Esse comportamento é uma consequência do fato de que os arrays JavaScript são objetos. Assim como nós obteríamos Indefinido se tentamos acessar uma propriedade de objeto inexistente, obtemos Indefinido ao acessar um índice de matriz inexistente.

Por outro lado, se tentarmos escrever para uma posição fora dos limites da matriz, como em

```
ninjas [4] = "Ishi";
```

a matriz se expandirá para acomodar a nova situação. Por exemplo, veja a figura 9.1: Essencialmente, criamos uma lacuna na matriz e o item no índice 3 é Indefinido.
Isso também muda o valor do comprimento propriedade, que agora relata um valor de 5,
mesmo que um item da matriz seja Indefinido.

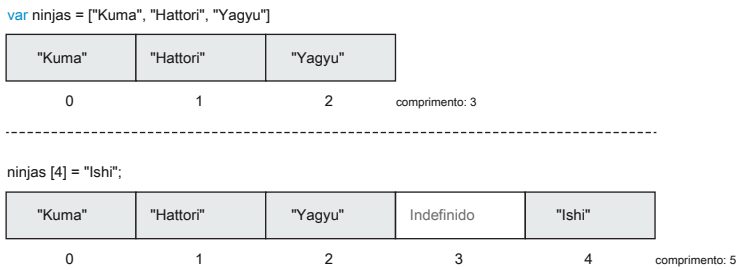


Figura 9.1 Gravar em um índice de array fora dos limites de array expande o array.

Ao contrário da maioria das outras linguagens, em JavaScript, arrays também exibem uma característica peculiar relacionada ao comprimento propriedade: Nada nos impede de alterar manualmente seu valor. Definir um valor maior do que o comprimento atual irá expandir a matriz com Indefinido
itens, ao passo que definir o valor com um valor inferior irá cortar a matriz, como em
ninjas.length = 2 ;.

Agora que passamos pelos fundamentos da criação de array, vamos examinar alguns dos métodos de array mais comuns.

9.1.2 Adicionar e remover itens nas extremidades de uma matriz

Vamos começar com os seguintes métodos simples que podemos usar para adicionar e remover itens de uma matriz:

- Empurre adiciona um item ao final da matriz.
- não mudar adiciona um item ao início da matriz.
- pop remove um item do final da matriz.
- mudança remove um item do início da matriz.

Você provavelmente já usou esses métodos, mas apenas no caso, vamos garantir que estamos na mesma página explorando a lista a seguir.

Listagem 9.2 Adicionando e removendo itens de array

```
const ninjas = [];
assert (ninjas.length === 0, "Um array começa vazio");

ninjas.push ("Kuma");
afirmar (ninjas [0] === "Kuma",
         "Kuma é o primeiro item da matriz");
assert (ninjas.length === 1, "Temos um item na matriz");

ninjas.push ("Hattori");
afirmar (ninjas [0] === "Kuma",
         "Kuma ainda é o primeiro");
assert (ninjas [1] === "Hattori",
         "Hattori é adicionado ao final da matriz"); assert (ninjas.length === 2,

         "Temos dois itens no array!");

ninjas.unshift ("Yagyu");
assert (ninjas [0] === "Yagyu",
         "Agora Yagyu é o primeiro item"); afirmar (ninjas [1] ===
"Kuma",
         "Kuma passou para o segundo lugar"); assert (ninjas [2] ===
"Hattori",
         "E Hattori para o terceiro lugar"); assert (ninjas.length === 3,

         "Temos três itens no array!");

const lastNinja = ninjas.pop (); assert (lastNinja ===
"Hattori",
         "Removemos o Hattori do final da matriz"); assert (ninjas [0] === "Yagyu",

         "Agora Yagyu ainda é o primeiro item"); afirmar (ninjas [1] === "Kuma,

         "Kuma ainda está em segundo lugar"); assert (ninjas.length ===
2,
         "Agora, existem dois itens na matriz");

const firstNinja = ninjas.shift (); assert (firstNinja === "Yagyu",

         "Removemos Yagyu do início da matriz"); afirmar (ninjas [0] === "Kuma",

         "Kuma mudou para o primeiro lugar"); assert (ninjas.length === 1,

         "Há apenas um ninja na matriz");
```

Cria um novo, array vazio

Empurra um novo item até o fim da matriz

Empurra outro item até o fim da matriz

Usa o método integrado unshift para inserir o item no início da matriz. Outros itens são ajustados de acordo.

Retira o último item do array

Remove o primeiro item da matriz. Outros itens são movido para a esquerda em conformidade.

Neste exemplo, primeiro criamos um novo e vazio ninjas variedade:

```
ninjas = [] // ninjas: []
```

Em cada array, podemos usar o embutido Empurre método para anexar um item ao final da matriz, alterando seu comprimento no processo:

```
ninjas.push ("Kuma"); // ninjas: ["Kuma"];
ninjas.push ("Hattori"); // ninjas: ["Kuma", "Hattori"];
```

Também podemos adicionar novos itens ao início da matriz usando o não mudar
método:

```
ninjas.unshift ("Yagyu"); // ninjas: ["Yagyu", "Kuma", "Hattori"];
```

Observe como os itens da matriz existentes são ajustados. Por exemplo, antes de chamar o não mudar
método, " Kuma " estava no índice 0, e depois está no índice 1

Também podemos remover elementos do final ou do início do array. Chamando o integrado pop método remove um elemento do final da matriz, reduzindo o comprimento da matriz no processo:

```
var lastNinja = ninjas.pop (); // ninjas: ["Yagyu", "Kuma"]
                                                    // lastNinja: "Hattori"
```

Também podemos remover um item do início da matriz usando o
mudança método:

```
var firstNinja = ninjas.shift (); // ninjas: ["Kuma"]
                                                    // firstNinja: "Yagyu"
```

A Figura 9.2 mostra como empurre, estale, mude, e não mudar modificar matrizes.

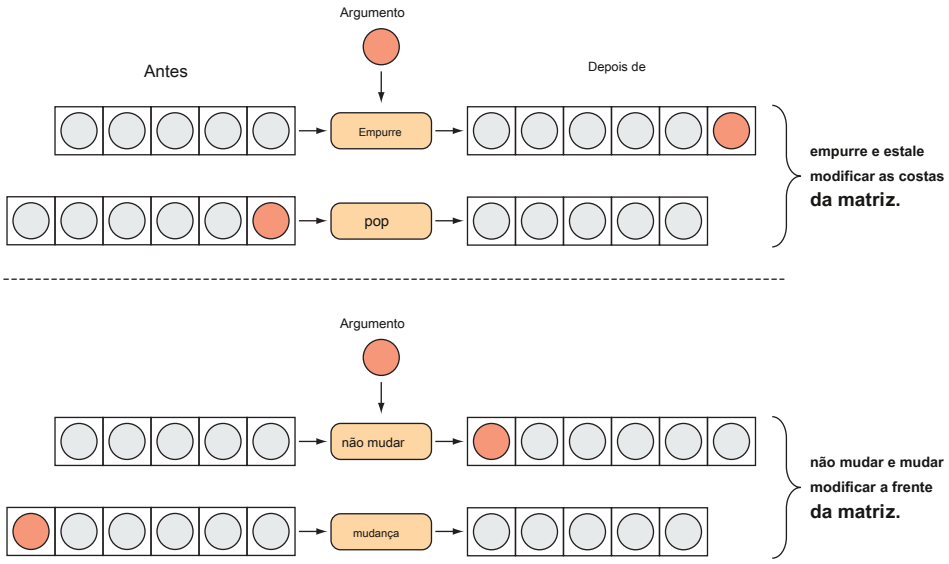


Figura 9.2 O **Empurre** e **pop** métodos modificam o final de uma matriz, enquanto **mudança** e **não mudar** modificar o início do array.

Considerações de desempenho: pop e push versus shift e unshift

O pop e Empurre métodos afetam apenas o último item em uma matriz: pop removendo o último item, e Empurre inserindo um item no final da matriz. Por outro lado, o mudança e não mudar métodos alteram o primeiro item na matriz. Isso significa que os índices de quaisquer itens subsequentes do array devem ser ajustados. Por esta razão, Empurre

e pop são operações significativamente mais rápidas do que mudança e sem mudança, e recomendamos usá-los, a menos que você tenha um bom motivo para fazer o contrário.

9.1.3 Adicionar e remover itens em qualquer localização da matriz

O exemplo anterior removeu itens do início e do final da matriz. Mas isso é muito restritivo - em geral, devemos ser capazes de remover itens de qualquer localização de array. Uma abordagem direta para fazer isso é mostrada na lista a seguir.

Listagem 9.3 Maneira ingênua de remover um item de array

```
const ninjas = ["Yagyu", "Kuma", "Hattori", "Fuma"];

exclua ninjas [1];                          <-- Usa o comando delete para deletar um item

assert (ninjas.length === 4,
        "Comprimento ainda relata que existem 4 itens");

assert (ninjas [0] === "Yagyu", "O primeiro item é Yagyu"); assert (ninjas [1] === undefined, "Nós simplesmente criamos um
buraco"); assert (ninjas [2] === "Hattori", "Hattori ainda é o terceiro item"); assert (ninjas [3] === "Fuma", "E Fuma é o último
item");

                                            Excluímos um item, mas a matriz ainda relata que possui 4 itens.
                                            Nós apenas criamos um buraco na matriz.
```

Esta abordagem para excluir um item de uma matriz não funciona. Efetivamente, apenas criamos uma lacuna na matriz. O array ainda informa que tem quatro itens, mas um deles - o que queríamos excluir - é Indefinido (veja a figura 9.3).

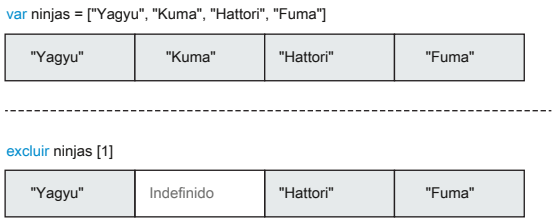


Figura 9.3 Excluir um item de uma matriz cria um buraco na matriz.

Da mesma forma, se quiséssemos inserir um item em uma posição arbitrária, por onde começaríamos? Como uma resposta a esses problemas, todas as matrizes de JavaScript têm acesso ao emenda
método: a partir de um determinado índice, este método remove e insere itens. Veja o seguinte exemplo.

Listagem 9.4 Removendo e adicionando itens em posições arbitrárias

```
const ninjas = ["Yagyu", "Kuma", "Hattori", "Fuma"];

var removedItems = ninjas.splice (1, 1);

assert (removedItems.length === 1, "Um item foi removido"); assert (removedItems [0] === "Kuma");

assert (ninjas.length === 3,
        "Existem agora três itens na matriz"); assert (ninjas [0] === "Yagyu",

        "O primeiro item ainda é Yagyu"); assert (ninjas [1] ===
"Hattori",
        "Hattori está agora em segundo lugar"); afirmar (ninjas [2] === "Fuma",

        “E Fuma está em terceiro lugar”);

removedItems = ninjas.splice (1, 2, "Mochizuki", "Yoshi", "Momochi"); assert (removedItems.length === 2, "Agora, removemos dois
itens"); assert (removedItems [0] === "Hattori", "Hattori foi removido"); assert (removedItems [1] === "Fuma", "Fuma foi removido");
assert (ninjas.length === 4, "Inserimos alguns itens novos"); assert (ninjas [0] === "Yagyu", "Yagyu ainda está aqui");

afirmam (ninjas [1] === "Mochizuki", "Mochizuki também"); assert (ninjas [2] === "Yoshi", "Yoshi
também"); afirmam (ninjas [3] === "Momochi", "e Momochi");
```

Cria uma nova matriz com quatro itens

Usa o embutido método de emenda para remova um elemento, começando no índice 1

splice retorna uma matriz dos itens removidos. Neste caso, nós removeu um item.

A matriz ninja não contém mais Kuma; itens subsequentes foram mudou automaticamente.

Podemos inserir um elemento em uma posição adicionando argumentos à chamada de splice.

Começamos criando uma nova matriz com quatro itens:

```
var ninjas = ["Yagyu", "Kuma", "Hattori", "Fuma"];
```

Então chamamos o embutido emenda método:

```
var removedItems = ninjas.splice (1,1); // ninjas: ["Yagyu", "Hattori", "Fuma"];
                                                    // removedItems: ["Kuma"]
```

Nesse caso, emenda recebe dois argumentos: o índice a partir do qual a emenda começa e o número de elementos a serem removidos (se omitirmos este argumento, todos os elementos até o final da matriz serão removidos). Neste caso, o elemento com índice 1 é removido da matriz e todos os elementos subsequentes são deslocados de acordo.

Além disso, o emenda método retorna uma matriz de itens que foram removidos. Nesse caso, o resultado é uma matriz com um único item: " Kuma ".

Usando o emenda , também podemos inserir itens em posições arbitrárias em uma matriz. Por exemplo, considere o seguinte código:

```
removedItems = ninjas.splice (1, 2, "Mochizuki", "Yoshi", "Momochi"); // ninjas: ["Yagyu", "Mochizuki", "Yoshi", "Momochi"]

// removedItems: ["Hattori", "Fuma"]
```

A partir do índice 1, primeiro remove dois itens e depois adiciona três itens: " Mochi-
zuki "," Yoshi ", e " Momochi ".

Agora que relembramos como funcionam os arrays, vamos continuar estudando algumas operações comuns que costumam ser realizadas em arrays. Isso o ajudará a escrever um código de manipulação de array mais elegante.

9.1.4 *Operações comuns em matrizes*

Nesta seção, exploraremos algumas das operações mais comuns em matrizes:

- *Iterando (* ou *atravessando)* através de matrizes
- *Mapeamento* itens da matriz existente para criar uma nova matriz com base neles
- *Testando* itens de matriz para verificar se eles satisfazem certas condições
- *Encontrando* itens específicos da matriz
- *Agregando* matrizes e computar um único valor com base em itens de matriz (por exemplo, calcular a soma de uma matriz)

Bem comece com o básico: iterações de array.

ITERANDO SOBRE ARRAYS

Uma das operações mais comuns é a iteração em uma matriz. Voltando à Ciência da Computação 101, uma iteração é mais frequentemente realizada da seguinte maneira:

```
const ninjas = ["Yagyu", "Kuma", "Hattori"];

para (seja i = 0; i <ninjas.length; i ++) {
    afirmar (ninjas [i]! == null, ninjas [i]);
}
```

Informa o valor de cada ninja

Este exemplo é tão simples quanto parece. Usa um pra loop para verificar todos os itens na matriz; os resultados são mostrados na figura 9.4.

Você provavelmente já escreveu algo assim tantas vezes que nem precisa pensar mais nisso. Mas, por precaução, vamos dar uma olhada mais de perto no pra ciclo.

Figura 9.4 A saída da verificação dos ninjas com um **pra** ciclo

Para percorrer uma matriz, temos que configurar uma variável de contador, eu, especificar o número até o qual queremos contar (ninjas.length), e

definir como o contador será modificado (i ++). É uma quantidade enorme de contabilidade para realizar uma ação tão comum e pode ser uma fonte de pequenos bugs irritantes. Além disso, torna nosso código mais difícil de ler. Os leitores devem olhar atentamente para cada parte do pra loop, apenas para ter certeza de que passa por todos os itens e não pula nenhum.

Para tornar a vida mais fácil, todos os arrays de JavaScript têm um built-in para cada método que podemos usar em tais situações. Veja o exemplo a seguir.

