importando uma exportação nomeada, como nos exemplos anteriores, apenas para ilustrar que podemos ter uma exportação padrão e uma série de exportações nomeadas em um único módulo. Por fim, instanciamos alguns ninja objetos e chamar o compareNinjas função, para confirmar que todas as importações funcionam como deveriam.

Nesse caso, ambas as importações são feitas a partir do mesmo arquivo. ES6 oferece uma sintaxe abreviada:

```
import ImportedNinja, {compareNinjas} de "Ninja.js";
```

Aqui, usamos o operador vírgula para importar as exportações padrão e nomeadas do arquivo Ninja.js, em uma única instrução.

RENAMING EXPORTS AND IMPORTS

Se necessário, também podemos renomear exportações e importações. Vamos começar renomeando as exportações, conforme mostrado no código a seguir (os comentários indicam em qual arquivo o código está localizado):

```
// ************* Greetings.js ************ /
function sayHi () {
    retornar "Olá";
}

assert (typeof sayHi === "função"
        && typeof sayHello === "undefined",
            "Dentro do módulo podemos acessar apenas sayHi");

export { diga olá como diga olá}

// ************* main.js ****************** /
import {sayHello} de "Greetings.js";

assert (typeof sayHi === "undefined"
        && typeof sayHello === "função",
            “Ao importar, só podemos acessar o alias”);
```

Define uma função chamado sayHi

Testa se podemos acessar apenas a função sayHi, mas não o alias!

Fornece um alias de identificador com a palavra-chave as

Ao importar, apenas dizer olá alias está disponível.

No exemplo anterior, definimos uma função chamada diga oi, e testamos se podemos acessar a função apenas embora o diga oi identificador, e não através do diga olá
alias que fornecemos no final do módulo por meio do como palavra-chave:

```
exportar {sayHi as sayHello}
```

Podemos realizar uma renomeação de exportação apenas neste formulário de exportação, e não prefixando a variável ou declaração de função com o exportar palavra-chave.

Então, quando realizamos uma importação da exportação renomeada, referenciamos a importação por meio do alias fornecido:

```
import {sayHello} de "Greetings.js";
```

Finalmente, testamos se temos acesso ao identificador com alias, mas não ao original:

```
assert (typeof sayHi === "undefined"
        && typeof sayHello === "função",
            “Ao importar, só podemos acessar o alias”);
```

A situação é semelhante ao renomear importações, conforme mostrado no seguinte segmento de código:

```
/ ************ Hello.js ************ /
função de exportação greet () {
    retornar "Olá";
}

/ ************ Salute.js ************ /
função de exportação greet () {
    retornar "Saudação";
}

/ ************ main.js ************ /
import { cumprimente dizendo olá} de "Hello.js"; import { cumprimentar como
saudação} de "Salute.js";

assert (typeof greet === "undefined",
        "Não podemos acessar saudar");

assert (sayHello () === "Hello" && salute () === "Salute",
        "Podemos acessar identificadores com alias!");
```

Exporta uma função com o nome greet do módulo Hello.js.

Exporta uma função com o mesmo nome saudação de Salute.js

Usa a palavra-chave as para importações de alias, evitando conflitos de nome

Não podemos acessar o nome da função original.

Mas podemos acessar os aliases.

Da mesma forma que exportar identificadores, também podemos usar o como palavra-chave para criar aliases ao importar identificadores de outros módulos. Isso é útil quando precisamos fornecer um nome melhor que seja mais adequado ao contexto atual, ou quando queremos evitar conflitos de nomenclatura, como é o caso neste pequeno exemplo.

Com isso, concluímos nossa exploração da sintaxe dos módulos ES6, que é recapitulada na tabela 11.1.

Tabela 11.1 Visão geral da sintaxe do módulo ES6

Código	Significado
export const ninja = "Yoshi"; função de exportação compare () {} exportar classe Ninja {}	Exporte uma variável nomeada. Exporte uma função nomeada. Exporte uma classe nomeada.
exportar a classe padrão Ninja {} exportar a função padrão Ninja () {}	Exporte a exportação de classe padrão. Exporte a exportação de função padrão.
const ninja = "Yoshi"; função compare () {}; exportar {ninja, comparar}; exportar {ninja como samurai, compare};	Exporte variáveis existentes. Exporte uma variável por meio de um novo nome.
importar Ninja de "Ninja.js"; importar {ninja, Ninja} de "Ninja.js";	Importe uma exportação padrão. Importar exportações nomeadas.
importar * como Ninja de "Ninja.js"; importar {ninja como iNinja} de "Ninja.js";	Importe todas as exportações nomeadas de um módulo. Importe uma exportação nomeada por meio de um novo nome.

11.3 Resumo

- Bases de código grandes e monolíticas têm muito mais probabilidade de serem difíceis de entender e manter do que as menores e bem organizadas. Uma forma de melhorar a estrutura e a organização de nossos programas é dividi-los em segmentos ou módulos menores e relativamente fracamente acoplados.

- Módulos são unidades maiores de código de organização do que objetos e funções e nos permitem dividir programas em clusters que pertencem uns aos outros.
- Em geral, os módulos promovem a compreensibilidade, facilitam a manutenção e melhoram a capacidade de reutilização do código.
- O JavaScript pré-ES6 não tem módulos embutidos e os desenvolvedores precisam ser criativos com os recursos de linguagem existentes para permitir a modularização do código. Uma das maneiras mais populares de criar módulos é combinando funções imediatamente invocadas com fechamentos.

  - As funções imediatas são usadas porque criam um novo escopo para definir variáveis de módulo que não são visíveis de fora desse escopo.

  - Os fechamentos são usados porque nos permitem manter as variáveis do módulo vivas. O padrão
  - mais popular é o padrão de módulo, que geralmente combina uma função imediata com o retorno de um novo objeto que representa a interface pública do módulo.

- Além do padrão de módulo, existem dois padrões de módulo populares: Definição de Módulo Assíncrono, projetado para habilitar módulos no navegador; e CommonJS, que é mais popular em JavaScript do lado do servidor.
  - A AMD pode resolver dependências automaticamente e os módulos são carregados de forma assíncrona, evitando assim o bloqueio.

  - CommonJS tem uma sintaxe simples, carrega módulos de forma síncrona (e, portanto, é mais apropriado para o servidor) e tem muitos pacotes disponíveis através do gerenciador de pacotes do nó (npm).

- Os módulos ES6 são projetados para levar em consideração os recursos do AMD e CommonJS. Esses módulos têm uma sintaxe simples influenciada pelo CommonJS e fornecem carregamento de módulo assíncrono como no AMD.
  - Módulos ES6 são baseados em arquivo, um módulo por arquivo.
  - Exportamos identificadores para que possam ser referenciados por outros módulos usando o novo exportar palavra-chave.

  - Importamos identificadores exportados de outros módulos usando o importar palavra-chave.

  - Um módulo pode ter um único predefinição exportação, que usamos se quisermos representar todo o módulo por meio de uma única exportação.
  - Tanto as importações quanto as exportações podem ser renomeadas com o como palavra-chave.

11.4 Exercícios

1 Qual mecanismo habilita variáveis de módulo privadas no padrão de módulo?

uma Protótipos

b Fechamentos

c Promessas

2 No código a seguir que usa módulos ES6, quais identificadores podem ser acessados se o módulo for importado?

```
const spy = "Yagyu"; comando de
função () {
    retorno geral + "ordena que você faça guerra!";
}
export const general = "Minamoto";
```

uma espião

b comando

c em geral

3 No código a seguir que usa módulos ES6, quais identificadores podem ser acessados quando o módulo é importado?

```
const ninja = "Yagyu"; comando de função
() {
    retorno geral + "ordena que você faça guerra!";
}
const general = "Minamoto";

exportar {ninja como espião};
```

uma espião

b comando

c em geral

d ninja

4 Quais das seguintes importações são permitidas?

```
// Arquivo: staff.js
const ninja = "Yagyu"; comando de função
() {
    retorno geral + "ordena que você faça guerra!";
}
const general = "Minamoto";

exportar {ninja como espião};
```

uma import {ninja, spy, general} de "staff.js"

b import * as Personnel de "staff.js"

c importar {spy} de "staff.js"

5 Se tivermos o seguinte código de módulo, qual instrução importará o Ninja aula?

```
//Ninja.js
exportar classe padrão Ninja {
    skulk () {return "skulking"; }
}
```

uma importar Ninja de “Ninja.js”

b importar * como Ninja de “Ninja.js”

c importar * de “Ninja.js”

Parte 4

Reconhecimento de navegador

Nomo que exploramos os fundamentos da linguagem JavaScript, passaremos para os navegadores, o ambiente no qual a maioria dos aplicativos JavaScript é executada.

No capítulo 12, examinaremos mais de perto o DOM, explorando técnicas eficientes para modificá-lo e obter aplicativos da Web rápidos e altamente dinâmicos.

No capítulo 13, você aprenderá sobre eventos, com foco especial no loop de eventos e sua influência no desempenho percebido do aplicativo da web.

Finalmente, o livro conclui com um tópico não tão agradável, mas necessário: desenvolvimento de crossbrowser. Embora a situação tenha melhorado muito nos últimos anos, ainda não podemos presumir que nosso código funcionará da mesma maneira em todos os navegadores disponíveis. Portanto, o capítulo 14 apresenta estratégias para o desenvolvimento de aplicativos da web entre navegadores.

12 Trabalhando o DOM

Este capítulo cobre

- Inserindo HTML no DOM
- Noções básicas sobre atributos e propriedades DOM
- Descobrindo estilos computados
- Lidando com a alteração do layout

Até agora, você aprendeu principalmente sobre a linguagem JavaScript e, embora haja muitas nuances no JavaScript puro, o desenvolvimento de aplicativos da web definitivamente não fica mais fácil quando colocamos o Document Object Model (DOM) do navegador na mistura . Um dos principais meios para alcançar aplicativos da web altamente dinâmicos que respondem às ações do usuário é modificar o DOM. Mas se abríssemos uma biblioteca JavaScript, você notaria o comprimento e a complexidade do código por trás de operações simples do DOM. Mesmo operações presumivelmente simples, como cloneNode e removeChild têm implementações relativamente complexas.

Isso levanta duas questões:

- Por que esse código é tão complexo?
- Por que você precisa entender como funciona se a biblioteca vai cuidar disso para você?

O motivo mais convincente é *atuação*. Entender como a modificação DOM funciona em bibliotecas pode permitir que você escreva um código melhor e mais rápido que usa a biblioteca ou, alternativamente, permite que você use essas técnicas em seu próprio código.

Portanto, começaremos este capítulo vendo como criar novas partes de nossas páginas, sob demanda, injetando HTML arbitrário. Continuaremos examinando todos os enigmas que os navegadores nos lançam com respeito às propriedades e atributos dos elementos, e descobriremos por que os resultados nem sempre são exatamente os que esperamos.

O mesmo vale para Cascading Style Sheets (CSS) e o estilo dos elementos. Muitas das dificuldades que encontraremos ao construir um aplicativo da web dinâmico resultam das complicações de configuração e obtenção de estilo de elemento. Este livro não pode cobrir tudo o que se sabe sobre como lidar com o estilo dos elementos (isso é o suficiente para preencher outro livro inteiro), mas os fundamentos básicos são discutidos.

Concluiremos o capítulo dando uma olhada em algumas das dificuldades de desempenho que podem surgir se você não prestar atenção à maneira como modifica e lê as informações do DOM. Vamos começar vendo como injetar HTML arbitrário em nossas páginas.

Você sabe?

Por que você precisa preparar elementos de fechamento automático em uma página antes de injetar HTML nela?

Quais são os benefícios de trabalhar com fragmentos DOM ao inserir HTML?

Como você determina as dimensões de um elemento escondido ment em uma página?

12.1 Injetando HTML no DOM

Nesta seção, veremos uma maneira eficiente de inserir HTML em um documento em qualquer local, considerando esse HTML como uma string. Apresentamos essa técnica específica porque ela é frequentemente usada para criar páginas da web altamente dinâmicas nas quais a interface do usuário é modificada como uma resposta às ações do usuário ou dados recebidos do servidor. Isso é particularmente útil para os seguintes cenários:

- Injetar HTML arbitrário em uma página e manipular e inserir modelos do lado do cliente
- Recuperando e injetando HTML enviado de um servidor

Pode ser tecnicamente desafiador implementar essa funcionalidade corretamente (especialmente quando comparada à construção de uma API de construção DOM de estilo orientado a objetos, que certamente é mais fácil de implementar, mas requer uma camada extra de abstração do que injetar o HTML). Considere este exemplo de criação de elementos HTML a partir de uma string HTML que podemos usar com jQuery:

```
$ (document.body) .append ("<div> <h1> Saudações </h1> <p> Yoshi aqui </p> </div>")
```

E compare isso com uma abordagem que usa apenas a API DOM:

```
const h1 = document.createElement ("h1"); h1.textContent = "Saudações";


const p = document.createElement ("p");
p.textContent = "Yoshi aqui";

const div = document.createElement ("div");

div.appendChild (h1);
div.appendChild (p);

document.body.appendChild (div);
```

Qual você prefere usar?

Por esses motivos, implementaremos nossa própria maneira de fazer a manipulação limpa do DOM do zero. A implementação requer as seguintes etapas:

1. Converta uma string HTML arbitrária, mas válida, em uma estrutura DOM.
2. Injete essa estrutura DOM em qualquer local no DOM da maneira mais eficiente possível.

Essas etapas fornecem aos autores de páginas uma maneira inteligente de injetar HTML em um documento. Vamos começar.

12.1.1 Convertendo HTML em DOM

Converter uma string HTML em uma estrutura DOM não envolve muita mágica. Na verdade, ele usa uma ferramenta com a qual você provavelmente já está familiarizado: o innerHTML propriedade dos elementos DOM.

Usá-lo é um processo de várias etapas:

1. Certifique-se de que a string HTML contém um código HTML válido.
2. Envolva a string em qualquer marcação envolvente exigida pelas regras do navegador. Insira a
3. string HTML, usando innerHTML, em um elemento DOM fictício. Extraia os nós DOM de volta.
4.

As etapas não são excessivamente complexas, mas a inserção real tem alguns truques que precisaremos levar em consideração. Vamos dar uma olhada em cada etapa em detalhes.

PREPROCESSANDO A CADEIA DE FONTE HTML

Para começar, precisaremos limpar o código-fonte HTML para atender às nossas necessidades. Por exemplo, vamos dar uma olhada em um esqueleto HTML que nos permite escolher um ninja (por meio do opção
) e que mostra os detalhes do ninja escolhido em uma tabela, detalhes que devem ser adicionados posteriormente:

```
<option> Yoshi </option>
<option> Kuma </option>
<mesa />
```

Esta string HTML tem dois problemas. Primeiro, os elementos de opção não devem se sustentar por si próprios. Se você seguir a semântica HTML adequada, eles devem estar contidos em um
selecionar elemento. Em segundo lugar, embora as linguagens de marcação geralmente nos permitam fechar automaticamente os elementos sem filhos, como < tabela />, em HTML, o fechamento automático funciona apenas para um pequeno subconjunto de elementos (tabela não sendo um deles). Tentar usar essa sintaxe em outros casos pode causar problemas em alguns navegadores.

Vamos começar resolvendo o problema dos elementos de fechamento automático. Para oferecer suporte a esse recurso, podemos fazer uma preparação rápida na string HTML para converter elementos como
<mesa /> para < tabela> </table> (que será tratado uniformemente em todos os navegadores), conforme mostrado na lista a seguir.