Listagem 9.5 Usando o **para cada** método

```
const ninjas = ["Yagyu", "Kuma", "Hattori"];

ninjas. para cada( ninja => {
    afirmar (ninja! == nulo, ninja);
));
```

Usa o embutido método forEach para iterar sobre a matriz

Fornecemos um retorno de chamada (neste caso, uma função de seta) que é chamado *imediatamente,* para cada item da matriz. É isso - não precisa mais se preocupar com o índice inicial, a condição final ou a natureza exata do incremento. O mecanismo JavaScript cuida de tudo isso para nós, nos bastidores. Observe como esse código é muito mais fácil de entender e como ele tem menos pontos de geração de bug.

Continuaremos elevando as coisas e vendo como podemos mapear arrays para outros arrays.

ARRAYS DE MAPEAMENTO

Imagine que você tem uma série de ninja objetos. Cada ninja tem um nome e uma arma favorita, e você deseja extrair uma variedade de armas do ninjas variedade. Armado com o conhecimento do para cada método, você pode escrever algo como a lista a seguir.

Listagem 9.6 Extração ingênua de uma matriz de armas

```
const ninjas = [
    {nome: "Yagyu", arma: "shuriken"}, {nome: "Yoshi", arma: "katana"},
    {nome: "Kuma", arma: "wakizashi"}

];

armas const = [];
ninjas.forEach (ninja => {
    weapons.push (ninja.weapon);
});

afirmar (armas [0] === "shuriken"
        && armas [1] === "katana"
        && armas [2] === "wakizashi" && weapons.length ===
        3,
        "O novo array contém todas as armas");
```

Cria uma nova matriz e usa um loop forEach sobre ninjas para extrair armas ninja individuais

Isso não é tão ruim: criamos um novo array vazio e usamos o para cada método para iterar sobre o ninjas variedade. Então, para cada ninja objeto, adicionamos a arma atual ao armas variedade.

Como você pode imaginar, criar novos arrays com base nos itens de um array existente é surpreendentemente comum - tão comum que tem um nome especial: *mapeamento* uma matriz. A ideia é mapear cada item de um array para um novo item de um novo array. Convenientemente, o JavaScript tem um mapa função que faz exatamente isso, conforme mostrado na lista a seguir.

Listagem 9.7 Mapeando um array

```
const ninjas = [
    {nome: "Yagyu", arma: "shuriken"}, {nome: "Yoshi", arma: "katana"},
    {nome: "Kuma", arma: "wakizashi"}

];

armas const = ninjas.map (ninja => ninja.weapon);

afirmar (armas [0] === "shuriken"
        && armas [1] === "katana"
        && armas [2] === "wakizashi"
        && weapons.length == 3, "O novo array contém todas as armas");
```

O método de mapa embutido usa uma função que é chamada para cada item da matriz.

O embutido mapa método constrói um array completamente novo e então itera sobre o array de entrada. Para cada item na matriz de entrada, mapa coloca exatamente um item na matriz recém-construída, com base no resultado do retorno de chamada fornecido para mapa. O funcionamento interno do mapa função são mostradas na figura 9.5.

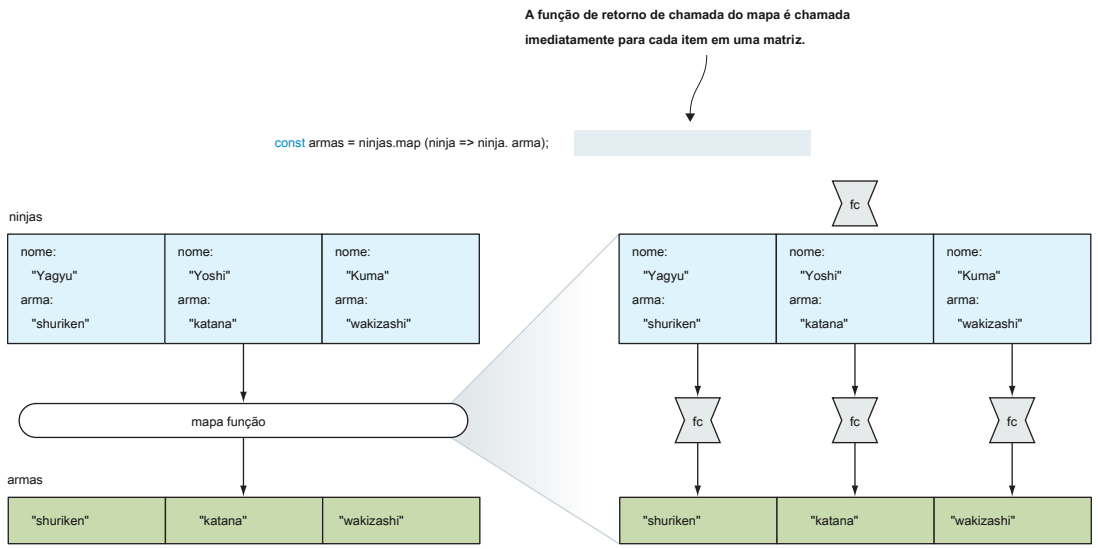


Figura 9.5 O **mapa** function chama a função de retorno de chamada fornecida (fc) em cada item da matriz e cria uma nova matriz com valores de retorno de chamada.

Agora que sabemos como mapear arrays, vamos ver como testar itens de array para certas condições.

ARRAY ITEMS DE TESTE

Ao trabalhar com coleções de itens, muitas vezes nos deparamos com situações em que precisamos saber se *todo* ou pelo menos *alguns* dos itens da matriz satisfazem certas condições. Para escrever este código da forma mais eficiente possível, todos os arrays JavaScript têm acesso ao
cada e alguns métodos, mostrados a seguir.

Listagem 9.8 Testando matrizes com o **cada** e **alguns** métodos

```
const ninjas = [
    {nome: "Yagyu", arma: "shuriken"}, {nome: "Yoshi"},

    {nome: "Kuma", arma: "wakizashi"}
];

const allNinjasAreNamed = ninjas.every (ninja => "nome" em ninja); const allNinjasAreArmed = ninjas.every (ninja => "arma" em
ninja);

assert (allNinjasAreNamed, "Todo ninja tem um nome"); assert (! allNinjasAreArmed, "Mas nem todo ninja está
armado");

const someNinjasAreArmed = ninjas.some (ninja => "arma" em ninja); assert (someNinjasAreArmed, "Mas alguns ninjas estão
armados");
```

O método every integrado recebe um retorno de chamada que é chamado para cada item do array. Ele retorna verdadeiro se o retorno de chamada retornar um valor verdadeiro para *todo* variedade itens, ou falso caso contrário.

O método embutido também recebe um retorno de chamada. Ele retorna verdadeiro se o retorno de chamada retornar um valor verdadeiro para em pelo menos um item da matriz, ou falso caso contrário.

A Listagem 9.8 mostra um exemplo onde temos uma coleção de ninja objetos, mas não têm certeza de seus nomes e se todos eles estão armados. Para chegar à raiz deste problema, primeiro aproveitamos cada:

```
var allNinjasAreNamed = ninjas. cada( ninja => "nome" em ninja);
```

O cada O método recebe um retorno de chamada que, para cada ninja da coleção, verifica se sabemos o nome do ninja. cada retorna verdade apenas se o retorno de chamada passado retornar verdade pra *cada* item na matriz. A Figura 9.6 mostra como cada funciona.

Em outros casos, só nos importamos se *alguns* os itens da matriz satisfazem uma determinada condição. Para essas situações, podemos usar o método integrado alguns:

```
const someNinjasAreArmed = ninjas.some (ninja => "arma" em ninja);
```

Começando com o primeiro item da matriz, alguns chama o retorno de chamada em cada item da matriz até que um item seja encontrado para o qual o retorno de chamada retorna um verdade valor. Se tal item for encontrado, o valor de retorno é verdade; se não, o valor de retorno é falso.

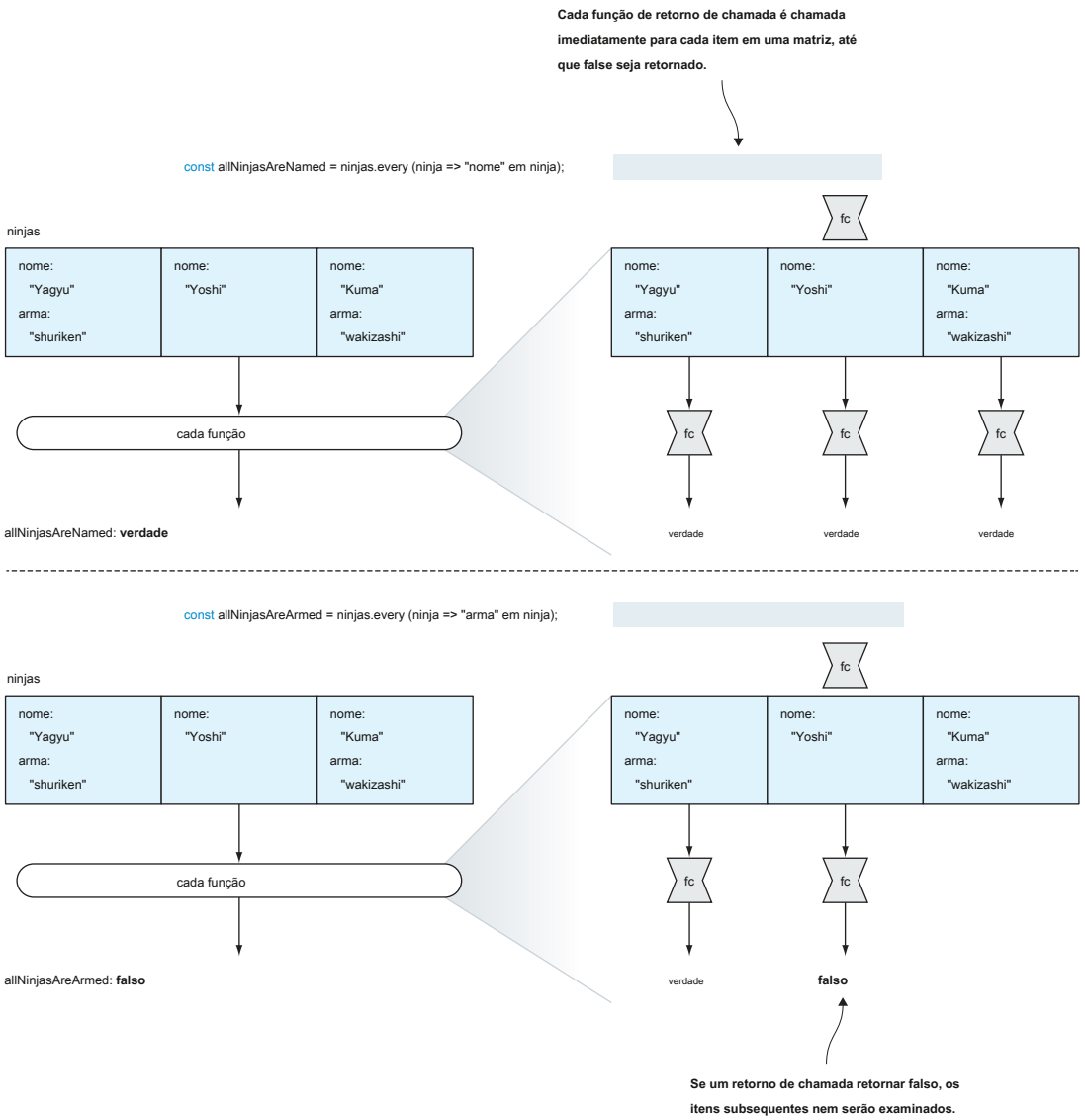


Figura 9.6 O **cada** método testa se todos os itens em uma matriz satisfazem uma determinada condição representada por um retorno de chamada.

A Figura 9.7 mostra como alguns funciona nos bastidores: Pesquisamos uma matriz para descobrir se alguns ou todos os seus itens satisfazem uma determinada condição. A seguir, exploraremos como pesquisar um array para encontrar um item específico.

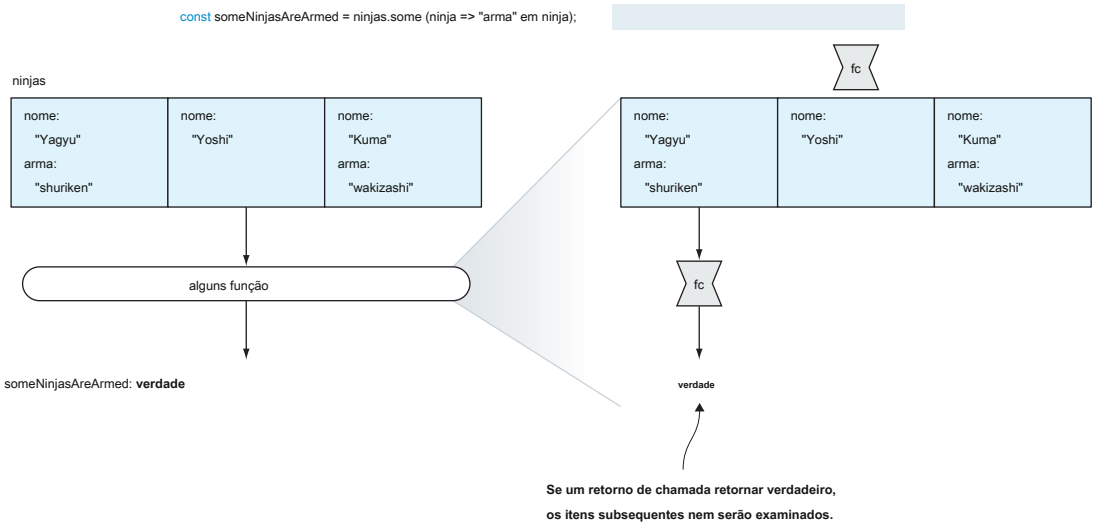


Figura 9.7 O **alguns** O método verifica se pelo menos um item da matriz satisfaz uma condição representada pelo retorno de chamada passado.

PESQUISANDO ARRAYS

Outra operação comum que você deve usar, mais cedo ou mais tarde, é localizar itens em um array. Novamente, essa tarefa é bastante simplificada com outro método de array integrado: achar. Vamos estudar a seguinte lista.

NOTA O embutido achar método faz parte do padrão ES6. Para compatibilidade do navegador atual, consulte http://mng.bz/U532 .

Listagem 9.9 Encontrando itens do array

```
const ninjas = [
    {nome: "Yagyu", arma: "shuriken"}, {nome: "Yoshi"},

    {nome: "Kuma", arma: "wakizashi"}
];

const ninjaWithWakizashi = ninjas.find (ninja => {
    return ninja.weapon === "wakizashi"; });


assert (ninjaWithWakizashi.name === "Kuma"
        && ninjaWithWakizashi.weapon === "wakizashi", "Kuma está empunhando um
        wakizashi");
```

Usa o método find para encontrar o primeiro item da matriz que satisfaça uma certa condição, representado por um retorno de chamada passado.

```
const ninjaWithKatana = ninjas.find (nina => {
    return ninja.weapon === "katana"; });

assert (ninjaWithKatana === undefined,
          "Não conseguimos encontrar um ninja que empunhava uma katana");

const armadoNinjas = ninjas.filter (ninja => "arma" em ninja);

assert (armadoNinjas.length === 2, "Existem dois ninjas armados:"); assert (armadoNinjas [0] .name === "Yagyu"

        && armadoNinjas [1] .nome === "Kuma", "Yagyu e Kuma");
```

O método find retorna undefined se um item não puder ser encontrado.

Use o método de filtro para encontrar vários itens que satisfaçam uma determinada condição.

É fácil encontrar um item de matriz que satisfaça uma determinada condição: Usamos o achar , passando a ele um retorno de chamada que é invocado para cada item na coleção até que o item de destino seja encontrado. Isso é indicado pelo retorno de chamada verdade. Por exemplo, a expressão

```
ninjas.find (ninja => ninja.weapon === "wakizashi");
```

encontra Kuma, o primeiro ninja do ninjas matriz que está empunhando um wakizashi.