Listagem 12.1 Certificando-se de que os elementos de fechamento automático são interpretados corretamente

```
tags const =
↪ / ^ ( area | base | br | col | embed | hr | img | input | keygen | link | menuitem | meta | param |
↪         fonte | faixa | wbr) $ / i;
function convert (html) {
    return html.replace (
↪                       / (<(\ w +) [^>] *?) \ /> / g, (all, front, tag) => {return tags.test (tag)? todo :

                                                front + "> </" + tag + ">";
        });
    }
    assert (convert ("<a/>") === "<a> </a>", "Verifique a conversão âncora."); assert (convert ("<hr />") === "<hr />", "Verifique a
    conversão de hr.");
```

Usa uma expressão regular para corresponder ao nome da tag de quaisquer elementos com os quais não precisamos nos preocupar

Uma função que usa expressões regulares para converter auto-fechamento tags para a forma "normal"

Quando aplicamos o converter a esta string HTML de exemplo, terminamos com a seguinte string HTML:

```
<option> Yoshi </option>
<option> Kuma </option>
<table> </table>
```

<tabela /> expandida

Feito isso, ainda temos que resolver o problema que nosso opção elementos não estão contidos em um selecionar elemento. Vamos ver como determinar se um elemento precisa ser encapsulado.

WRAPPING HTML

De acordo com a semântica do HTML, alguns elementos HTML devem estar dentro de certos elementos de contêiner antes de serem injetados. Por exemplo, um < opção> elemento deve estar contido em um < selecione>.

Podemos resolver esse problema de duas maneiras, sendo que ambas requerem a construção de um mapeamento entre os elementos problemáticos e seus contêineres:

- A string pode ser injetada diretamente em um pai específico usando innerHTML, onde o pai foi construído anteriormente usando o integrado documento createElement. Embora isso possa funcionar em alguns casos e em alguns navegadores, não é universalmente garantido.

- A string pode ser envolvida com a marcação necessária apropriada e, em seguida, injetada diretamente em qualquer elemento de contêiner (como um < div>). Isso é mais infalível, mas também dá mais trabalho.

A segunda técnica é preferida; envolve pouco código específico do navegador, em contraste com a primeira abordagem, que requer uma boa quantidade de código principalmente específico do navegador.

O conjunto de elementos problemáticos que precisam ser incluídos em elementos de contêiner específicos é felizmente um conjunto de sete bastante gerenciável. Na tabela 12.1, as reticências (...) indicam os locais onde os elementos devem ser injetados.

Tabela 12.1 Elementos que precisam estar contidos em outros elementos

Nome do elemento	Elemento ancestral
<option>, <optgroup>	<selecionar vários> ... </select>
<legenda>	<fieldset> ... </fieldset>
<thead>, <tbody>, <tfoot>, <table> ... </table>	<colgroup>, <caption>
<tr>	<table><thead>...</thead> </table> <table><tbody>...</tbody> </table> <table><tfoot>...</tfoot> </table>
<td>, <th>	<table><tbody><tr>...</tr></tbody> </table>
<col>	<table> <tbody> </tbody> <colgroup> ... </colgroup> </table>

Quase todos eles são diretos, exceto pelos seguintes pontos, que requerem um pouco de explicação:

- A < selecione> elemento com o múltiplo atributo é usado (ao contrário de uma seleção não múltipla) porque ele não verificará automaticamente nenhuma das opções que são colocadas dentro dele (enquanto uma única seleção verificará automaticamente a primeira opção). O < col> correção inclui um < tbody>, sem o qual o < colgroup>
- não será gerado corretamente.

Com os elementos devidamente mapeados para seus requisitos de agrupamento, vamos começar a gerar.

Com as informações da tabela 12.1, podemos gerar o HTML que precisamos inserir em um elemento DOM, conforme mostrado na lista a seguir.

Listagem 12.2 Gerando uma lista de nós DOM de alguma marcação

```
function getNodes (htmlString, doc) {
    const map = {
        "<td": [3, "<table> <tbody> <tr>", "</tr> </tbody> </table>"], "<th": [3, "<table> <tbody > <tr> "," </tr>
        </tbody> </table> "]," <tr ": [2," <table> <thead> "," </thead> </table> "] ,

        "<option": [1, "<select multiple>", "</select>"],
        "<optgroup": [1, "<select multiple>", "</select>"],
        "<legend": [1, "<fieldset>", "</fieldset>"],
        "<thead": [1, "<table>", "</table>"],
        "<tbody": [1, "<table>", "</table>"],
        "<tfoot": [1, "<table>", "</table>"],
        "<colgroup": [1, "<table>", "</table>"],
        "<caption": [1, "<table>", "</table>"],
        "<col": [2, "<table> <tbody> </tbody> <colgroup>", "</colgroup> </table>"],
    };
    const tagName = htmlString.match (/ <\ w + /);
    let mapEntry = tagName? map [tagName [0]]: null; if (! mapEntry) {mapEntry = [0, "", ""];}

    let div = (doc || document) .createElement ("div");
    div.innerHTML = mapEntry [1] + htmlString + mapEntry [2]; while (mapEntry [0] -) {div = div.lastChild;}
    return div.childNodes;
}
assert (getNodes ("<td> test </td> <td> test2 </td>") .length === 2,
        "Obtenha dois nós de volta do método.");
assert (getNodes ("<td> test </td>") [0] .nodeName === "TD",
        "Verifique se estamos obtendo o nó correto.");
```

Mapa de tipos de elementos que precisam de contêineres pai especiais. Cada entrada tem a profundidade do novo nó, abrindo HTML para os pais e fechando o HTML para os pais.

Partidas a abertura suporte e marcar nome

Se estiver no mapa, captura a entrada; caso contrário, constrói uma entrada falsa com marcação "pai" vazia e profundidade zero.

Cria um elemento no qual criar os novos nós. Observe que usamos um documento aprovado, se existir, ou o padrão para o documento atual, caso não exista.

Retorna o elemento recém-criado

Envolve a entrada marcação com o pais da entrada do mapa e o injeta como o HTML interno do recém-criado

Desce a árvore recém-criada até a profundidade indicada pela entrada do mapa. Este deve ser o pai do nó desejado criado a partir da marcação.

Criamos um mapa de todos os tipos de elementos que precisam ser colocados em contêineres pai especiais, um mapa que contém a profundidade do nó, bem como o HTML que o contém. Em seguida, usamos uma expressão regular para combinar o colchete de abertura e o nome da tag do elemento que queremos inserir:

```
const tagName = htmlString.match (/ <\ w + /);
```

Em seguida, selecionamos uma entrada do mapa e, caso não haja uma, criamos uma entrada fictícia com uma marcação de elemento pai vazia:

```
let mapEntry = tagName? map [tagName [0]]: null; if (! mapEntry) {mapEntry = [0, "", ""]; }
```

Seguimos isso criando um novo div elemento, envolvendo-o com o HTML mapeado e inserindo o HTML recém-criado no criado anteriormente div elemento:

```
let div = (doc || document) .createElement ("div");
div.innerHTML = mapEntry [1] + htmlString + mapEntry [2]
```

Finalmente, encontramos o pai do nó desejado criado a partir de nossa string HTML e retornamos o nó recém-criado:

```
while (mapEntry [0] -) {div = div.lastChild;} return div.childNodes;
```

Depois de tudo isso, temos um conjunto de nós DOM que podemos começar a inserir no documento.

Se voltarmos ao nosso exemplo motivador e aplicar o getNodes função, vamos acabar com algo nas seguintes linhas:

```
<selecionar vários>
    <option> Yoshi </option>
    <option> Kuma </option>
</select>
<table> </table>
```

Elementos de opção têm foi embalado dentro de um elemento de seleção.

12.1.2 Inserindo elementos no documento

Depois de termos os nós DOM, é hora de inseri-los no documento. Algumas etapas são necessárias, e as trabalharemos nesta seção.

Como temos uma matriz de elementos que precisamos inserir - potencialmente em qualquer número de locais dentro do documento - queremos tentar manter o número de operações realizadas ao mínimo. Podemos fazer isso usando *Fragmentos DOM.* Os fragmentos DOM fazem parte da especificação W3C DOM e são suportados em todos os navegadores. Esse recurso útil nos dá um contêiner para armazenar uma coleção de nós DOM.