Se nós examinamos toda a matriz sem um único item retornando verdade, o resultado final da pesquisa é Indefinido. Por exemplo, o código

```
ninjaWithKatana = ninjas.find (ninja => ninja.weapon === "katana");
```

retorna Indefinido, porque não há um ninja empunhando uma katana. A Figura 9.8 mostra o funcionamento interno do achar função.

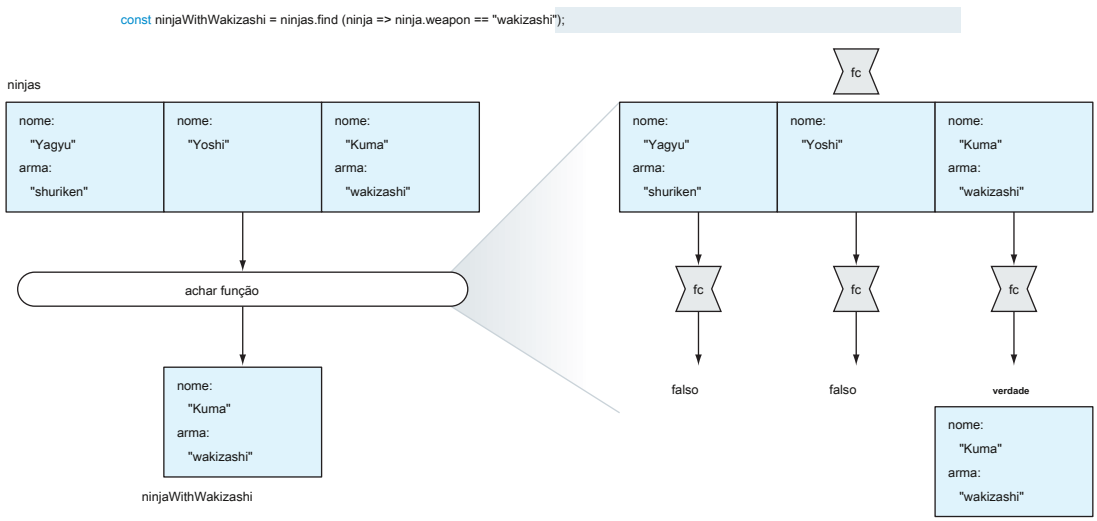


Figura 9.8 O **achar** função encontra um item em uma matriz: o primeiro item para o qual o **achar** retorno de chamada **verdade.**

Se precisarmos encontrar vários itens que satisfaçam um determinado critério, podemos usar o filtro método, que cria um *novo* array contendo todos os itens que atendem a esse critério. Por exemplo, a expressão

```
const armadoNinjas = ninjas.filter (ninja => "arma" em ninja);
```

cria um novo Armado Ninjas array que contém apenas ninja s com uma arma. Nesse caso, o pobre Yoshi desarmado fica de fora. A Figura 9.9 mostra como o filtro função funciona.

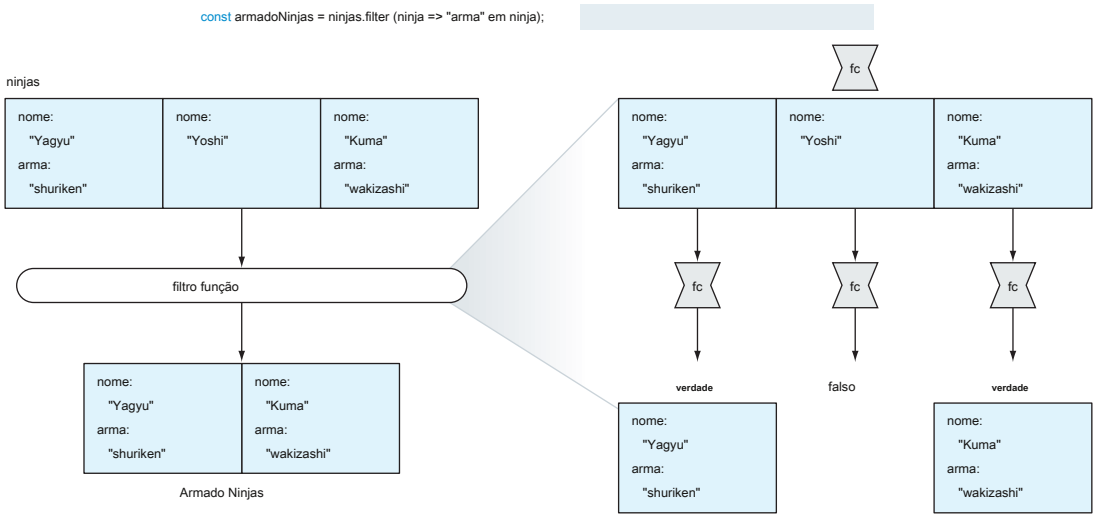


Figura 9.9 O **filtro** função cria uma nova matriz que contém todos os itens para os quais o retorno de chamada retorna **verdade.**

Ao longo deste exemplo, você viu como encontrar itens específicos em uma matriz, mas em muitos casos também pode ser necessário encontrar o índice de um item. Vamos dar uma olhada mais de perto, com o exemplo a seguir.

Listagem 9.10 Encontrando índices de array

```
const ninjas = ["Yagyu", "Yoshi", "Kuma", "Yoshi"];

assert (ninjas.indexOf ("Yoshi") === 1, "Yoshi está no índice 1"); assert (ninjas.lastIndexOf ("Yoshi") === 3, "e no
índice 3");

const yoshiIndex = ninjas.findIndex (ninja => ninja === "Yoshi");

assert (yoshiIndex === 1, "Yoshi ainda está no índice 1");
```

Para encontrar o índice de um item específico, usamos a índice de método, passando para ele o item cujo índice queremos encontrar:

```
ninjas.indexOf ("Yoshi")
```

Nos casos em que um determinado item pode ser encontrado várias vezes em uma matriz (como é o caso de " Yoshi " e a ninjas matriz), também podemos estar interessados em encontrar o último índice em que Yoshi aparece. Para isso, podemos usar o lastIndexOf método:

```
ninjas.lastIndexOf ("Yoshi")
```

Finalmente, no caso mais geral, quando não temos uma referência ao item exato cujo índice queremos pesquisar, podemos usar o findIndex método:

```
const yoshiIndex = ninjas.findIndex (ninja => ninja === "Yoshi");
```

O findIndex método recebe um retorno de chamada e retorna o índice do primeiro item para o qual o retorno de chamada retorna verdade. Em essência, funciona muito como o achar método, a única diferença é que achar retorna um item específico, enquanto findIndex retorna o índice desse item.

SORTING ARRAYS

Uma das operações de array mais comuns é *Ordenação*- organizar os itens sistematicamente em alguma ordem. Infelizmente, implementar algoritmos de classificação corretamente não é a mais fácil das tarefas de programação: temos que selecionar o melhor algoritmo de classificação para a tarefa, implementá-lo e adaptá-lo às nossas necessidades, enquanto, como sempre, tomamos cuidado para não introduzir bugs sutis. Para tirar esse fardo de nossas costas, como você viu no capítulo 3, todos os arrays de JavaScript têm acesso ao organizar método, cujo uso é semelhante a este:

```
array.sort ((a, b) => a - b);
```

O mecanismo JavaScript implementa o algoritmo de classificação. A única coisa que temos a fornecer é um retorno de chamada que informa o algoritmo de classificação sobre a relação entre dois itens do array. Os resultados possíveis são os seguintes:

- Se um retorno de chamada retornar um valor menor que 0, então item uma deve vir antes do item b.
- Se um retorno de chamada retornar um valor igual a 0, então itens uma e b estão em pé de igualdade (no que diz respeito ao algoritmo de classificação, eles são iguais).
- Se um retorno de chamada retornar um valor maior que 0, então item uma deve vir depois do item b.

A Figura 9.10 mostra as decisões tomadas pelo algoritmo de classificação dependendo do valor de retorno de chamada.

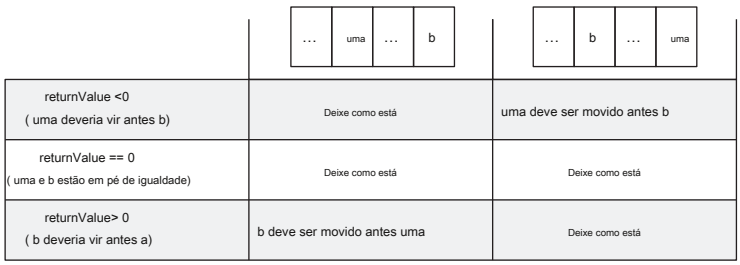


Figura 9.10 Se a chamada back retorna um valor menor que **0,** o primeiro item deve vir antes do segundo. Se o retorno de chamada retornar **0,** Ambas os itens devem ser deixados como estão. E se o valor de retorno for maior que **0,** o primeiro item deve vir após o segundo item.

E isso é tudo que você precisa saber sobre o algoritmo de classificação. A classificação real é realizada nos bastidores, sem a necessidade de mover manualmente os itens do array. Vejamos um exemplo simples.

Listagem 9.11 Classificando um array

```
const ninjas = [{nome: "Yoshi"}, {nome: "Yagyu"}, {nome: "Kuma"}];

ninjas.sort (função (ninja1, ninja2) {
    if (ninja1.nome <ninja2.nome) {retorno -1; } if (ninja1.name> ninja2.name) {return 1; }

    return 0;
});

assert (ninjas [0] .name === "Kuma", "Kuma é o primeiro"); assert (ninjas [1] .name === "Yagyu", "Yagyu
é o segundo"); assert (ninjas [2] .name === "Yoshi", "Yoshi é o terceiro");
```

Passa um retorno de chamada para o método de classificação integrado para especificar uma ordem de classificação

Na listagem 9.11, temos uma matriz de ninja objetos, onde cada ninja tem um nome. Nosso objetivo é classificar esse array lexicograficamente (em ordem alfabética), de acordo com os nomes dos ninjas. Para isso, naturalmente usamos o organizar função:

```
ninjas.sort (função (ninja1, ninja2) {
    if (ninja1.nome <ninja2.nome) {retorno -1; } if (ninja1.name> ninja2.name) {return 1; }

    return 0;
});
```

Ao organizar só precisamos passar um retorno de chamada usado para comparar dois itens do array. Porque queremos fazer uma comparação lexical, afirmamos que se ninja1 o nome de é “menor” que ninja2 nome de, o retorno de chamada retorna - 1 (lembre-se, isso significa ninja1 deveria vir antes ninja2, na ordem final de classificação); se for maior, o retorno de chamada retorna 1
(ninja1 deveria vir depois ninja2); se forem iguais, o retorno de chamada retorna 0 Observe que podemos usar operadores simples menor que (<) e maior que (>) para comparar dois nomes de ninja.

É sobre isso! O resto dos detalhes essenciais da classificação são deixados para o mecanismo JavaScript, sem que tenhamos que nos preocupar com eles.

AGREGANDO ARRAY ITMS

Quantas vezes você escreveu um código como o seguinte?

```
números const = [1, 2, 3, 4]; soma const = 0;

number.forEach (number => {
    soma + = número;
});

assert (sum === 10, "A soma dos quatro primeiros números é 10");
```

Esse código deve visitar cada item de uma coleção e agregar algum valor, basicamente reduzindo todo o array a um único valor. Não se preocupe - o JavaScript também tem algo para ajudar nessa situação: o reduzir método, conforme mostrado no exemplo a seguir.

Listagem 9.12 Agregando itens com reduzir

```
números const = [1, 2, 3, 4];

const sum = number.reduce ((agregado, número) =>
                                                    agregado + número, 0);

assert (sum === 10, "A soma dos quatro primeiros números é 10");
```

Usa reduzir para acumular um único valor de uma matriz

O reduzir método funciona pegando o valor inicial (neste caso, 0) e chamar a função de retorno de chamada em cada item da matriz com o resultado da chamada de retorno anterior (ou o valor inicial) e o item da matriz atual como argumentos. O resultado do reduzir
invocação é o resultado do último retorno de chamada, chamado no último item do array. A Figura 9.11 lança mais luz sobre o processo.

Esperamos ter convencido você de que o JavaScript contém alguns métodos de array úteis que podem tornar nossas vidas significativamente mais fáceis e seu código mais elegante, sem ter que recorrer a métodos incômodos pra rotações. Se você quiser saber mais sobre esses e outros métodos de array, recomendamos a explicação da rede do desenvolvedor Mozilla em http://mng.bz/cS21 . Agora, levaremos as coisas um pouco mais adiante e mostraremos como reutilizar esses métodos de array em seus próprios objetos personalizados.

9.1.5 *Reutilizar funções de matriz integradas*

Às vezes, podemos querer criar um objeto que contenha uma coleção de dados. Se a coleção fosse tudo que nos preocupasse, poderíamos usar um array. Mas, em certos casos, pode haver mais estado para armazenar do que apenas a coleção em si - talvez precisemos armazenar algum tipo de metadados sobre os itens coletados.

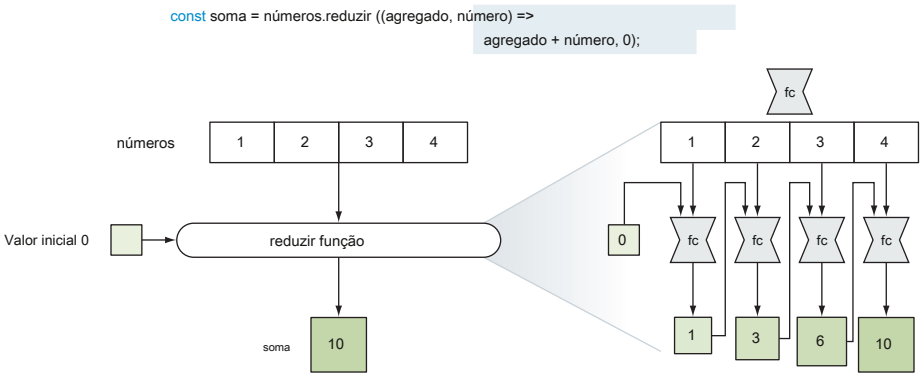


Figura 9.11 O **reduzir** A função aplica um retorno de chamada a um valor agregado e a cada item em uma matriz para reduzir a matriz a um único valor.

Uma opção pode ser criar um novo array sempre que desejar criar uma nova versão de tal objeto e adicionar as propriedades e métodos de metadados a ele. Lembre-se de que podemos adicionar propriedades e métodos a um objeto como desejarmos, incluindo matrizes. Geralmente, entretanto, isso pode ser lento, para não mencionar tedioso.

Vamos examinar a possibilidade de usar um objeto normal e *dando* é a funcionalidade que desejamos. Métodos que sabem como lidar com coleções já existem no Variedade
objeto; podemos enganá-los para trabalhar em nossos próprios objetos? Acontece que podemos, conforme mostrado na lista a seguir.

Listagem 9.13 Simulando métodos semelhantes a array

```
<body>
    <input id = "first" />
    <input id = "second" />
    <script>
        const elems = {
            comprimento: 0,
            add: function (elem) {
                Array.prototype.push.call (this, elem);
            },
            reunir: função (id) {
                this.add (document.getElementById (id));
            },
            find: function (callback) {
                return Array.prototype.find.call (this, callback);
            }
        };

        elems.gather ("primeiro");
        assert (elems.length === 1 && elems [0] .nodeType,
                    "Verifique se temos um elemento em nosso estoque");

        elems.gather ("segundo");
        assert (elems.length === 2 && elems [1] .nodeType,
                    “Verifique a outra inserção”);

        elems.find (elem => elem.id === "segundo"); assert (found && found.id ===
        "segundo",
                    "Encontramos nosso elemento");
    </script>
</body>
```

Armazena a contagem de elementos. A matriz precisa de um local para armazenar o número de itens que está armazenando.