Isso em si é bastante útil, mas também tem a vantagem de que o fragmento pode ser injetado e clonado em uma única operação, em vez de ter que injetar e clonar cada nó individual repetidamente. Isso tem o potencial de reduzir drasticamente o número de operações necessárias para uma página.

Antes de usarmos esse mecanismo em nosso código, vamos revisitar o getNodes () código da listagem 12.2 e ajuste-o um pouco para usar fragmentos DOM. As mudanças são pequenas e consistem em adicionar um fragmento parâmetro para a lista de parâmetros da função, como segue.

Listagem 12.3 Expandindo o **getNodes** função com fragmentos

```
function getNodes (htmlString, doc, fragmento){
        const map = {
            "<td": [3, "<table> <tbody> <tr>", "</tr> </tbody> </table>"], "<th": [3, "<table> <tbody > <tr> "," </tr>
            </tbody> </table> "]," <tr ": [2," <table> <thead> "," </thead> </table> "] ,

            "<option": [1, "<select multiple>", "</select>"],
            "<optgroup": [1, "<select multiple>", "</select>"],
            "<legend": [1, "<fieldset>", "</fieldset>"],
            "<thead": [1, "<table>", "</table>"],
            "<tbody": [1, "<table>", "</table>"],
            "<tfoot": [1, "<table>", "</table>"],
            "<colgroup": [1, "<table>", "</table>"],
            "<caption": [1, "<table>", "</table>"],
```

Adiciona um novo fragmento parâmetro para a função

```
        "<col": [2, "<table> <tbody> </tbody> <colgroup>", "</colgroup> </table>"],
    };
    const tagName = htmlString.match (/ <\ w + />;
    let mapEntry = tagName? map [tagName [0]]: null; if (! mapEntry) {mapEntry = [0, "", ""];}

    let div = (doc || document) .createElement ("div");
    div.innerHTML = mapEntry [1] + htmlString + mapEntry [2]; while (mapEntry [0] -) {div = div.lastChild;} if
    (fragmento) {

        while (div.firstChild) {
            fragment.appendChild (div.firstChild);
        }
    }

    return div.childNodes;
}
```

Se o fragmento existe, injeta o nós nele.

Neste exemplo, fazemos algumas alterações. Primeiro, modificamos a assinatura da função adicionando outro parâmetro, fragmento:

```
function getNodes (htmlString, doc, fragmento) {...}
```

Este parâmetro, se for passado, deve ser um fragmento DOM no qual queremos que os nós sejam injetados para uso posterior.

Para fazer isso, adicionamos o seguinte fragmento antes do Retorna declaração da função para adicionar os nós ao fragmento passado:

```
if (fragmento) {
    while (div.firstChild) {
        fragment.appendChild (div.firstChild);
    }
}
```

Agora, vamos ver em uso. Na lista a seguir, que assume que o getNodes função está no escopo, um fragmento é criado e passado para essa função (que, você deve se lembrar, converte a string HTML recebida em elementos DOM). Este DOM agora está anexado ao fragmento.

Listagem 12.4 Inserindo um fragmento DOM em vários locais no DOM

```
<div id = "test"> <b> Olá </b>, sou um ninja! </div> <div id = "test2"> </div>

<script>
    document.addEventListener ("DOMContentLoaded", () => {
        inserção de função (elems, args, callback) {
            if (elems.length) {
                const doc = elems [0] .ownerDocument || elems [0],
                        fragment = doc.createDocumentFragment (),
                        scripts = getNodes (args, doc, fragmento), primeiro = fragment.firstChild;
```

Configura alguns nós de teste

Cria um documento fragmento no qual vamos inserir nós

Cria HTML nós do String HTML

```
                if (primeiro) {
                    para (deixe i = 0; elems [i]; i ++) {
                        callback.call (root (elems [i], primeiro),
                                i> 0? fragment.cloneNode (true): fragmento);
                    }
                }
            }
        }
        const divs = document.querySelectorAll ("div"); inserir (divs, "<b> Nome: </b>", função
        (fragmento) {
            this.appendChild (fragmento);
        });

        insert (divs, "<span> primeiro </span> <span> último </span>",
                função (fragmento) {
                    this.parentNode.insertBefore (fragmento, isso);
                });
    });
</script>
```

Se precisarmos inserir os nós em mais de um elemento, temos que clonar o fragmento a cada vez.

Há outro ponto importante aqui: se estivermos inserindo esse elemento em mais de um local no documento, precisaremos clonar esse fragmento várias vezes. Se não estivéssemos usando um fragmento, teríamos que clonar cada nó individual todas as vezes, em vez de todo o fragmento de uma vez.

Com isso, desenvolvemos uma maneira de gerar e inserir elementos DOM arbitrários de maneira intuitiva. Vamos continuar esta exploração do DOM vendo a diferença entre os atributos e propriedades do DOM.

12.2 Usando atributos e propriedades DOM

Ao acessar os valores dos atributos do elemento, temos duas opções: usando os métodos tradicionais de DOM de getAttribute e setAttribute, ou usando propriedades dos objetos DOM que correspondem aos atributos.

Por exemplo, para obter o eu ia de um elemento cuja referência é armazenada na variável e, podemos usar um dos seguintes:

```
e.getAttribute ('id')
e.id
```

Qualquer um nos dará o valor do eu ia.

Vamos examinar o código a seguir para entender melhor como os valores dos atributos e suas propriedades correspondentes se comportam.

Listagem 12.5 Acessando valores de atributo por meio de métodos e propriedades DOM

```
<div> </div>
<script>
    document.addEventListener ("DOMContentLoaded", () => {
        const div = document.querySelector ("div");
```

Obtém um elemento referência

```
        div.setAttribute ("id", "ninja-1");
        assert (div.getAttribute ('id') === "ninja-1",
                "Atributo alterado com sucesso");
        div.id = "ninja-2";
        assert (div.id === "ninja-2",
                "Propriedade alterada com sucesso");

        assert (div.getAttribute ('id') === "ninja-2",
                "Atributo alterado com sucesso via propriedade");
        div.setAttribute ("id", "ninja-3");
        assert (div.id === "ninja-3",
                "Propriedade alterada com sucesso via atributo"); assert (div.getAttribute ('id') ===
        "ninja-3",
                "Atributo alterado com sucesso");
    });
</script>
```

Altera o valor do atributo id com o método setAttribute e testa se o valor mudou

Altera o valor da propriedade e testa se o valor mudou

Usando setAttribute também muda O valor que obtido através de propriedade.

Alterando a propriedade também muda o valor obtido com getAttribute.

Este exemplo mostra um comportamento interessante com relação aos atributos e propriedades do elemento. Ele começa definindo um < div> elemento que usaremos como assunto de teste. Dentro do documento DOMContentLoaded handler (para garantir que o DOM seja totalmente construído), obtemos uma referência ao único < div> elemento, const div = document

. querySelector ("div"), e, em seguida, execute alguns testes.

No primeiro teste, definimos o eu ia Atribua ao valor ninja-1 através do setAttribute () método. Então afirmamos que getAttribute () retorna o mesmo valor para esse atributo. Não deve ser surpresa que este teste funcione bem quando carregamos a página:

```
div.setAttribute ("id", "ninja-1");
assert (div.getAttribute ('id') === "ninja-1",
        "Atributo alterado com sucesso");
```

Da mesma forma, no próximo teste, definimos o eu ia propriedade para o valor ninja-2 e então verifique se o valor da propriedade realmente mudou. Sem problemas.

```
div.id = "ninja-2";
assert (div.id === "ninja-2",
        "Propriedade alterada com sucesso");
```

O próximo teste é quando as coisas ficam interessantes. Nós novamente definimos o eu ia propriedade para um novo valor, neste caso ninja-3, e verifique novamente se o valor da propriedade mudou. Mas então também afirmamos que não só deve mudar o valor da propriedade, mas também o valor do eu ia *atributo*. Ambas as afirmações passam. Com isso aprendemos que o eu ia propriedade e o eu ia atributo estão de alguma forma ligados entre si. Alterando a eu ia o valor da propriedade também altera o eu ia Valor do atributo:

```
div.id = "ninja-3";
assert (div.id === "ninja-3",
        "Propriedade alterada com sucesso");
assert (div.getAttribute ('id') === "ninja-3",
        "Atributo alterado com sucesso via propriedade");
```

O próximo teste prova que também funciona ao contrário: definir um valor de atributo também altera o valor da propriedade correspondente.

```
div.setAttribute ("id", "ninja-4");
assert (div.id === "ninja-4",
                "Propriedade alterada com sucesso via atributo");
assert (div.getAttribute ('id') === "ninja-4", "Atributo alterado");
```

Mas não se deixe enganar pensando que a propriedade e o atributo compartilham o mesmo valor - não estão. Veremos mais adiante neste capítulo que o atributo e a propriedade correspondente, embora vinculados, nem sempre são idênticos.