Implementa o método para adicionar elementos a uma coleção. O protótipo do Array tem um método para fazer isso, então por que não usá-lo em vez de reinventar a roda?

Implementa o método de coleta para encontrar elementos por seus valores de id e adicioná-los à coleção

Implementa o método para encontrar elementos na coleção. Semelhante ao método add, ele reutiliza o método find existente acessível a matrizes.

Neste exemplo, criamos um objeto “normal” e o instrumentamos para imitar alguns dos comportamentos de um array. Primeiro, definimos um comprimento propriedade para registrar o número de elementos que são armazenados, como um array. Em seguida, definimos um método para adicionar um elemento ao final do array simulado, chamando este método adicionar:

```
add: function (elem) {
    Array.prototype.push.call (this, elem);
}
```

Em vez de escrever nosso próprio código, podemos usar um método nativo de matrizes JavaScript: Array.prototype.push.

Normalmente, o Array.prototype.push método operaria em seu próprio array por meio de seu contexto de função. Mas aqui, estamos enganando o método a usar *nosso* objeto como seu contexto usando o chamar método (lembre-se do capítulo 4) e forçando nosso objeto a ser o contexto do Empurre método. (Observe como poderíamos ter usado facilmente o Aplique
método.) Empurre método, que incrementa o comprimento propriedade (pensando que é o comprimento propriedade de uma matriz), adiciona uma propriedade numerada ao objeto que faz referência ao elemento passado. De certa forma, esse comportamento é quase subversivo (como apropriado para ninjas!), Mas exemplifica o que podemos fazer com contextos de objetos mutáveis.

O adicionar método espera que uma referência de elemento seja passada para armazenamento. Embora às vezes possamos ter essa referência por perto, na maioria das vezes não o fazemos, então também definimos um método de conveniência, reunir, que procura o elemento por seu eu ia valor e o adiciona ao armazenamento:

```
reunir: função (id) {
    this.add (document.getElementById (id));
}
```

Finalmente, também definimos um achar método que nos permite encontrar um item arbitrário em nosso objeto personalizado, tirando proveito do array integrado achar método:

```
find: function (callback) {
    return Array.prototype.find.call (this, callback);
}
```

O comportamento nefasto limítrofe que demonstramos nesta seção não apenas revela o poder que os contextos de função maleável nos dão, mas também mostra como podemos ser inteligentes em reutilizar código já escrito, em vez de reinventar constantemente a roda.

Agora que já passamos algum tempo com matrizes, vamos passar para dois novos tipos de coleções introduzidos pelo ES6: mapas e conjuntos.

9,2 Mapas

Imagine que você é um desenvolvedor do freelanceninja.com, um site que deseja atender a um público mais internacional. Para cada texto no site - por exemplo, “Ninjas de aluguel” - você gostaria de criar um mapeamento para cada idioma de destino, como “ レ ン タ ル 用 の 忍者 " em japonês, " 忍者 出租 ”Em chinês, ou“ 고용 닌자 ”Em coreano (esperemos que o Google Translate tenha feito um trabalho adequado). Essas coleções, que mapeiam uma chave para um valor específico, são chamadas por nomes diferentes em linguagens de programação diferentes, mas na maioria das vezes são conhecidas como *dicionários* ou *mapas.*

Mas como você gerencia com eficiência essa localização em JavaScript? Uma abordagem tradicional é aproveitar as vantagens do fato de que os objetos são coleções de propriedades e valores nomeados e criar algo como o seguinte dicionário:

```
dicionário const = {
    "ja": {
        "Ninjas para alugar": " レ ン タ ル 用 の 忍者 "
    },
    "zh": {
        "Ninjas para alugar": " 忍者 出租 "
    },
    "ko": {
        "Ninjas para alugar": " 고용 닌자 "
    }
}
assert (dictionary.ja ["Ninjas para alugar"] === " レ ン タ ル 用 の 忍者 ");
```

À primeira vista, isso pode parecer uma abordagem perfeitamente adequada para esse problema e, para este exemplo, não é nada ruim. Mas, infelizmente, em geral, você não pode confiar nisso.

9.2.1 Não use objetos como mapas

Imagine que em algum lugar do nosso site precisamos acessar a tradução da palavra *construtor,* portanto, estendemos o exemplo do dicionário para o código a seguir.

Listagem 9.14 Objetos têm acesso a propriedades que não foram definidas explicitamente

```
dicionário const = {
    "ja": {
        "Ninjas para alugar": " レ ン タ ル 用 の 忍者 "
    },
    "zh": {
        "Ninjas para alugar": " 忍者 出租 "
    },
    "ko": {
        "Ninjas para alugar": " 고용 닌자 "
    }
};

assert (dictionary.ja ["Ninjas para alugar"] === " レ ン タ ル 用 の 忍者 ",
            "Nós sabemos dizer 'Ninjas de aluguel' em japonês!");

assert (typeof dictionary.ja ["constructor"] === "undefined",
            dicionário.ja ["construtor"]);
```

Tentamos acessar a tradução da palavra *constructor* - uma palavra que esquecemos tolamente de definir em nosso dicionário. Normalmente, nesse caso, esperamos que o dicionário retorne Indefinido. Mas esse não é o resultado, como você pode ver na figura 9.12.

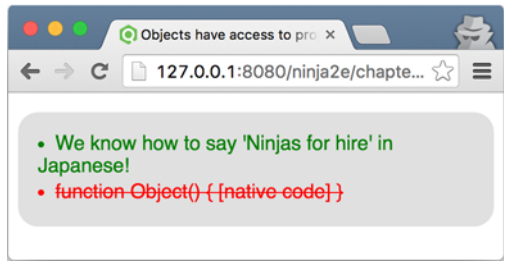


Figura 9.12 Executar a listagem 9.14 mostra que os objetos não são bons mapas, porque eles têm acesso a propriedades que não foram definidas explicitamente (por meio de seus protótipos).

Como você pode ver, acessando o construtor propriedade, obtemos a seguinte string:

```
"function Object () {[código nativo]}"
```

O que há com isso? Como você aprendeu no capítulo 7, todos os objetos têm protótipos; mesmo se definirmos objetos novos e vazios como nossos mapas, eles ainda terão acesso às propriedades dos objetos de protótipo. Uma dessas propriedades é construtor (lembre-se disso constructor é a propriedade do objeto de protótipo que aponta de volta para a função do construtor), e é o culpado por trás da bagunça que agora temos em nossas mãos.

Além disso, com objetos, as chaves podem ser apenas valores de string; se você quiser criar um mapeamento para qualquer outro valor, esse valor será convertido silenciosamente em uma string sem que ninguém pergunte nada! Por exemplo, imagine que queremos rastrear algumas informações sobre nós HTML, como na lista a seguir.

Listagem 9.15 Mapeando valores para nós HTML com objetos

Define dois elementos HTML e os busca usando o método interno document.getElementById

Define um objeto que nós usar como um mapa para loja adicional em formação sobre nosso HTML elementos

Armazena informações sobre o primeiro elemento, e verifica se foi armazenado corretamente

Informações de lojas sobre o segundo elemento e verifica que estava correto armazenado

Themapping pela primeira vez elemento é agora inválido!

```
<div id = "firstElement"> </div>
<div id = "secondElement"> </div>
<script>
    const firstElement = document.getElementById ("firstElement"); const secondElement = document.getElementById
    ("secondElement");

    mapa const = {};

    mapa [firstElement] = {data: "firstElement"}; assert (map [firstElement] .data === "firstElement",
        "O primeiro elemento está mapeado corretamente");

    mapa [secondElement] = {data: "secondElement"}; assert (map [secondElement] .data ===
    "secondElement",
        "O segundo elemento está mapeado corretamente");

    assert (map [firstElement] .data === "firstElement",
        "Mas agora o firstElement foi anulado!"); </script>
```

Na listagem 9.15, criamos dois elementos HTML, firstElement e secondElement, que então buscamos do DOM usando o document.getElementById método. Para criar um mapeamento que armazenará informações adicionais sobre cada elemento, definimos um objeto JavaScript simples e antigo:

```
mapa const = {};
```

Em seguida, usamos o elemento HTML como uma chave para nosso objeto de mapeamento e associamos alguns dados a ele:

```
mapa [firstElement] = {data: "firstElement"}
```

E verificamos se podemos recuperar esses dados. Como isso funciona como deveria, repetimos todo o processo para o segundo elemento:

```
mapa [secondElement] = {data: "secondElement"};
```

Novamente, tudo parece ótimo; associamos com sucesso alguns dados ao nosso elemento HTML. Mas ocorre um problema se decidirmos revisitar o primeiro elemento:

```
map [firstElement] .data
```

Seria normal supor que obteríamos novamente as informações sobre o primeiro elemento, mas não é o caso. Em vez disso, como mostra a figura 9.13, as informações sobre o segundo elemento são retornadas.

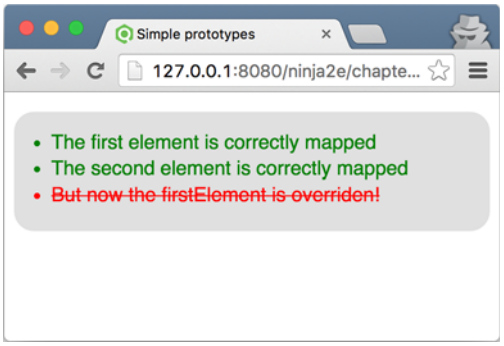


Figura 9.13 Executar o código da listagem 9.15 mostra que os objetos são convertidos em strings se tentarmos usá-los como propriedades do objeto.

Isso acontece porque, com objetos, as chaves são armazenadas como strings. Isso significa que quando tentamos usar qualquer valor não string, como um elemento HTML, como uma propriedade de um objeto, esse valor é silenciosamente convertido em uma string chamando seu método. Aqui, isso retorna a string "[object HTMLDivElement]", e a informações sobre o primeiro elemento são armazenadas como o valor do [objeto HTMLDivElement] propriedade.

Em seguida, quando tentamos criar um mapeamento para o segundo elemento, algo semelhante acontece. O segundo elemento, que também é um elemento HTML div, também é convertido em uma string e seus dados adicionais também são atribuídos ao [objeto HTMLDivElement] , substituindo o valor que definimos para o primeiro elemento.

Por esses dois motivos - propriedades herdadas por meio de protótipos e suporte para chaves somente de string - objetos simples geralmente não são úteis como mapas. Devido a esta limitação, o comitê ECMAScript especificou um tipo completamente novo: Mapa.

NOTA Os mapas são parte do padrão ES6. Para compatibilidade do navegador atual, consulte: http://mng.bz/JYYM .

9.2.2 Criando nosso primeiro mapa

Criar mapas é fácil: usamos um novo e integrado Mapa construtor. Veja o exemplo a seguir.

Listagem 9.16 Criando nosso primeiro mapa

```
const ninjaIslandMap = new Map ();

const ninja1 = {nome: "Yoshi"}; const ninja2 = {nome: "Hattori"};
const ninja3 = {nome: "Kuma"};

ninjaIslandMap.set (ninja1, {homeIsland: "Honshu"}); ninjaIslandMap.set (ninja2, {homeIsland:
"Hokkaido"});

assert (ninjaIslandMap.get (ninja1) .homeIsland === "Honshu",
        "O primeiro mapeamento funciona");
assert (ninjaIslandMap.get (ninja2) .homeIsland === "Hokkaido",
        "O segundo mapeamento funciona");

assert (ninjaIslandMap.get (ninja3) === undefined,
        "Não há mapeamento para o terceiro ninja!");

assert (ninjaIslandMap.size === 2,
        "Criamos dois mapeamentos");

afirmar (ninjaIslandMap.has (ninja1)
      && ninjaIslandMap.has (ninja2),
        "Temos mapeamentos para os dois primeiros ninjas"); assert (! ninjaIslandMap.has
(ninja3),
        "Mas não para o terceiro ninja!");

ninjaIslandMap.delete (ninja1);
assert (! ninjaIslandMap.has (ninja1)
      && ninjaIslandMap.size () === 1,
        "Não há mais mapeamento do primeiro ninja!");

ninjaIslandMap.clear ();
assert (ninjaIslandMap.size === 0,
        "Todos os mapeamentos foram limpos");
```

Usa o construtor de mapa para criar um mapa

Define três objetos ninja

Cria um mapeamento para os dois primeiros objetos ninja usando o método map set

Obtém o mapeamento para os primeiros dois objetos ninja por usando o método map get

Verifica se não há mapeamento para o terceiro ninja

Verifica se o mapa contém mapeamentos para os dois primeiros ninjas, mas não para o terceiro!

Usa o método has para verificar se existe um mapeamento para uma determinada chave

Usa o método delete para deletar uma chave do mapa

Usa o método claro para limpar completamente o mapa

Neste exemplo, criamos um novo mapa chamando o Mapa construtor:

```
const ninjaIslandMap = new Map ();
```

Em seguida, criamos três objetos ninja, habilmente chamados ninja1, ninja2, e ninja3. Em seguida, usamos o mapa integrado definir método:

```
ninjaIslandMap.set (ninja1, {homeIsland: "Honshu"});
```

Isso cria um mapeamento entre uma chave - neste caso, o ninja1 objeto - e um valor - neste caso, um objeto que carrega as informações sobre a ilha natal do ninja. Fazemos isso para os dois primeiros ninjas, ninja1 e ninja2.

Na próxima etapa, obtemos o mapeamento para os dois primeiros ninjas usando outro método de mapa integrado, obter:

```
assert (ninjaIslandMap.get (ninja1) .homeIsland === "Honshu",
        "O primeiro mapeamento funciona");
```

O mapeamento, é claro, existe para os dois primeiros ninjas, mas não existe para o terceiro ninja, porque não usamos o terceiro ninja como um argumento para o definir método. O estado atual do mapa é mostrado na figura 9.14.

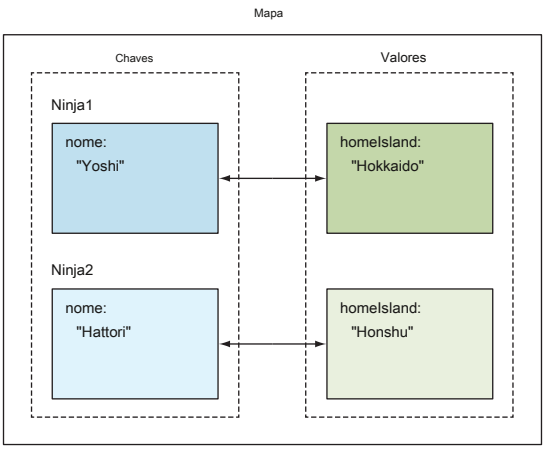


Figura 9.14 Um mapa é uma coleção de pares de valores-chave, onde uma chave pode ser qualquer coisa - até mesmo outro objeto.

Além de obter e definir métodos, cada mapa também tem um Tamanho propriedade e tem e excluir métodos. O Tamanho propriedade nos diz quantos mapeamentos criamos. Nesse caso, criamos apenas dois mapeamentos.

O tem método, por outro lado, nos notifica se um mapeamento para uma determinada chave já existe:

```
ninjaIslandMap.has (ninja1); //verdade
ninjaIslandMap.has (ninja3); //falso
```

O excluir método nos permite remover itens de nosso mapa:

```
ninjaIslandMap.delete (ninja1);
```

Um dos conceitos fundamentais ao lidar com mapas é determinar quando duas chaves de mapa são iguais. Vamos explorar esse conceito.

IGUALDADE CHAVE

Se você vem de uma formação um pouco mais tradicional, como C #, Java ou Python, pode se surpreender com o próximo exemplo.

Listagem 9.17 Igualdade de chave em mapas

```
mapa const = novo Mapa ();
const currentLocation = location.href;

const firstLink = novo URL (currentLocation); const secondLink = novo URL
(currentLocation);

map.set (firstLink, {descrição: "firstLink"}); map.set (secondLink, {descrição: "secondLink"});
```

Usa a propriedade interna location.href para obter o URL da página atual

Cria dois links para a página atual

Adiciona um mapeamento para ambos os links

```
assert (map.get (firstLink) .description === "firstLink",
            "Mapeamento do primeiro link");
assert (map.get (secondLink) .description === "secondLink",
             "Segundo mapeamento de link");
assert (map.size === 2, "Existem dois mapeamentos");
```

Cada link obtém seu próprio mapeamento, embora eles apontam para a mesma página.

Na Listagem 9.17, usamos o integrado location.href propriedade para obter o URL da página atual. Em seguida, usando o construtor de URL integrado, criamos dois novos objetos de URL que vinculam à página atual. Em seguida, associamos um objeto de descrição a cada link. Por fim, verificamos se os mapeamentos corretos foram criados, conforme mostrado na figura 9.15.

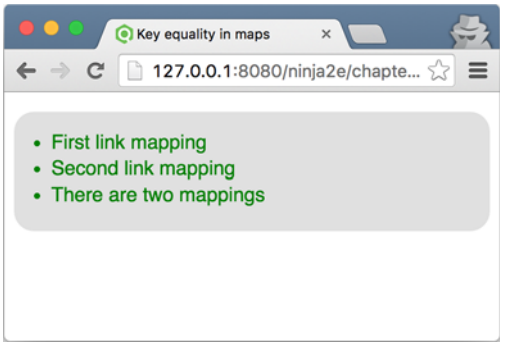


Figura 9.15 Se executarmos o código da Listagem 9.17, podemos ver que a igualdade de chave nos mapas é baseada na igualdade do objeto.

Pessoas que trabalharam principalmente em JavaScript podem não achar este resultado inesperado: Temos dois objetos diferentes para os quais criamos dois mapeamentos ent. Mas observe que os dois objetos URL, mesmo que sejam objetos separados, ainda apontam para o mesmo local do URL: o local da página atual. Poderíamos argumentar que, ao criar mapeamentos, esses dois objetos devem ser considerados iguais. Mas em JavaScript, não podemos sobrecarregar o operador de igualdade, e os dois objetos, embora tenham o mesmo conteúdo, são sempre considerados diferentes. Este não é o caso com outras linguagens, como Java e C #, então tome cuidado!

9.2.3 *Iterando em mapas*

Até agora, você viu algumas das vantagens dos mapas: Você pode ter certeza de que eles contêm apenas os itens que você colocou neles e pode usar qualquer coisa como chave. Mas tem mais!

Como os mapas são coleções, não há nada que nos impeça de iterar sobre eles com para de rotações. (Lembre-se, usamos o para de loop para iterar sobre os valores criados pelos geradores no capítulo 6.) Você também tem a garantia de que esses valores serão visitados na ordem em que foram inseridos (algo em que não podemos confiar ao iterar objetos usando o para ... em ciclo). Vejamos o seguinte exemplo.

Listagem 9.18 Iterando sobre mapas

```
diretório const = novo Map ();

directory.set ("Yoshi", "+81 26 6462"); directory.set ("Kuma", "+81 52 2378
6462"); directory.set ("Hiro", "+81 76 277 46");
```

Cria um novo mapa, assim como fizemos até agora

Cria um diretório ninja que armazena o número de telefone de cada ninja

```
para (deixar item do diretório) {
    assert (item [0]! == null, "Chave:" + item [0]); assert (item [1]! == null, "Value:" + item
    [1]);
}
```

Itera sobre cada item em um dicionário usando o loop for ... of. Cada item é uma matriz de dois itens: uma chave e um valor.

```
para (deixe a chave do diretório.keys ()) {
    assert (key! == null, "Key:" + key); assert (directory.get (key)! = null,

                "Valor:" + diretório.get (chave));
}
```

Também podemos iterar sobre as chaves usando o método de chaves integradas ...

```
para (valor var de diretório.values   ()) {
    assert (valor! == null, "Valor:" + valor);
}
```

... E sobre os valores usando o método de valores embutidos.

Como mostra a lista anterior, depois de criar um mapeamento, podemos facilmente iterar sobre ele usando o para de ciclo:

```
para (var item do diretório) {
    assert (item [0]! == null, "Chave:" + item [0]); assert (item [1]! == null, "Value:" + item
    [1]);
}
```

Em cada iteração, isso fornece uma matriz de dois itens, onde o primeiro item é uma chave e o segundo item é o valor de um item de nosso mapa de diretório. Também podemos usar o chaves e valores métodos para iterar sobre, bem, chaves e valores contidos em um mapa.

Agora que vimos os mapas, vamos visitar outro novato em JavaScript: *conjuntos,* que são coleções de itens exclusivos.

9,3 Jogos

Em muitos problemas do mundo real, temos que lidar com coleções de *distinto* itens (o que significa que cada item não pode aparecer mais de uma vez) chamados *conjuntos*. Até o ES6, isso era algo que você mesmo precisava implementar imitando conjuntos com objetos padrão. Para um exemplo bruto, consulte a próxima lista.

Listagem 9.19 Conjuntos de imitação com objetos

```
function Set () {
    this.data = {};
    this.length = 0;
}
```

Usa um objeto para armazenar itens

```
Set.prototype.has = function (item) {
    return typeof this.data [item]! == " Indefinido";
};
```

Verifica se o o item já está armazenado

```
Set.prototype.add = function (item) {
    if (! this.has (item)) {
        this.data [item] = verdadeiro;
        this.length ++;
    }
};
```

Adiciona um item apenas se ainda não estiver contido no conjunto

```
Set.prototype.remove = function (item) {
    if (this.has (item)) {
        delete this.data [item];
        this.length--;
    }
};
```

Remove um item se já estiver contido no set

```
const ninjas = new Set (); ninjas.add ("Hattori");

ninjas.add ("Hattori");
```

Tenta adicionar Hattori duas vezes

```
assert (ninjas.has ("Hattori") && ninjas.length === 1,
            "Nosso conjunto contém apenas um Hattori");
```

Verifica que Hattori foi adicionado apenas uma vez

```
ninjas.remove ("Hattori");
assert (! ninjas.has ("Hattori") && ninjas.length === 0,
            "Nosso conjunto agora está vazio");
```

Remove Hattori e verifica se ele foi removido do conjunto

A Listagem 9.19 mostra um exemplo simples de como os conjuntos podem ser imitados com objetos. Usamos um objeto de armazenamento de dados, dados, para acompanhar nossos itens definidos, e expomos três métodos: tem, que verifica se um item já está contido no conjunto; adicionar, que adiciona um item apenas se o mesmo item ainda não estiver contido no conjunto; e retirar,
que remove um item já contido do conjunto.

Mas este é um pobre doppelganger. Porque com mapas, você não pode realmente armazenar objetos - apenas strings e números - e sempre há o risco de acessar objetos de protótipo. Por essas razões, o comitê ECMAScript decidiu introduzir um tipo completamente novo de coleção: *conjuntos.*

NOTA Os conjuntos são parte do padrão ES6. Para compatibilidade do navegador atual, consulte http://mng.bz/QRTS .

9.3.1 Criando nosso primeiro conjunto

A pedra angular da criação de conjuntos é a função construtora recém-introduzida, convenientemente chamada Definir. Vejamos um exemplo.

Listagem 9.20 Criando um conjunto

O construtor Set pode ter uma matriz de itens com os quais o conjunto é inicializado.

```
const ninjas = novo Conjunto (["Kuma", "Hattori", "Yagyu", "Hattori"]);

assert (ninjas.has ("Hattori"), "Hattori está em nosso conjunto");
assert (ninjas.size === 3, "Existem apenas três ninjas em nosso conjunto!");
```

Descarta todos os itens duplicados

```
afirmar (! ninjas.has ("Yoshi"), "Yoshi não está, ainda .."); ninjas.add ("Yoshi");

assert (ninjas.has ("Yoshi"), "Yoshi é adicionado");
assert (ninjas.size === 4, "Existem quatro ninjas em nosso conjunto!");

assert (ninjas.has ("Kuma"), "Kuma já foi adicionado"); ninjas.add ("Kuma");

assert (ninjas.size === 4, "Adicionar Kuma novamente não tem efeito");

para (vamos ninja de ninjas) {
    afirmar (ninja! == nulo, ninja);
}
```

Podemos adicionar novos itens que não são já contido no conjunto.

Adicionando existente itens não tem efeito.

Itera através do definido com um for ... de loop

Aqui, usamos o integrado Definir construtor para criar um novo ninjas conjunto que conterá ninjas distintos. Se não passarmos nenhum argumento, um conjunto vazio é criado. Também podemos passar uma matriz, como esta, que preenche previamente o conjunto:

```
novo conjunto (["Kuma", "Hattori", "Yagyu", "Hattori"]);
```

Como já mencionamos, conjuntos são coleções de itens únicos e seu objetivo principal é nos impedir de armazenar várias ocorrências do mesmo objeto. Neste caso, isso significa " Hattori ", que tentamos adicionar duas vezes, é adicionado apenas uma vez.

Vários métodos são acessíveis a partir de cada conjunto. Por exemplo, o tem método verifica se um item está contido no conjunto:

```
ninjas.has ("Hattori")
```

e a adicionar método é usado para adicionar itens únicos ao conjunto:

```
ninjas.add ("Yoshi");
```

Se você está curioso sobre como muitos itens estão em um conjunto, você sempre pode usar o Tamanho propriedade.

Semelhante a mapas e matrizes, conjuntos são coleções, então podemos iterar sobre eles com um para de ciclo. Como você pode ver na figura 9.16, os itens são sempre iterados na ordem em que foram inseridos.

Agora que passamos pelos fundamentos dos conjuntos, vamos visitar algumas operações comuns nos conjuntos: uniões, interseções e diferenças.

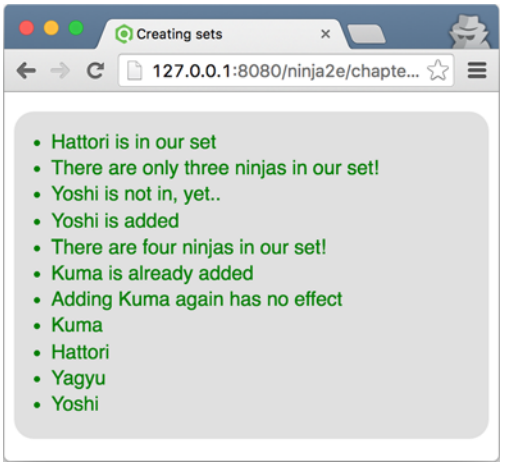


Figura 9.16 Executar o código da listagem 9.20 mostra que os itens em um conjunto são iterados na ordem em que foram inseridos.

9.3.2 União de conjuntos

Uma união de dois conjuntos, UMA e B, cria um novo conjunto que contém todos os elementos de ambos UMA e B. Naturalmente, cada item não pode ocorrer mais de uma vez no novo conjunto.

Listagem 9.21 Usando conjuntos para realizar uma união de coleções

Cria uma série de ninjas e samurais. Observe que Hattori é um ninja e um samurai.

```
const ninjas = ["Kuma", "Hattori", "Yagyu"]; const samurai = ["Hattori", "Oda",
"Tomoe"];

guerreiros const = novo conjunto ([... ninjas, ... samurai]);

afirmar (warriors.has ("Kuma"), "Kuma está aqui"); afirme (warriors.has ("Hattori"), "E
Hattori");
assert (warriors.has ("Yagyu"), "And Yagyu");
afirmar (warriors.has ("Oda"), "E Oda");
assert (warriors.has ("Tomoe"), "Tomoe, por último, mas não menos importante");

afirme (warriors.size === 5, "Há 5 guerreiros no total");
```

Cria um novo conjunto de guerreiros espalhando ninjas e samurai

Todos os ninjas e samurais são incluído em novo conjunto de guerreiros.

Não há duplicatas no novo conjunto. Mesmo que Hattori esteja nos conjuntos de ninjas e samurai, ele é incluído apenas uma vez.

Primeiro criamos uma matriz de ninjas e uma série de samurai. Observe que Hattori leva uma vida agitada: samurai de dia, ninja à noite. Agora imagine que precisamos criar um grupo de pessoas que possamos chamar às armas se um daimyo vizinho decidir que sua província está um pouco apertada. Criamos um novo conjunto, guerreiros, isso inclui todos os ninjas e todos os samurais. Hattori está em ambas as coleções, mas queremos incluí-lo apenas uma vez - não é como se dois Hattoris respondessem ao nosso chamado.

Nesse caso, um conjunto é perfeito! Não precisamos controlar manualmente se um item já foi incluído: o conjunto cuida disso sozinho, automaticamente. Ao criar este novo conjunto, usamos o operador spread [... ninjas, ... samurai] (lembre-se do capítulo 3) para criar um novo array que contenha todos os ninjas e todos os samurais. Caso você esteja se perguntando, o Hattori está presente duas vezes nesta nova matriz. Mas quando finalmente passamos essa matriz para o Definir construtor, Hattori é incluído apenas uma vez, conforme mostrado na figura 9.17.

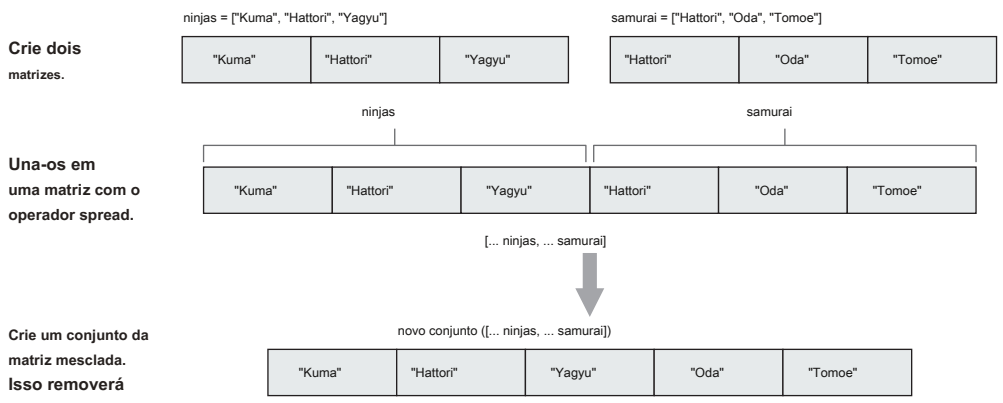


Figura 9.12 Uma união de dois conjuntos mantém os itens de ambas as coleções (sem duplicatas).

9.3.3 Interseção de conjuntos

O *interseção* de dois conjuntos, UMA e B, cria um conjunto que contém elementos de UMA que também estão em B. Por exemplo, podemos encontrar ninjas que também são samurais, como mostrado a seguir.

Listagem 9.22 Intersecção de conjuntos

```
const ninjas = novo conjunto (["Kuma", "Hattori", "Yagyu"]); Usa o operador spread para virar
const samurai = new Set (["Hattori", "Oda", "Tomoe"]); nosso conjunto em uma matriz para que possamos
                                                       use o método de filtro da matriz para
                                                       manter apenas ninjas que são
const ninjaSamurais = novo conjunto (                  contido no conjunto de samurai
    [... ninjas] .filter (ninja => samurai.has (ninja))
);

assert (ninjaSamurais.size === 1, "Há apenas um samurai ninja"); assert (ninjaSamurais.has ("Hattori"), "Hattori é o nome dele");
```

A ideia por trás da listagem 9.22 é criar um novo conjunto que contenha apenas ninjas que também são samurais. Fazemos isso aproveitando as vantagens do array filtro método, que, como você deve se lembrar, cria uma nova matriz que contém apenas os itens que correspondem a um determinado critério. Nesse caso, o critério é que o ninja também seja um samurai (está contido no conjunto de samurai). Porque o filtro método só pode ser usado em matrizes, temos que transformar o ninjas definido em uma matriz usando o operador spread:

```
[... ninjas]
```

Por fim, verificamos que encontramos apenas um ninja que também é samurai: o pau para toda obra, Hattori.