É importante observar que nem todos os atributos são representados por propriedades de elementos. Embora seja geralmente verdadeiro para atributos que são nativamente especificados pelo HTML DOM, *atributos personalizados* que podemos colocar nos elementos em nossas páginas não são automaticamente representados pelas propriedades do elemento. Para acessar o valor de um atributo personalizado, precisamos usar os métodos DOM getAttribute () e setAttribute ().

Se você não tiver certeza se uma propriedade para um atributo existe, você sempre pode testá-la e voltar aos métodos DOM se ela não existir. Aqui está um exemplo:

```
const value = element.someValue? element.someValue
                                                    : element.getAttribute ('someValue');
```

GORJETA Em HTML5, use o prefixo dados- para todos os atributos personalizados para mantê-los válidos de acordo com a especificação HTML5. É uma boa convenção que separa claramente os atributos personalizados dos atributos nativos.

12.3 Dores de cabeça de atributo de estilo

Tal como acontece com os atributos gerais, obter e definir atributos de estilo pode ser uma dor de cabeça. Tal como acontece com os atributos e propriedades na seção anterior, novamente temos duas abordagens para lidar com estilo valores: o valor do atributo e a propriedade do elemento criada a partir dele.

O mais comumente usado é o estilo propriedade do elemento, que não é uma string, mas um objeto que contém propriedades correspondentes aos valores de estilo especificados na marcação do elemento. Além disso, você verá que há um método para acessar as informações de estilo computado de um elemento, onde *estilo computado* significa o estilo que será aplicado ao elemento após avaliar todas as informações de estilo herdadas e aplicadas.

Esta seção descreve o que você precisa saber ao trabalhar com estilos em navegadores. Vamos começar examinando onde as informações de estilo são registradas.

12.3.1 Onde estão meus estilos?

As informações de estilo localizadas no estilo propriedade de um elemento DOM é inicialmente definida a partir do valor especificado para o estilo atributo na marcação do elemento. Por exemplo,
estilo = "cor: vermelho;" resulta na inserção dessas informações de estilo no objeto de estilo. Durante a execução da página, o script pode definir ou modificar valores no objeto de estilo e essas alterações afetarão ativamente a exibição do elemento.

Muitos autores de script ficam desapontados ao descobrir que nenhum valor da página < estilo> elementos ou folhas de estilo externas estão disponíveis no estilo objeto. Mas não ficaremos desapontados por muito tempo - em breve você verá uma maneira de obter essas informações.

Por enquanto, vamos ver como o estilo propriedade obtém seus valores. Examine o seguinte código.

Listagem 12.6 Examinando o **estilo** propriedade

```
<style>
    div {tamanho da fonte: 1.8em; borda: 0 ouro maciço; } </style>

<div style = "color: # 000;" title = "Poder Ninja!">
    忍者 パ ワ ー
</div>
<script>
    document.addEventListener ("DOMContentLoaded", () => {
        const div = document.querySelector ("div"); assert (div.style.color === 'rgb (0, 0, 0)' ||
                    div.style.color === '# 000',
                    'a cor foi registrada');
        assert (div.style.fontSize === '1.8em',
                    'fontSize foi gravado');
        assert (div.style.borderWidth === '0',
                    'borderWidth foi gravado');
        div.style.borderWidth = "4px";
        assert (div.style.borderWidth === '4px',
                    'borderWidth foi substituído');
    });
</script>
```

Declara uma folha de estilo in-page que se aplica tamanho da fonte e informações de borda

Este elemento de teste deve receber vários estilos de vários lugares, incluindo seu próprio atributo de estilo e a folha de estilo.

Testa que o estilo de cor embutido foi gravado

Testa se o estilo de tamanho de fonte herdado foi registrado

Testa se o estilo de largura de borda herdado foi registrado

Substitui a fronteira estilo de largura

Testa o estilo da largura da borda mudança foi registrada

Neste exemplo, configuramos um < estilo> elemento para estabelecer uma folha de estilo interna cujos valores serão aplicados aos elementos da página. A folha de estilo especifica que todos os elementos aparecerão em um tamanho de fonte 1,8 vezes maior do que o padrão, com uma borda de ouro sólido 0 largura. Quaisquer elementos aos quais isso seja aplicado possuirão uma borda, mas não serão visíveis porque têm uma largura de 0

```
<style>
    div {tamanho da fonte: 1.8em; borda: 0 ouro maciço; } </style>
```

Em seguida, criamos um < div> elemento com um atributo de estilo embutido que colore o texto do elemento em preto:

```
<div style = "color: # 000;" title = "Poder Ninja!">
    忍者 パ ワ ー
</div>
```

Em seguida, começamos o teste. Depois de obter uma referência ao < div> elemento, testamos que o estilo atributo recebe um cor propriedade que representa a cor atribuída ao elemento. Observe que embora o cor é especificado como # 000 no estilo inline, é normalizado para notação RGB quando definido no estilo na maioria dos navegadores (por isso verificamos os dois formatos).

```
assert (div.style.color === 'rgb (0, 0, 0)' ||
                div.style.color === '# 000',
                'a cor foi registrada');
```

Olhando para frente, na figura 12.1, vemos que esse teste foi aprovado.

Em seguida, testamos ingenuamente se o estilo do tamanho da fonte e a largura da borda especificada na folha de estilo embutida foram registrados no objeto de estilo. Porém, embora possamos ver na figura 12.1 que o estilo de tamanho da fonte foi aplicado ao elemento, o teste falha. Isso ocorre porque o objeto de estilo não reflete nenhuma informação de estilo herdada das folhas de estilo CSS:

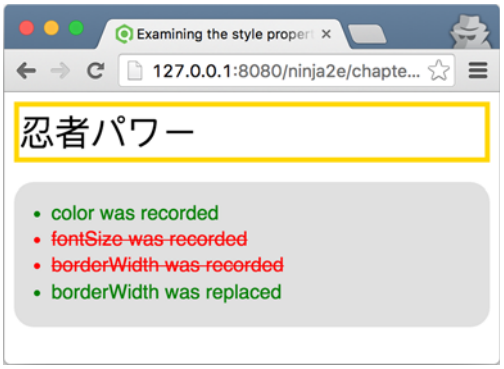


Figura 12.1 Ao executar este teste, podemos ver que os estilos embutidos e atribuídos são registrados, mas os estilos herdados não.

```
assert (div.style.fontSize === '1.8em',
                'fontSize foi gravado');
assert (div.style.borderWidth === '0',
                'borderWidth foi gravado');
```

Continuando, usamos uma atribuição para alterar o valor do borderWidth propriedade no objeto de estilo para 4 pixels de largura e teste se a alteração é aplicada. Podemos ver na figura 12.1 que o teste passa e que a borda anteriormente invisível é aplicada ao elemento. Esta atribuição causa um borderWidth propriedade para aparecer no estilo propriedade do elemento, conforme comprovado pelo teste.

```
div.style.borderWidth = "4px";
assert (div.style.borderWidth === '4px',
                'borderWidth foi substituído');
```

Deve-se notar que quaisquer valores em um elemento estilo A propriedade tem precedência sobre qualquer coisa herdada por uma folha de estilo (mesmo se a regra da folha de estilo usar o! importante anotação).

Uma coisa que você deve ter notado na listagem 12.6 é que o CSS especifica a propriedade do tamanho da fonte como tamanho da fonte, mas no script você o referencia como tamanho da fonte. Por que é que?

12.3.2 Nomenclatura de propriedade de estilo

Os atributos CSS causam relativamente poucas dificuldades entre navegadores quando se trata de acessar os valores fornecidos pelo navegador. Mas existem diferenças entre como os estilos de nomes CSS e como os acessamos no script, e alguns nomes de estilo diferem entre os navegadores.

Os atributos CSS que abrangem mais de uma palavra separam as palavras com um hífen; exemplos são font-weight, font-size, e cor de fundo. Você deve se lembrar que nomes de propriedades em JavaScript *posso* contêm um hífen, mas a inclusão de um hífen impede que a propriedade seja acessada por meio do operador ponto.

Considere este exemplo:

```
const fontSize = element.style ['font-size'];
```

O anterior é perfeitamente válido. Mas o seguinte não é:

```
const fontSize = element.style.font-size;
```

O analisador JavaScript veria o hífen como um operador de subtração e ninguém ficaria feliz com o resultado. Em vez de forçar os desenvolvedores de página a sempre usar o formulário geral para acesso à propriedade, *nomes de estilo CSS com várias palavras são convertidos em caixa de camelo quando usados como um nome de propriedade*. Como um resultado, tamanho da fonte torna-se tamanho da fonte, e fundo-cor torna-se cor de fundo.

Podemos nos lembrar de fazer isso ou escrever uma API simples para definir ou obter estilos que manipulam automaticamente a caixa do camelo, conforme mostrado na lista a seguir.

Listagem 12.7 Um método simples para acessar estilos

Define a função de estilo que atribuirá um valor a uma propriedade de estilo caso um valor seja fornecido e, caso não seja, simplesmente retornará o valor da propriedade de estilo. Podemos usar esta função para definir e obter o valor de uma propriedade de estilo.

```
<div style = "color: red; font-size: 10px; background-color: #eee;"> </div> <script>

    estilo da função (elemento, nome, valor) {
        nome = nome.substituir (/ - ([az]) / ig, (todos, letra) => {
            return letter.toUpperCase ();
        });
        if (typeof value! == 'undefined') {
            element.style [nome] = valor;
        }
        return element.style [name];
    }
    document.addEventListener ("DOMContentLoaded", () => {
        const div = document.querySelector ("div");
        assert (style (div, 'color') === "red", style (div, 'color')); assert (style (div, 'font-size') === "10px", style (div, 'font-size')); assert
        (style (div, 'background-color') ===

        "rgb (238, 238, 238)", estilo (div, 'cor de fundo'));
    });
</script>
```

Converte o nome para a caixa de camelo

Sempre retorna O valor que do estilo propriedade

O novo valor da propriedade de estilo é definido, se um valor for fornecido.

A função de estilo tem duas características importantes:

- Ele usa uma expressão regular para converter o nome parâmetro para notação camel-case. (Se a operação de conversão orientada por regex o deixou coçando a cabeça, você pode querer revisar o material no capítulo 10.)
- Ele pode ser usado como setter e getter, inspecionando sua própria lista de argumentos. Por exemplo, podemos obter o valor da propriedade font-size com
  estilo (div, 'font-size'), e podemos definir um novo valor com estilo (div, 'font-size', '5px').

Considere o seguinte código:

```
estilo da função (elemento, nome, valor) {
        . . .
    if (typeof value! == 'undefined') {
        element.style [nome] = valor;
    }

    return element.style [name];
}
```

Se um valor argumento é passado para a função, a função atua como um setter, definindo o valor passado como o valor do atributo. Se o valor argumento é omitido e apenas os dois primeiros argumentos são passados, ele atua como um getter, recuperando o valor do atributo especificado. Em qualquer um dos casos, o valor do atributo é retornado, o que facilita o uso da função em qualquer um de seus modos em uma cadeia de chamada de função.

O estilo propriedade de um elemento não inclui nenhuma informação de estilo que um elemento herda de folhas de estilo no escopo do elemento. Muitas vezes seria útil saber todo o estilo computado que foi aplicado a um elemento, então vamos ver se há uma maneira de obtê-lo.

12.3.3 Buscando estilos calculados

A qualquer momento, o *estilo computado* de um elemento é uma combinação de todos os estilos embutidos fornecidos pelo navegador, todos os estilos aplicados a ele por meio de folhas de estilo, o elemento
estilo atributo, e quaisquer manipulações do estilo propriedade por script. A Figura 12.2 mostra como as ferramentas de desenvolvedor de navegador diferenciam os estilos.

O método padrão, implementado por todos os navegadores modernos, é o getComputedStyle método. Este método aceita um elemento cujos estilos devem ser calculados e retorna uma interface por meio da qual as consultas de propriedade podem ser feitas. A interface retornada fornece um método chamado getPropertyValue para recuperar o estilo calculado de uma propriedade de estilo específica.

Ao contrário das propriedades de um elemento estilo objeto, o getPropertyValue
método aceita nomes de propriedade CSS (como tamanho da fonte e cor de fundo)
em vez das versões em caixa de camelo desses nomes.

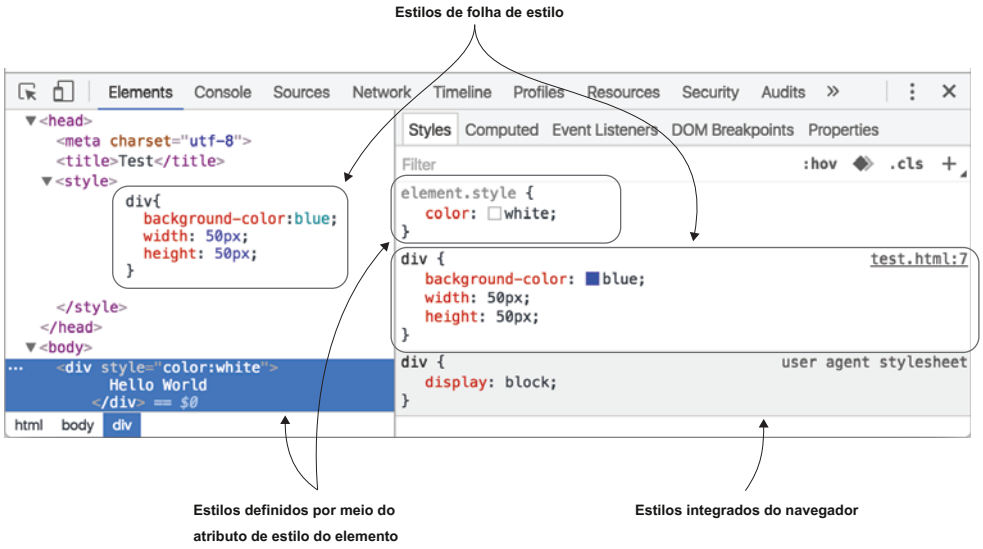


Figura 12.2 O estilo final associado a um elemento pode vir de muitas coisas: os estilos internos do navegador (folha de estilo do agente do usuário), os estilos atribuídos por meio da propriedade de estilo e os estilos das regras CSS definidas no código CSS.

A lista a seguir mostra um exemplo simples.