9.3.4 Diferença de conjuntos

A diferença de dois conjuntos, UMA e B, contém todos os elementos que estão no conjunto UMA mas são *não* em conjunto B. Como você pode imaginar, isso é semelhante à interseção de conjuntos, com uma diferença pequena, mas significativa. Na próxima lista, queremos encontrar apenas ninjas verdadeiros (não aqueles que também fazem luar como samurais).

Listagem 9.23 Diferença de conjuntos

```
const ninjas = novo conjunto (["Kuma", "Hattori", "Yagyu"]); const samurai = new Set (["Hattori", "Oda",
"Tomoe"]);                                                  Com diferença definida,
                                                            nos preocupamos apenas
const pureNinjas = new Set (                                com ninjas que não são
    [... ninjas] .filter (ninja =>! samurai.has (ninja))    samurais!
);

assert (pureNinjas.size === 2, "Há apenas um samurai ninja"); assert (pureNinjas.has ("Kuma"), "Kuma é um verdadeiro
ninja"); assert (pureNinjas.has ("Yagyu"), "Yagyu é um verdadeiro ninja");
```

A única mudança é especificar que nos preocupamos apenas com os ninjas que são *não* também samurai, colocando um ponto de exclamação (!) antes do samurai.has (ninja) expressão.

9,4 Resumo

- Arrays são um tipo especial de objeto com um comprimento propriedade e Array.prototype como seu protótipo.
- Podemos criar novos arrays usando a notação literal de array ([]) ou chamando o Variedade construtor.
- Podemos modificar o conteúdo de uma matriz usando vários métodos acessíveis a partir de objetos de matriz:

  - O embutido Empurre e pop métodos adicionam itens e removem itens do final da matriz.
  - O embutido mudança e não mudar métodos adicionam itens e removem itens do início da matriz.
  - O embutido emenda O método pode ser usado para remover itens e adicionar itens a posições arbitrárias de array.
- Todos os arrays têm acesso a vários métodos úteis:

  - O mapa método cria uma nova matriz com os resultados de chamar um retorno de chamada em cada elemento.
  - O cada e alguns métodos determinam se todos ou alguns itens da matriz satisfazem um determinado critério.
  - O achar e filtro métodos encontram itens de array que satisfazem uma certa condição.
  - O organizar método classifica uma matriz. O reduzir método agrega todos os itens em uma matriz em
  - um único valor.
- Você pode reutilizar os métodos de matriz integrados ao implementar seus próprios objetos, definindo explicitamente o contexto de chamada do método com o chamar ou Aplique método. Mapas e dicionários são
- objetos que contêm mapeamentos entre uma chave e um valor.
- Objetos em JavaScript são mapas ruins porque você só pode usar valores de string como chaves e porque sempre há o risco de acessar propriedades de protótipo. Em vez disso, use o novo Mapa coleção.
- Os mapas são coleções e podem ser iterados usando o para de ciclo. Conjuntos são coleções
- de itens exclusivos.

9,5 Exercícios

1 Qual será o conteúdo do samurai array, depois de executar o seguinte código?

```
const samurai = ["Oda", "Tomoe"]; samurai [3] = "Hattori";
```

2 Qual será o conteúdo do ninjas array, depois de executar o seguinte código?

```
const ninjas = [];

ninjas.push ("Yoshi");
```

```
ninjas.unshift ("Hattori");

ninjas.length = 3;

ninjas.pop ();
```

3 Qual será o conteúdo do samurai array, depois de executar o seguinte código?

```
const samurai = [];

samurai.push ("Oda");
samurai.unshift ("Tomoe");
samurai.splice (1, 0, "Hattori", "Takeda"); samurai.pop ();
```

4 O que será armazenado nas variáveis primeiro segundo, e terceiro, depois de executar o código abaixo?

```
const ninjas = [{nome: "Yoshi", idade: 18},
                {nome: "Hattori", idade: 19},
                {nome: "Yagyu", idade: 20}];

const primeiro = pessoas.map (ninja => ninja.age); const segundo = primeiro.filtro (idade =>
idade% 2 == 0);
const third = first.reduce ((agregado, item) => agregado + item, 0);
```

5 O que será armazenado nas variáveis primeiro e segundo, depois de executar o seguinte código?

```
const ninjas = [{nome: "Yoshi", idade: 18},
                {nome: "Hattor", idade: 19}, {nome: "Yagyu", idade:
                20}];

const primeiro = ninjas.some (ninja => ninja.age% 2 == 0); segundo const = ninjas.every (ninja => ninja.age%
2 == 0);
```

6 Qual das afirmações a seguir será aprovada?

```
const samuraiClanMap = new Map ();

const samurai1 = {nome: "Toyotomi"}; const samurai2 = {nome:
"Takeda"}; const samurai3 = {nome: "Akiyama"};


const oda = {clã: "Oda"};
const tokugawa = {clã: "Tokugawa"}; const takeda = {clã: "Takeda"};


samuraiClanMap.set (samurai1, oda);
samuraiClanMap.set (samurai2, tokugawa);
samuraiClanMap.set (samurai2, takeda);
```

assert (samuraiClanMap.size === 3, "Existem três mapeamentos"); assert (samuraiClanMap.has (samurai1), "O primeiro samurai tem um mapeamento"); assert (samuraiClanMap.has (samurai3), "O terceiro samurai tem um mapeamento");

7 Qual das afirmações a seguir será aprovada?

const samurai = novo Conjunto ("Toyotomi", "Takeda", "Akiyama", "Akiyama"); assert (samurai.size === 4, "Existem quatro samurais no conjunto");

samurai.add ("Akiyama");
assert (samurai.size === 5, "Existem cinco samurais no conjunto");

assert (samurai.has ("Toyotomi", "Toyotomi está dentro!"); assert (samurai.has ("Hattori", "Hattori está dentro!");

10 Wrangling expressões regulares

Este capítulo cobre

- Uma atualização sobre expressões regulares
- Compilando expressões regulares
- Captura com expressões regulares
- Trabalhar com expressões idiomáticas frequentemente encontradas

As expressões regulares são uma necessidade do desenvolvimento moderno. Lá, nós dissemos isso. Embora muitos desenvolvedores da Web possam passar a vida ignorando as expressões regulares, alguns problemas que precisam ser resolvidos no código JavaScript não podem ser resolvidos com elegância sem as expressões regulares.

Claro, pode haver outras maneiras de resolver os mesmos problemas. Mas frequentemente, algo que pode ocupar uma meia tela de código pode ser destilado em uma única instrução com o uso adequado de expressões regulares. Todos os ninjas do JavaScript precisam de expressões regulares como parte essencial de seus kits de ferramentas.

As expressões regulares trivializam o processo de separar strings e procurar informações. Onde quer que você olhe nas principais bibliotecas JavaScript, você verá o uso predominante de expressões regulares para várias tarefas pontuais:

- Manipulando strings de nós HTML
- Localizando seletores parciais em uma expressão de seletor CSS
- Determinar se um elemento tem um nome de classe específico

- Validação de entrada
- E mais

Vamos começar vendo um exemplo.

GORJETA Tornar-se fluente em expressões regulares exige muita prática. Você pode encontrar um site como JS Bin (http://jsbin.com) útil para brincar com exemplos. Alguns sites são dedicados a testes de expressão regular, como a página de teste de expressão regular para JavaScript (www.regexplanet.com/advanced/javascript / index.html) e regex101 (www.regex101.com/#javascript) . regex101 é um site especialmente útil para iniciantes, porque também gera automaticamente explicações para a expressão regular de destino.

Você sabe?

Quando você prefere usar um literal RegExp em vez de um Objeto RegExp?

O que é correspondência aderente e como você a habilita?

Como a correspondência difere ao usar um global versus um expressão regular não global?

10.1 Por que as expressões regulares são demais

Digamos que desejamos validar que uma string, talvez inserida em um formulário por um usuário de site, segue o formato de um código postal dos EUA de nove dígitos. Todos nós sabemos que o Serviço Postal dos EUA tem pouco senso de humor e insiste que um código postal (também conhecido como CEP) siga este formato específico:

```
99999-9999
```

Cada 9 representa um dígito decimal, e o formato é 5 dígitos decimais, seguidos por um hífen, seguido por 4 dígitos decimais. Se você usar qualquer outro formato, seu pacote ou carta será desviado para o buraco negro do departamento de classificação manual e boa sorte ao prever quanto tempo levará para emergir novamente.

Vamos criar uma função que, dada uma string, verifique se o US Postal Service ficará feliz. Poderíamos recorrer à comparação de cada personagem, mas somos um ninja e isso é uma solução muito deselegante, resultando em muitas repetições desnecessárias. Em vez disso, considere a seguinte solução.

Listagem 10.1 Testando um padrão específico em uma string

```
function isThisAZipCode (candidate) {
    if (typeof candidate! == "string" ||
            candidato.comprimento! = 10) retorna falso; para (seja n = 0; n <comprimento
    candidato; n ++) {
        seja c = candidato [n];
```

Curto-circuitos obviamente candidatos falsos

```
        switch (n) {
            caso 0: caso 1: caso 2: caso 3: caso 4: caso 6: caso 7: caso 8: caso 9:

                if (c <'0' || c> '9') retorna falso; intervalo;

            caso 5:
                if (c! = '-') retorna falso; intervalo;

        }
    }
    return true;
}
```

Executa testes com base no índice de personagem

Se tudo der certo, foram bons!

Este código tira vantagem do fato de que temos apenas duas verificações a fazer, dependendo da posição do caractere dentro da string. Ainda precisamos realizar até nove comparações em tempo de execução, mas temos que escrever cada comparação apenas uma vez.

Mesmo assim, alguém consideraria esta solução *elegante?* É mais elegante do que a abordagem de força bruta e não literária, mas ainda parece uma quantidade enorme de código para uma verificação tão simples. Agora considere esta abordagem:

```
function isThisAZipCode (candidate) {
    return /^\d{5}-\d{4}$/.test(candidate);
}
```

Exceto por alguma sintaxe esotérica no corpo da função, ela é muito mais sucinta e elegante, não? Esse é o poder das expressões regulares e é apenas a ponta do iceberg. Não se preocupe se essa sintaxe parecer que a iguana de estimação de alguém atravessou o teclado; estamos prestes a recapitular as expressões regulares antes de você aprender a usá-las de maneira ninja em suas páginas.

10.2 Uma atualização de expressão regular

Por mais que gostaríamos, não podemos oferecer um tutorial exaustivo sobre expressões regulares no espaço que temos. Afinal, livros inteiros foram dedicados a expressões regulares. Mas faremos o nosso melhor para atingir todos os pontos importantes.

Para mais detalhes do que podemos oferecer neste capítulo, os livros *Dominando Expressões Regulares* por Jeffrey EF Friedl, *Apresentando Expressões Regulares* por Michael Fitzgerald, e *Livro de receitas de expressões regulares* por Jan Goyvaerts e Steven Levithan, todos da O'Reilly, são escolhas populares.

Vamos cavar.

10.2.1 Expressões regulares explicadas

O termo *expressão regular* deriva da matemática de meados do século, quando um matemático chamado Stephen Kleene descreveu modelos de autômatos computacionais como "conjuntos regulares". Mas isso não nos ajudará a entender nada sobre expressões regulares, então vamos simplificar as coisas e dizer que uma expressão regular é uma forma de expressar uma *padronizar* para combinar sequências de texto. A própria expressão consiste em termos e

operadores que nos permitem definir esses padrões. Veremos em que consistem esses termos e operadores em breve.

Em JavaScript, como na maioria dos outros tipos de objeto, temos duas maneiras de criar uma expressão regular:

- Por meio de uma expressão regular literal
- Construindo uma instância de um RegExp objeto

Por exemplo, se quisermos criar uma expressão regular mundana (ou *regex,* para breve) que corresponde à string teste exatamente, poderíamos fazer isso com um literal regex:

```
padrão const = / teste /;
```

Isso pode parecer estranho, mas os literais regex são delimitados por barras da mesma forma que literais de string são delimitados por aspas.

Alternativamente, poderíamos construir um RegExp instância, passando o regex como uma string:

```
padrão const = novo RegExp ("teste");
```

Ambos os formatos resultam na mesma regex sendo criada na variável padronizar.

GORJETA A sintaxe literal é preferida quando a regex é conhecida no tempo de desenvolvimento e a abordagem do construtor é usada quando a regex é construída em tempo de execução, construindo-a dinamicamente em uma string.

Um dos motivos pelos quais a sintaxe literal é preferível a expressar regexes em uma string é que (como você verá em breve) o caractere de barra invertida desempenha um papel importante nas expressões regulares. Mas o caractere barra invertida é *Além disso* o caractere de escape para literais de string, portanto, para expressar uma barra invertida dentro de um literal de string, precisamos usar uma barra invertida dupla (\\). Isso pode tornar as expressões regulares, que já possuem uma sintaxe enigmática, ainda mais estranhas quando expressas em strings.

Além da própria expressão, cinco sinalizadores podem ser associados a um regex:

- eu - Torna o regex insensível a maiúsculas e minúsculas, então / teste / i corresponde não só *teste,* mas também *Teste, teste, teste,* e assim por diante.
- g - Corresponde a todas as instâncias do padrão, em oposição ao padrão de *local,* que corresponde apenas à primeira ocorrência. Mais sobre isso mais tarde.
- m —Permite correspondências em várias linhas, como pode ser obtido a partir do valor de um área de texto elemento.
- y —Ativa a correspondência fixa. Uma expressão regular executa a correspondência fixa na string de destino, tentando corresponder a partir da última posição de correspondência.
- você —Ativa o uso de pontos de escape Unicode (\ você{...}).

Esses sinalizadores são anexados ao final do literal (por exemplo, / test / ig) ou passado em uma string como o segundo parâmetro para o RegExp construtor (novo RegExp ("teste", "ig")).

Combinando a string exata *teste (* mesmo que não faça distinção entre maiúsculas e minúsculas) não é interessante - afinal, podemos fazer essa verificação específica com uma comparação de string simples. Portanto, vamos dar uma olhada nos termos e operadores que fornecem às expressões regulares seu imenso poder para corresponder a padrões mais atraentes.

10.2.2 Termos e operadores

As expressões regulares, como a maioria das outras expressões com as quais estamos familiarizados, são compostas de termos e operadores que qualificam esses termos. Nas seções a seguir, você verá como esses termos e operadores podem ser usados para expressar padrões.

CORRESPONDÊNCIA EXATA

Qualquer caractere que não seja um caractere ou operador especial (que apresentaremos à medida que prosseguirmos) deve aparecer literalmente na expressão. Por exemplo, em nosso / teste/ regex, quatro termos representam caracteres que devem aparecer literalmente em uma string para corresponder ao padrão expresso.

Colocar esses caracteres um após o outro denota implicitamente uma operação que significa *seguido pela.* Então / teste/ meios *t* seguido pela *e* seguido pela *s* seguido pela *t.*

CORRESPONDÊNCIA DE UMA CLASSE DE PERSONAGENS

Muitas vezes, não queremos corresponder a um caractere literal específico, mas a um caractere de um conjunto finito de caracteres. Podemos especificar isso com o operador set (também chamado de *classe de personagem* operador) colocando o conjunto de caracteres que queremos corresponder entre colchetes: [abc].