Listagem 12.8 Buscando valores de estilo computados

```
<style>
    div {
        cor de fundo: #ffc; display: inline; tamanho da fonte: 1.8em; borda: 1px carmesim sólido; cor verde;

    }
</style>
<div style = "color: crimson;" id = "testSubject" title = "Poder Ninja!">
    忍者 パ ワ ー
</div>
<script>
    function fetchComputedStyle (elemento, propriedade) {
        const computedStyles = getComputedStyle (elemento); if (computedStyles) {

            property = property.replace (/ ([AZ]) / g, '- $ 1'). toLowerCase (); return computedStyles.getPropertyValue
            (propriedade);
        }
    }
    document.addEventListener ("DOMContentLoaded", () => {
        const div = document.querySelector ("div"); relatório ("cor de fundo:" +

                    fetchComputedStyle (div, 'background-color'));
        relatório ("display:" +
                    fetchComputedStyle (div, 'display'));
```

Cria um assunto de teste com um atributo de estilo

Define um função que vai pegar o computado valor de um estilo propriedade

Usa o método getComputedStyle integrado para obter um objeto descritor

Testes que nós pode obter o valores de vários propriedades de estilo, usando diferente notações

Substitui caixa de camelo notação com travessões

```
        relatório ("tamanho da fonte:" +
                fetchComputedStyle (div, 'fontSize'));
        relatório ("cor:" +
                fetchComputedStyle (div, 'color'));
        relatório ("border-top-color:" +
                fetchComputedStyle (div, 'borderTopColor'));
        relatório ("border-top-width:" +
                fetchComputedStyle (div, 'border-top-width'));
    });
</script>
```

Para testar a função que iremos criar, configuramos um elemento que especifica informações de estilo em sua marcação e uma folha de estilo que fornece regras de estilo a serem aplicadas ao elemento. Esperamos que os estilos calculados sejam o resultado da aplicação dos estilos imediato e aplicado ao elemento.

Em seguida, definimos a nova função, que aceita um elemento e a propriedade de estilo para a qual queremos encontrar o valor calculado. E para ser especialmente amigável (afinal, somos ninjas - tornar as coisas mais fáceis para aqueles que usam nosso código faz parte do trabalho), permitiremos que nomes de propriedades com várias palavras sejam especificados em qualquer formato: tracejado ou camelcased. Em outras palavras, aceitaremos ambos cor de fundo e cor de fundo.
Veremos como fazer isso em breve.

A primeira coisa que queremos fazer é obter a interface de estilo computado, que armazenamos em uma variável, computedStyles, para referência posterior. Queremos fazer as coisas dessa maneira porque não sabemos o quão caro pode ser essa ligação e é melhor evitar repeti-la desnecessariamente.

```
const computedStyles = getComputedStyle (elemento); if (computedStyles) {

    property = property.replace (/ ([AZ]) / g, '- $ 1'). toLowerCase (); return computedStyles.getPropertyValue
    (propriedade);
}
```

Se isso for bem-sucedido (e não conseguimos pensar em nenhuma razão para não conseguir, mas frequentemente vale a pena ser cauteloso), chamamos o getPropertyValue () método da interface para obter o valor de estilo calculado. Mas primeiro ajustamos o nome da propriedade para acomodar a versão com caixa de camelo ou tracejado do nome da propriedade. O getPropertyValue método espera a versão tracejada, então usamos o Fragmento de substituir() método, com uma expressão regular simples, mas inteligente, para inserir um hífen antes de cada caractere maiúsculo e depois minúsculo tudo. (Aposto que foi mais fácil do que você pensava.)

Para testar a função, fazemos chamadas para a função, passando vários nomes de estilo em vários formatos, e exibimos os resultados, conforme mostrado na Figura 12.3.

Observe que os estilos são buscados independentemente de serem declarados explicitamente no elemento ou herdados da folha de estilo. Observe também que o cor propriedade, especificada na folha de estilo e diretamente no elemento, retorna o valor explícito. Estilos especificados por um elemento estilo atributo sempre tem precedência sobre estilos herdados, mesmo se marcado! importante.

Precisamos estar cientes de mais um tópico ao lidar com propriedades de estilo: *amálgama* propriedades. CSS nos permite usar uma notação de atalho para o amálgama de propriedades como o fronteira-

Figura 12.3 Os estilos calculados incluem todos os estilos especificados com o elemento, bem como aqueles herdados das folhas de estilo.

propriedades. Em vez de nos forçar a especificar cores, larguras e estilos de borda individualmente e para todas as quatro bordas, podemos usar uma regra como esta:

```
borda: 1px carmesim sólido;
```

Usamos essa regra exata na listagem 12.8. Isso economiza muita digitação, mas precisamos estar cientes de que, quando recuperamos as propriedades, precisamos buscar as propriedades individuais de nível inferior. Não podemos buscar fronteira, mas podemos buscar estilos como border-top-color e border-topwidth, exatamente como fizemos no exemplo.

Pode ser um pouco complicado, especialmente quando todos os quatro estilos têm os mesmos valores, mas essa é a mão que recebemos.

12.3.4 Conversão de valores de pixel

Um ponto importante a considerar ao definir valores de estilo é a atribuição de valores numéricos que representam pixels. Ao definir um valor numérico para uma propriedade de estilo, devemos especificar a unidade para que funcione de forma confiável em todos os navegadores. Por exemplo, digamos que queremos definir o altura valor de estilo de um elemento para 10 pixels. Uma das opções a seguir é uma maneira segura de fazer isso em vários navegadores:

```
element.style.height = "10px";
element.style.height = 10 + "px";
```

O seguinte não é seguro em navegadores:

```
element.style.height = 10;
```

Você pode pensar que seria fácil adicionar um pouco de lógica ao estilo() função da listagem 12.7 para unir um px ao final de um valor numérico que entra na função. Mas não tão rápido! Nem todos os valores numéricos representam pixels! Algumas propriedades de estilo assumem valores numéricos que não representam uma dimensão de pixel. A lista inclui o seguinte:

- z-index
- espessura da fonte
- opacidade
- ampliação
- altura da linha

Para estes (e quaisquer outros que você possa imaginar), vá em frente e estenda a função da listagem 12.6 para lidar automaticamente com valores não pixelados. Além disso, ao tentar ler um valor de pixel de um atributo de estilo, o parseFloat deve ser usado para garantir que você obtenha o valor pretendido em todas as circunstâncias.

Agora, vamos dar uma olhada em um conjunto de propriedades de estilo importantes que podem ser difíceis de manusear.

12.3.5 Medindo alturas e larguras

Propriedades de estilo, como altura e largura representam um problema especial, porque seus valores padrão para auto quando não especificado, para que o elemento se dimensione de acordo com seu conteúdo. Como resultado, não podemos usar o altura e largura propriedades de estilo para obter valores precisos, a menos que valores explícitos sejam fornecidos na string de atributo.

Felizmente, o offsetHeight e offsetWidth As propriedades fornecem exatamente isso: um meio bastante confiável de acessar a altura e a largura de um elemento. Mas esteja ciente de que os valores atribuídos a essas duas propriedades incluem o preenchimento do elemento. Essas informações geralmente são exatamente o que desejamos se estivermos tentando posicionar um elemento sobre o outro. Mas às vezes podemos querer obter informações sobre as dimensões do elemento com e sem bordas e preenchimento.

Algo a se observar, no entanto, é que em sites altamente interativos, os elementos provavelmente passarão parte de seu tempo em um estado não exibido (com o exibição estilo sendo definido para Nenhum), e quando um elemento não faz parte da exibição, ele não tem dimensões. Qualquer tentativa de buscar o offsetWidth ou offsetHeight propriedades de um elemento não exibido resultarão em um valor de 0

Para esses elementos ocultos, se quisermos obter as dimensões não ocultas, podemos empregar um truque para revelar momentaneamente o elemento, capturar os valores e ocultá-lo novamente. Claro, queremos fazer isso de uma forma que não deixemos nenhuma pista visível de que isso está acontecendo nos bastidores. Como podemos tornar um elemento oculto não oculto sem torná-lo visível?

Empregando nossas habilidades ninja, podemos fazer isso! Veja como:

1. Mudar o exibição propriedade para quadra.
2. Definir visibilidade para escondido.
3. Definir posição para absoluto.
4. Pegue os valores de dimensão.
5. Restaure as propriedades alteradas.

Alterando a exibição propriedade para quadra nos permite agarrar os valores de offsetHeight e offsetWidth, mas isso torna o elemento parte da tela e, portanto, visível. Para tornar o elemento invisível, vamos definir o visibilidade propriedade para escondido. Mas (sempre há outro *mas)* isso vai deixar um grande buraco onde o elemento é posicionado, então também definimos o posição propriedade para absoluto para tirar o elemento do fluxo normal de exibição.