O exemplo anterior significa que queremos corresponder a qualquer um dos caracteres a, b, ou c. Observe que, embora essa expressão tenha cinco caracteres (três letras e dois colchetes), ela corresponde a apenas um único caractere na string candidata.

Outras vezes, queremos combinar qualquer coisa *mas* um conjunto finito de caracteres. Podemos especificar isso colocando um caractere circunflexo (^)ˆ logo após o colchete de abertura do operador de conjunto:

[^ abc]

Isso muda o significado de qualquer caractere *mas* a, b, ou c.

Há mais uma variação inestimável para a operação de conjunto: a capacidade de especificar uma faixa de valores. Por exemplo, se quisermos combinar qualquer um dos caracteres minúsculos entre *uma* e *m,* poderíamos escrever [abcdefghijklm]. Mas podemos expressar isso muito mais sucintamente da seguinte forma:

[sou]

O traço indica que todos os personagens de uma Através dos m inclusivos (e lexicograficamente) estão incluídos no conjunto.

ESCAPANDO

Nem todos os caracteres representam seu equivalente literal. Certamente, todos os caracteres alfabéticos e decimais representam a si mesmos, mas, como você verá, os caracteres especiais como $ e o ponto (.) Representam correspondências com algo diferente de si mesmos,

ou operadores que qualificam o termo anterior. Na verdade, você já viu como o
[,], -, e ^ caracteres são usados para representar algo diferente de seus eus literais.

Como especificamos que desejamos corresponder a um literal [ou $ ou ^ otherou outro caractere especial? Em uma regex, o caractere de barra invertida escapa de qualquer caractere que o segue, tornando-o um termo de correspondência literal. Portanto, \ [especifica uma correspondência literal com o caractere [, em vez da abertura de uma expressão de classe de caractere. Uma barra invertida dupla (\\) corresponde a uma única barra invertida.

COMEÇA E TERMINA

Freqüentemente, podemos querer garantir que um padrão corresponda ao início de uma string ou talvez ao final de uma string. O caractere circunflexo, quando usado como o primeiro caractere da regex, ancora a correspondência no início da string, de modo que / ^ teste/
corresponde apenas se a substring teste aparece no início da string que está sendo correspondida. (Observe que isso é uma sobrecarga do caractere ^, porque também é usado para negar um conjunto de classes de caracteres.)

Da mesma forma, o cifrão ($) significa que o padrão deve aparecer no final da string:

```
/ test $ /
```

O uso de ^ e $ indica que o padrão especificado deve abranger toda a string candidata:

```
/ ^ teste $ /
```

OCORRÊNCIAS REPETIDAS

Se quisermos combinar uma série de quatro uma personagens, podemos expressar isso com / aaaa /,
mas e se quisermos combinar *nenhum* número do mesmo personagem? As expressões regulares nos permitem especificar várias opções de repetição:

- Para especificar que um caractere é opcional (ele pode aparecer uma vez ou não), siga-o com?. Por exemplo, / teste/ combina com ambos *teste* e *Husa.*
- Para especificar que um caractere deve aparecer uma ou várias vezes, use +, como em / t + est /, que combina *test, ttest,* e *tttest,* mas não *Husa.*
- Para especificar que o personagem aparece *Zero um,* ou *vários* vezes, use *, como em / teste/, que combina *test, ttest, tttest e est.*
- Para especificar um número fixo de repetições, indique o número de repetições permitidas entre colchetes. Por exemplo, / um {4} / indica uma partida em quatro consecutivas uma personagens.
- Para especificar um intervalo para a contagem de repetição, indique o intervalo com um separador de vírgula. Por exemplo, / a {4,10} / corresponde a qualquer sequência de 4 a 10 consecutivos uma personagens.
- Para especificar um intervalo aberto, omita o segundo valor do intervalo (mas deixe a vírgula). O regex / a {4,} / corresponde a qualquer sequência de quatro ou mais consecutivas uma personagens.

Qualquer um desses operadores de repetição pode ser *ambicioso* ou *não melancólico*. Por padrão, eles são gananciosos: eles vão consumir todos os personagens possíveis que compõem uma combinação. Anotando o operador com um? personagem (uma sobrecarga do operador?), como em a + ?, torna a operação não séria: vai consumir *só* caracteres suficientes para fazer uma correspondência.

Por exemplo, se estivermos combinando com a string aaa, a expressão regular / a + / corresponderia a todos os três uma caracteres, enquanto a expressão não agitada / a +? / corresponderia a apenas um uma personagem, porque um único uma caráter é tudo o que é necessário para satisfazer o a + prazo.

CLASSES DE CARÁTER PREDEFINIDAS

Alguns caracteres que podemos desejar corresponder são impossíveis de especificar com caracteres literais (por exemplo, caracteres de controle, como um retorno de carro). Além disso, muitas vezes podemos querer combinar classes de caracteres, como um conjunto de dígitos decimais ou um conjunto de caracteres de espaço em branco. A sintaxe da expressão regular fornece termos predefinidos que representam esses caracteres ou classes comumente usadas para que possamos usar a correspondência de caracteres de controle em nossas expressões regulares e não precisemos recorrer ao operador de classe de caracteres para conjuntos de caracteres comumente usados.

A Tabela 10.1 lista esses termos e o caractere ou conjunto que eles representam. Esses conjuntos predefinidos ajudam a evitar que nossas expressões regulares pareçam excessivamente enigmáticas.

Tabela 10.1 Classes de personagens e termos de personagens predefinidos

Termo predefinido	Partidas
\ t	Aba horizontal
\ b	Backspace
\ v	Aba vertical
\ f	Feed de formulário
\ r	Retorno de carruagem
\ n	Nova linha
\ cA: \ cZ	Personagens de controle
\ u0000: \ uFFFF	Hexadecimal Unicode
\ x00: \ xFF	Hexadecimal ASCII
.	Qualquer caractere, exceto para caracteres de espaço em branco (\ s)
\ d	Qualquer dígito decimal; equivalente a [0-9]
\ D	Qualquer caractere, exceto um dígito decimal; equivalente a [^ 0-9]
\C	Qualquer caractere alfanumérico incluindo sublinhado; equivalente a [A-Za-z0-9_]
\C	Qualquer caractere, exceto caracteres alfanuméricos e sublinhados; equivalente a [^ A-Za-z0-9_]

Tabela 10.1 Classes de caracteres e termos de caracteres predefinidos (***contínuo)***

Termo predefinido	Partidas
\ s	Qualquer caractere de espaço em branco (espaço, tabulação, alimentação de formulário e assim por diante)
\ S	Qualquer caractere, exceto um caractere de espaço em branco
\ b	Um limite de palavra
\ B	Sem limite de palavra (dentro de uma palavra)

AGRUPAMENTO

Até agora, você viu que os operadores (como + e *) afetam apenas o termo anterior. Se quisermos aplicar o operador a um grupo de termos, podemos usar parênteses para grupos, assim como em uma expressão matemática. Por exemplo, /(ab) + / corresponde a uma ou mais ocorrências consecutivas da substring ab.

Quando uma parte de uma regex é agrupada com parênteses, ela tem uma dupla função, criando também o que é conhecido como *capturar.* Há muitas capturas e as discutiremos com mais detalhes na seção 10.4.

ALTERNAÇÃO (OU)

As alternativas podem ser expressas usando o caractere barra vertical (|). Por exemplo: / a | b /
combina com o uma ou b personagem, e / (ab) + | (cd) + / corresponde a uma ou mais ocorrências de qualquer ab ou CD.

BACKREFERENCES

Os termos mais complexos que podemos expressar em expressões regulares são referências anteriores a *captura* definido no regex. Abordamos as capturas detalhadamente na seção 10.4, mas por enquanto apenas pense nelas como as partes de uma string candidata que são correspondidas com êxito aos termos da expressão regular. A notação para tal termo é a barra invertida seguida pelo número da captura a ser referenciada, começando com 1, como \ 1, \ 2, e assim por diante.

Um exemplo é / ^ ([dtn]) a \ 1 /, que corresponde a uma string que começa com qualquer um dos d, t, ou n caracteres, seguidos por um uma, seguido por qualquer caractere que corresponda à primeira captura. Este último ponto é importante! Este não é o mesmo que / [dtn] a [dtn] /. O personagem seguindo o uma não pode ser nenhum de d, ou t, ou n, mas deve ser aquele que acionar a correspondência para o primeiro caractere. Como tal, qual personagem o \ 1 irá corresponder não pode ser conhecido até a hora da avaliação.

Um bom exemplo de onde isso pode ser útil é na correspondência de elementos de marcação do tipo XML. Considere o seguinte regex:

```
/<(\w+)>(.+)<\/\1>/
```

Isso nos permite combinar elementos simples, como **<** forte**>** tanto faz **</** forte**>**. Sem a capacidade de especificar uma referência anterior, isso não seria possível, porque não teríamos como saber com antecedência qual tag de fechamento corresponderia à tag de abertura.

GORJETA Foi uma espécie de curso intensivo sobre expressões regulares. Se eles ainda estão fazendo você puxar seus cabelos e você se encontrar atolado no material a seguir, recomendamos o uso de um dos recursos mencionados anteriormente neste capítulo.

Agora que você conhece as expressões regulares, está pronto para ver como usá-las sabiamente em seu código.

10.3 Compilando expressões regulares

As expressões regulares passam por várias fases de processamento e entender o que acontece durante cada fase pode nos ajudar a otimizar o código JavaScript que usa expressões regulares. As duas fases principais são compilação e execução.

Compilação ocorre quando a expressão regular é criada pela primeira vez. *Execução* ocorre quando usamos a expressão regular compilada para combinar padrões em uma string.

Durante a compilação, a expressão é analisada pelo mecanismo JavaScript e convertida em sua representação interna (seja ela qual for). Esta fase de análise e conversão deve ocorrer toda vez que uma expressão regular é criada (descontando quaisquer otimizações internas realizadas pelo navegador).

Freqüentemente, navegadores *estamos* inteligente o suficiente para determinar quando expressões regulares idênticas estão sendo usadas e para armazenar em cache os resultados da compilação para aquela expressão específica. Mas não podemos contar com isso sendo o caso em todos os navegadores. Para expressões complexas, em particular, podemos começar a obter algumas melhorias de velocidade perceptíveis predefinindo (e, portanto, pré-compilando) nossas expressões regulares para uso posterior.

Como aprendemos em nossa visão geral das expressões regulares na seção anterior, há duas maneiras de criar uma expressão regular compilada em JavaScript: por meio de um literal e por meio de um construtor. Vejamos um exemplo na lista a seguir.

Listagem 10.2 Duas maneiras de criar uma expressão regular compilada

```
const re1 = / teste / i;                                    <-- Cria um regex por meio de um literal
const re2 = novo RegExp ("teste", "i"); assert (re1.toString () === "/
test / i",                                                  <-- Cria um regex por meio do construtor
        "Verifique o conteúdo da expressão.");
assert (re1.test ("TesT"), "Sim, não faz distinção entre maiúsculas e minúsculas."); assert (re2.test ("TesT"),
"Este também é."); assert (re1.toString () === re2.toString (),

        "As expressões regulares são iguais.");
assert (re1! == re2, "Mas eles são objetos diferentes.");
```

Neste exemplo, ambas as expressões regulares estão em seu estado compilado após a criação. Se substituíssemos todas as referências a re1 com o literal / test / i, é possível que o mesmo regex seja compilado repetidamente, portanto, compilar um regex *uma vez* e armazená-lo em uma variável para referência posterior pode ser uma otimização importante.

Observe que cada regex tem uma representação de objeto exclusiva: Cada vez que uma expressão regular é criada (e, portanto, compilada), um novo objeto de expressão regular é criado.

Isso é diferente de outros tipos primitivos (como string, número e assim por diante), porque o resultado sempre será único.

De particular importância é o uso do construtor (novo RegExp (...)) para criar uma expressão regular. Essa técnica nos permite construir e compilar uma expressão a partir de uma string que podemos criar dinamicamente em tempo de execução. Isso pode ser extremamente útil para construir expressões complexas que serão amplamente reutilizadas.

Por exemplo, digamos que desejamos determinar quais elementos em um documento têm um nome de classe específico, cujo valor não saberemos até o tempo de execução. Como os elementos podem ter vários nomes de classes associados a eles (armazenados de forma inconveniente em uma string delimitada por espaço), isso serve como um exemplo interessante de tempo de execução, compilação de expressão regular (consulte a lista a seguir).

Listagem 10.3 Compilando uma expressão regular de tempo de execução para uso posterior

```
<div class = "samurai ninja"> </div>
<div class = "ninja samurai"> </div>
<div> </div>
<span class = "samurai ninja ronin"> </span> <script>

    function findClassInElements (className, type) {
        const elems =
            document.getElementsByTagName (digite || "*");
        const regex =
            new RegExp ("(^ | \\ s)" + className + "(\\ s | $)"); resultados const = [];

        para (seja i = 0, comprimento = elems.length; i <comprimento; i ++)
            if (regex.test (elems [i] .className)) {
                results.push (elems [i]);
            }
        resultados de retorno;
    }
    assert (findClassInElements ("ninja", "div"). length === 2,
                "A quantidade certa de ninjas div foi encontrada.");
    assert (findClassInElements ("ninja", "span"). length === 1,
                "A quantidade certa de ninjas span foi encontrada."); assert (findClassInElements
    ("ninja"). length === 3,
                "A quantidade certa de ninjas foi encontrada.");
</script>
```

Cria assuntos de teste de vários elementos com vários nomes de classe

Coleta elementos por tipo

Compila um regex usando o nome de classe passado

Armazena o resultados

Testes para correspondências de regex

Podemos aprender várias coisas interessantes na Listagem 10.3. Para começar, configuramos uma série de assuntos de teste < div> e < span> elementos com várias combinações de nomes de classes. Em seguida, definimos nossa função de verificação de nome de classe, que aceita como parâmetros o nome da classe para a qual verificaremos e o tipo de elemento a ser verificado.

Em seguida, coletamos todos os elementos do tipo especificado usando o getElementsByTagName método integrado e configure nossa expressão regular:

```
const regex = new RegExp ("(^ | \\ s)" + className + "(\\ s | $)");
```

Observe o uso do novo RegExp () construtor para compilar uma expressão regular com base no nome da classe passado para a função. Esta é uma instância em que não podemos usar um literal de regex, pois o nome da classe que pesquisaremos não é conhecido de antemão.

Construímos (e, portanto, compilamos) esta expressão uma vez para evitar recompilações frequentes e desnecessárias. Porque o conteúdo da expressão é dinâmico (com base na entrada nome da classe argumento), podemos obter grandes economias de desempenho manipulando a expressão dessa maneira.

A própria regex corresponde ao início da string ou a um caractere de espaço em branco, seguido pelo nome da classe de destino, seguido por um caractere de espaço em branco ou o final da string. Observe o uso de um escape duplo (\\) no novo regex: \\ s.

Ao criar expressões regulares literais com termos incluindo a barra invertida, temos que fornecer a barra invertida apenas uma vez. Mas, como estamos escrevendo essas barras invertidas em uma string, devemos fazer um duplo escape delas. Isso é um incômodo, com certeza, mas devemos estar atentos ao construir expressões regulares em strings em vez de literais.

Depois que o regex é compilado, usá-lo para coletar os elementos correspondentes é fácil por meio do teste método:

```
regex.test (elems [i] .className)
```

A pré-construção e pré-compilação de expressões regulares para que possam ser reutilizadas (executadas) vezes sem conta é uma técnica recomendada que fornece ganhos de desempenho que não podem ser ignorados. Praticamente todas as situações complexas de expressão regular podem se beneficiar com o uso dessa técnica.