Tudo isso parece mais complicado do que a implementação, que é mostrada na lista a seguir.

Listagem 12.9 Capturando as dimensões de elementos ocultos

```
<div>
    Lorem ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse congue facilisis dignissim. Fusce
    sodales, odio commodo accumsan commodo, lacus odio aliquet purus,

    <img src = "../ images / ninja-with-pole.png" id = "withPole" alt = "ninja pole" /> <img src = "../ images / ninja-with-shuriken.png"

            id = estilo "withShuriken" = "display: nenhum" alt = "ninja shuriken" /> vel rhoncus elit sem quis libero. Cum sociis
    natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Em hac habitasse platea dictumst. Donec adipiscing
    urna ut nibh vestibulum vitae mattis leo rutrum. Etiam a lectus ut nunc mattis laoreet em



    placerat nulla. Aenean tincidunt lorem eu dolor commodo ornare.

</div>
<script
    (função(){                                                          <-- Cria um escopo privado Define
        const PROPERTIES = {                                            <-- as propriedades de destino
            posição: "absoluta",
            visibilidade: "oculto",
            display: "bloquear"
        };
        window.getDimensions = element => {                             <-- Cria a nova função Lembra
            const anterior = {};                                        <-- configurações
            para (deixe digitar PROPERTIES) {
                anterior [chave] = element.style [chave];
                element.style [chave] = PROPRIEDADES [chave];           <-- Substitui as configurações
            }
            const result = {                                            <-- Coleta dimensões
                largura: element.offsetWidth,
                height: element.offsetHeight
            };
            para (deixe entrar PROPERTIES) {                            <-- Restaura as configurações
                element.style [chave] = anterior [chave];
            }
            resultado de retorno;
        };
    }) ();
    document.addEventListener ("DOMContentLoaded", () => {
        setTimeout (() => {
            const withPole = document.getElementById ('withPole'),
                  withShuriken = document.getElementById ('withShuriken'); assert (withPole.offsetWidth === 41,

                      "Largura da imagem do pólo obtida; real:" + withPole.offsetWidth + ", esperado:
                  41"); assert (withPole.offsetHeight === 48,                <-- Testa elemento visível

                      "Altura da imagem do pólo obtida: real:" + withPole.offsetHeight + ", esperado
                  48"); assert (withShuriken.offsetWidth === 36,
                                                                             <-- Testa elemento oculto
                      "Largura da imagem Shuriken obtida; real:" + withShuriken.offsetWidth + ", esperado:
                  36"); assert (withShuriken.offsetHeight === 48,

                      "Altura da imagem Shuriken obtida: real:" +
```

```
                withShuriken.offsetHeight + ", esperado 48"); dimensões const = getDimensions (com
        Shuriken); afirmar (dimensões. largura === 36,                          <-- Usa nova função

                "Largura da imagem Shuriken obtida; real:" + dimensões.largura + ", esperada:
            36"); afirmar (dimensões.altura === 48,                             <-- Reteste o elemento oculto

                "Altura da imagem Shuriken obtida: real:" + dimensões.altura + ", esperado 48");

    }, 3000);
  });
</script>
```

É uma longa lista, mas a maior parte é código de teste; a implementação da nova função de busca de dimensão abrange apenas uma dúzia ou mais de linhas de código.

Vamos dar uma olhada peça por peça. Primeiro, configuramos os elementos para testar: a < div> elemento que contém um monte de texto com duas imagens incorporadas, justificado à esquerda por estilos em uma folha de estilo externa. Esses elementos de imagem serão os objetos de nossos testes; um está visível e o outro está oculto.

Antes de executar qualquer script, os elementos aparecem conforme mostrado na figura 12.4. Se a segunda imagem não estivesse escondida, ela apareceria como um segundo ninja logo à direita do visível.

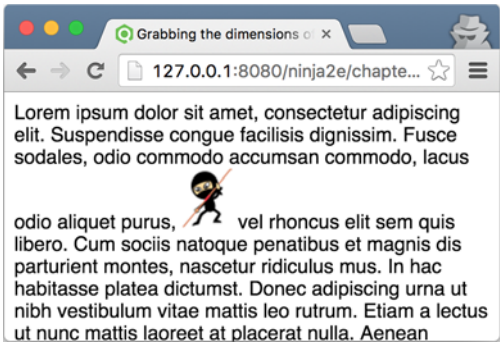


Figura 12.4 Usaremos duas imagens - uma visível, uma oculta - para testar a obtenção de dimensões de elementos ocultos.

Em seguida, começamos a definir nossa nova função. Vamos usar um hash para algumas informações importantes, mas não queremos poluir o namespace global com esse hash; queremos que ele esteja disponível para a função em seu escopo local, mas não além disso.

Conseguimos isso encerrando a definição de hash e a função declaração dentro de uma função imediata, que cria um escopo local. O hash não está acessível fora da função imediata, mas o getDimensions A função que também definimos na função imediata tem acesso ao hash por meio de seu encerramento. Legal, hein?

```
(função(){
    const PROPERTIES = {
        posição: "absoluta",
        visibilidade: "oculto",
        display: "bloquear"
    };
    window.getDimensions = element => {
        const anterior = {};
        para (deixe digitar PROPERTIES) {
            anterior [chave] = element.style [chave];
            element.style [chave] = PROPRIEDADES [chave];
        }
```

```
        const result = {
            largura: element.offsetWidth,
            height: element.offsetHeight
        };
        para (deixe digitar PROPERTIES) {
            element.style [chave] = anterior [chave];
        }
        resultado de retorno;
    };
}) ();
```

Nossa nova função de busca de dimensão é então declarada, aceitando o elemento a ser medido. Dentro dessa função, primeiro criamos um hash chamado anterior no qual registraremos os valores anteriores das propriedades de estilo que iremos piscar, para que possamos restaurá-los mais tarde. Fazendo um loop nas propriedades de substituição, registramos cada um de seus valores anteriores e substituímos esses valores pelos novos.

Feito isso, estamos prontos para medir o elemento, que agora faz parte do layout de exibição, invisível e posicionado absolutamente. As dimensões são registradas em um hash atribuído à variável local resultado.

Agora que roubamos o que buscamos, apagamos nossos rastros, restaurando os valores originais das propriedades de estilo que modificamos e retornamos os resultados como um hash contendo largura e altura propriedades.

Tudo bem, mas funciona? Vamos descobrir.

Em um gerenciador de carga, realizamos os testes em um retorno de chamada para um temporizador de 3 segundos. Porque você pergunta? O gerenciador de carga garante que não realizemos o teste até que saibamos que o DOM foi construído, e o cronômetro nos permite observar a tela enquanto o teste está em execução, para garantir que nenhuma falha de exibição ocorra enquanto mexemos nas propriedades do elemento oculto. Afinal, se a tela for perturbada de alguma forma quando executamos nossa função, é um fracasso.

No retorno de chamada do cronômetro, primeiro obtemos uma referência aos nossos assuntos de teste (as duas imagens) e afirmamos que podemos obter as dimensões da imagem visível usando as propriedades de deslocamento. Esse teste passa, o que podemos ver se dermos uma olhada na figura 12.5.

Em seguida, fazemos o mesmo teste no elemento oculto, assumindo incorretamente que as propriedades de deslocamento funcionarão com uma imagem oculta. Não é de surpreender que, como já reconhecemos que isso não vai funcionar, o teste falha.

Em seguida, chamamos nossa nova função na imagem oculta e, em seguida, testamos novamente com esses resultados. Sucesso! Nosso teste passa, conforme mostrado na figura 12.5.

Se observarmos a exibição da página enquanto o teste está sendo executado - lembre-se, atrasamos a execução do teste até 3 segundos depois que o DOM é carregado - podemos ver que a exibição não é perturbada de nenhuma forma por nossos bastidores ajustes das propriedades do elemento oculto.

GORJETA Verificando o offsetWidth e offsetHeight propriedades de estilo para zeros podem servir como um meio incrivelmente eficiente de determinar a visibilidade de um elemento.