Anteriormente neste capítulo, mencionamos que o uso de parênteses em expressões regulares não serve apenas para agrupar termos para a aplicação do operador, mas também cria *captura.*

Vamos descobrir mais sobre isso.

10.4 Captura de segmentos correspondentes

A altura da utilidade em relação às expressões regulares é percebida quando nós *capturar* os resultados que são encontrados para que possamos fazer algo com eles. Determinar se uma string corresponde a um padrão é um primeiro passo óbvio e muitas vezes tudo de que precisamos, mas determinar *o que* foi correspondido também é útil em muitas situações.

10.4.1 Execução de capturas simples

Digamos que queremos extrair um valor embutido em uma string complexa. Um bom exemplo de tal string é o valor da propriedade de transformação do CSS, por meio do qual podemos modificar a posição visual de um elemento HTML.

Listagem 10.4 Uma função simples para capturar um valor incorporado

```
<div id = "square" style = "transform: translateY (15px);"> </div> <script>

    function getTranslateY (elem) {
        const transformValue = elem.style.transform;
```

Define o cobaia

```
        if (transformValue) {
            const match = transformValue.match (/ translateY \ (([^ \)] +) \) /); jogo de volta? correspondência [1]: "";

        }

        Retorna "";
    }

    const square = document.getElementById ("square");

    assert (getTranslateY (square) === "15px",
                "Extraímos o valor translateY");
</script>
```

Extrai o translateY valor da string

Definimos um elemento que especifica o estilo que traduzirá sua posição em 15 px:

```
"transform: translateY (15px);"
```

Infelizmente, o navegador não oferece uma API para buscar facilmente a quantidade pela qual o elemento é traduzido. Então, criamos nossa própria função:

```
function getTranslateY (elem) {
        const transformValue = elem.style.transform; if (transformValue) {

            const match = transformValue.match (/ translateY \ (([^ \)] +) \) /); jogo de volta? correspondência [1]: "";

        }
          Retorna "";
      }
```

O código de análise de transformação pode parecer confuso no início:

```
const match = transformValue.match (/ translateY \ (([^ \)] +) \) /); jogo de volta? correspondência [1]: "";
```

Mas não é tão ruim quando o quebramos. Para começar, precisamos determinar se um transformar até mesmo existe propriedade para analisarmos. Caso contrário, retornaremos uma string vazia. Se o transformar propriedade é residente, podemos descer para a extração do valor de opacidade. O Combine método de uma expressão regular retorna uma matriz de valores capturados se uma correspondência for encontrada, ou nulo se nenhuma correspondência for encontrada.

A matriz retornada por Combine inclui toda a correspondência no primeiro índice e, em seguida, cada captura subsequente. Portanto, a entrada zero seria toda a sequência combinada de translateY (15px), e a entrada na próxima posição seria 15px.

Lembre-se de que as capturas são definidas por parênteses na expressão regular. Assim, quando combinamos o valor de transformação, o valor está contido no [1] posição da matriz, porque a única captura que especificamos em nossa regex foi criada pelos parênteses que incorporamos após o traduzir Y parte do regex.

Este exemplo usa uma expressão regular local e o Combine método. As coisas mudam quando usamos expressões globais. Vamos ver como.

10.4.2 Correspondência usando expressões globais

Como vimos na seção anterior, usando uma expressão regular local (uma sem o sinalizador global) com o Fragmento do objeto Combine métodos retorna uma matriz contendo toda a string correspondida, junto com quaisquer capturas correspondidas na operação.

Mas quando fornecemos uma expressão regular global (uma com a g bandeira incluída), Combine retorna algo diferente. Ainda é uma matriz de resultados, mas no caso de uma expressão regular global, que corresponde a todas as possibilidades na string candidata, e não apenas à primeira correspondência, a matriz retornada contém as correspondências globais; captura *dentro de* cada correspondência não é retornada neste caso.

Podemos ver isso em ação no código e nos testes a seguir.

Listagem 10.5 Diferenças entre pesquisas globais e locais com Combine

```
const html = "<div class = 'test'> <b> Olá </b> <i>world!</i> </div>"; resultados const = html.match (/ <(\ /?) (\ w +) ([^>] *?)> /);

assert (results [0] === "<div class = 'test'>", "A correspondência inteira."); assert (results [1] === "", "A barra (ausente).");

assert (resultados [2] === "div", "O nome da tag.");
assert (resultados [3] === "classe = 'teste'", "Os atributos.");

const all = html.match (/ <(\ /?) (\ w +) ([^>] *?)> / g); assert (all [0] === "<div class = 'test'>", "Abrindo tag div.");
assert (all [1] === "<b>", "Abrindo tag b.");

assert (all [2] === "</b>", "Fechando tag b."); assert (all [3] === "<i>", "Abrindo tag i."); assert
(all [4] === "</i>", "Fechando a tag i."); assert (all [5] === "</div>", "Fechando tag div.");
```

Jogos usando um regex local

Jogos usando um regex global

Podemos ver que quando fazemos uma partida local, html.match (/ <(\ /?) (\ w +) ([^>] *?)> /), uma única instância é correspondida e as capturas nessa correspondência também são retornadas. Mas quando usamos uma correspondência global, html.match (/ <(\ /?) (\ w +) ([^>] *?)> / g), o que é retornado é a lista de correspondências.

Se as capturas forem importantes para nós, podemos recuperar essa funcionalidade enquanto ainda fazemos uma pesquisa global usando a expressão regular exec método. Este método pode ser repetidamente chamado em uma expressão regular, fazendo com que ela retorne o próximo conjunto de informações correspondentes toda vez que for chamada. Um padrão típico de uso é mostrado na lista a seguir.

Listagem 10.6 Usando o exec método para fazer a captura e uma pesquisa global

```
const html = "<div class = 'test'> <b> Olá </b> <i>world!</i> </div>"; tag const = / <(\ /?) (\ w +) ([^>] *?)> / g;

deixe corresponder, num = 0;
while ((match = tag.exec (html))! == null) {
    assert (match.length === 4,
            "Cada correspondência encontra cada tag e 3 capturas.");
    num ++;
}
assert (num === 6, "3 tags de abertura e 3 tags de fechamento encontradas.");
```

Repetidamente chama exec

Neste exemplo, chamamos repetidamente o exec método:

```
while ((match = tag.exec (html))! == null) {...}
```

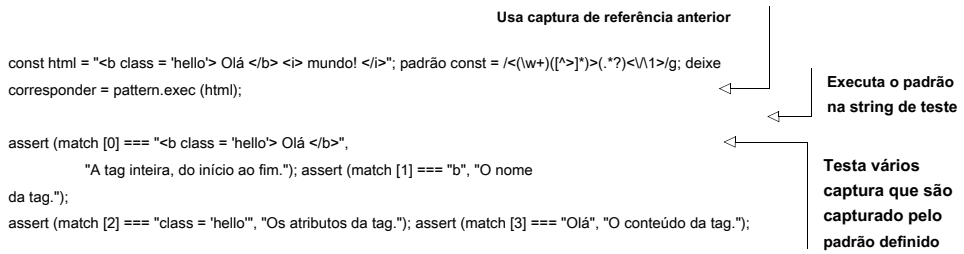
Isso retém o estado da chamada anterior para que cada chamada subsequente avance para a próxima correspondência global. Cada chamada retorna a próxima partida *e* suas capturas.

Usando qualquer Combine ou exec, sempre podemos encontrar as correspondências exatas (e capturas) que estamos procurando. Mas precisamos ir mais longe se quisermos nos referir às próprias capturas dentro da regex.

10.4.3 Capturas de referência

Podemos nos referir a partes de uma correspondência que capturamos de duas maneiras: uma dentro da própria correspondência e outra dentro de uma string de substituição (quando aplicável). Por exemplo, vamos revisitar a correspondência na listagem 10.6 (na qual combinamos uma tag HTML de abertura ou fechamento) e modificá-la na lista a seguir para também corresponder ao conteúdo interno da própria tag.

Listagem 10.7 Usando referências anteriores para corresponder ao conteúdo de uma tag HTML




```
const html = "<b class = 'hello'> Olá </b> <i> mundo! </i>"; padrão const = /<(\w+)([^>]*)>(.*?)<\/\1>/g; deixe
corresponder = pattern.exec (html);

assert (match [0] === "<b class = 'hello'> Olá </b>",
        "A tag inteira, do início ao fim."); assert (match [1] === "b", "O nome
da tag.");
assert (match [2] === "class = 'hello'", "Os atributos da tag."); assert (match [3] === "Olá", "O conteúdo da tag.");

match = pattern.exec (html);
assert (match [0] === "<i> mundo! </i>",
        "A tag inteira, do início ao fim."); assert (match [1] === "i", "O nome da tag."); assert (match [2] ===
"", "Os atributos da tag."); assert (match [3] === "mundo!", "O conteúdo da tag.");
```

Nós usamos \ 1 para se referir à primeira captura dentro da expressão, que neste caso é o nome da tag. Usando essas informações, podemos combinar a tag de fechamento apropriada, referindo-se a qualquer que seja a captura correspondida. (Isso tudo pressupõe, é claro, que não existem tags incorporadas com o mesmo nome na tag atual, portanto, este dificilmente é um exemplo completo de correspondência de tag.)

Além disso, podemos obter referências de captura dentro da string de substituição de uma chamada para o substituir método. Em vez de usar os códigos de referência anterior, como na listagem 10.7, usamos a sintaxe de $ 1, $ 2, $ 3, através de cada número de captura. Aqui está um exemplo:

```
assert ("fontFamily" .replace (/ ([AZ]) / g, "- $ 1"). toLowerCase () ===
          "font-family", "Converta o camelCase em notação tracejada.");
```

Neste código, o valor da primeira captura (neste caso, a letra maiúscula F) é referenciado no *substituir string (* via $ 1). Isso nos permite especificar uma string de substituição, mesmo sem saber qual será seu valor até o momento da correspondência. Essa é uma arma poderosa do tipo ninja para empunhar.

A capacidade de fazer referência a capturas de expressão regular ajuda a tornar bastante fácil muitos códigos que de outra forma seriam difíceis. A natureza expressiva que ele fornece acaba permitindo algumas declarações concisas que de outra forma poderiam ser um tanto obtusas, complicadas e extensas.

Como as capturas e o agrupamento de expressões são especificados usando parênteses, não há como o processador de expressões regulares saber quais conjuntos de parênteses adicionamos ao regex para agrupamento e quais deveriam indicar capturas. Ele trata todos os conjuntos de parênteses como grupos e capturas, o que pode resultar na captura de mais informações do que realmente pretendíamos, porque precisávamos especificar algum agrupamento na regex. O que podemos fazer nesses casos?

10.4.4 Grupos de não captura

Como observamos, os parênteses têm uma dupla função: eles não apenas agrupam termos para operações, mas também especificam capturas. Isso geralmente não é um problema, mas em expressões regulares nas quais muitos agrupamentos estão ocorrendo, isso pode causar muitas capturas desnecessárias, o que pode tornar a classificação das capturas resultantes entediante.

Considere o seguinte regex:

```
padrão const = / ((ninja -) +) espada /;
```

Aqui, a intenção é criar um regex que permita o prefixo ninja- aparecer uma ou mais vezes antes da palavra espada, e queremos capturar o prefixo inteiro. Este regex requer dois conjuntos de parênteses:

- Os parênteses que definem a captura (tudo antes da string espada)
- Os parênteses que agrupam o texto ninja- para o operador +

Tudo isso funciona bem, mas resulta em mais do que a única captura pretendida por causa do conjunto interno de parênteses de agrupamento.

Para indicar que um conjunto de parênteses não deve resultar em captura, a sintaxe da expressão regular permite colocar a notação?: Imediatamente após o parêntese de abertura. Isso é conhecido como *subexpressão passiva*.

Alterando esta expressão regular para

```
padrão const = / ((?: ninja -) +) espada /;
```

faz com que apenas o conjunto externo de parênteses crie uma captura. Os parênteses internos foram convertidos em uma subexpressão passiva.

Para testar isso, dê uma olhada no código a seguir.

Listagem 10.8 Agrupando sem capturar

```
padrão const = / ((?: ninja -) +) espada /;
const ninjas = "espada-ninja-ninja" .match (padrão);

assert (ninjas.length === 2, "Apenas uma captura foi retornada."); assert (ninjas [1] === "ninja-ninja-",

        "Combinou ambas as palavras, sem qualquer captura extra.");
```

Usa um passivo subexpressão

Executando esses testes, podemos ver que a subexpressão passiva / ((?: ninja -) +) espada / evita capturas desnecessárias.

Sempre que possível em nossas expressões regulares, devemos nos esforçar para usar grupos de não captura (passivos) no lugar da captura quando a captura for desnecessária, de modo que o mecanismo de expressão tenha muito menos trabalho a fazer para lembrar e retornar as capturas. Se não precisarmos dos resultados capturados, não há necessidade de solicitá-los! O preço que pagamos é que expressões regulares já complexas podem se tornar um pouco mais enigmáticas.

Agora vamos voltar nossa atenção para outra maneira que as expressões regulares nos dão poderes ninja: usando funções com o Fragmento do objeto substituir método.

10.5 Substituindo usando funções

O substituir método do Fragmento object é um método poderoso e versátil, que vimos usado brevemente em nossa discussão sobre capturas. Quando uma expressão regular é fornecida como o primeiro parâmetro para substituir, irá causar uma substituição em uma partida (ou *partidas* se a regex for global) para o padrão em vez de em uma string fixa.

Por exemplo, digamos que desejamos substituir todos os caracteres maiúsculos em uma string por X. Poderíamos escrever o seguinte:

```
"ABCDEfg" .replace (/ [AZ] / g, "X")
```

Isso resulta em um valor de XXXXXfg. Legal.

Mas talvez o recurso mais poderoso apresentado por substituir é a capacidade de fornecer uma função como valor de substituição, em vez de uma string fixa.

Quando o valor de substituição (o segundo argumento) é uma função, é invocado para cada correspondência encontrada (lembre-se de que uma pesquisa global irá corresponder a todas as instâncias do padrão na string de origem) com uma lista variável de parâmetros:

- O texto completo da partida
- As capturas da correspondência, um parâmetro para cada O índice da
- correspondência dentro da string original A string de origem
-

O valor retornado da função serve como valor de substituição.

Isso fornece uma enorme margem de manobra para determinar qual deve ser a string de substituição em tempo de execução, com muitas informações sobre a natureza da correspondência ao nosso alcance. Por exemplo, na lista a seguir, usamos a função para

fornece um valor de substituição dinâmica para converter uma string com palavras separadas por traços em seu equivalente em caixa de camelo.

Listagem 10.9: Convertendo uma string tracejada em caixa de camelo

```
função superior (tudo, letra) {return letter.toUpperCase (); } assert ("border-bottom-width" .replace (/ - (\ w) / g,
superior)
        === "borderBottomWidth",
      "Camel envolve uma string hifenizada.");
```

Converte para maiúsculas

Partidas tracejadas personagens

Aqui, fornecemos uma regex que corresponde a qualquer caractere precedido por um traço. Uma captura na regex global identifica o caractere que foi correspondido (sem o traço). Cada vez que a função é chamada (duas vezes neste exemplo), é passada a string de correspondência completa como o primeiro argumento e a captura (apenas uma para esta regex) como o segundo argumento. Não estamos interessados no restante dos argumentos, então não os especificamos.

Na primeira vez que a função é chamada, ela é passada - b e b; e na segunda vez que é chamado, já passou - C e C. Em cada caso, a letra capturada é maiúscula e retornada como a string de substituição. Acabamos com - b substituído por B e com - C

substituído por C.

Como uma regex global fará com que essa função de substituição seja executada para cada correspondência em uma string de origem, essa técnica pode até ser estendida além de fazer substituições rotineiras. Podemos usar a técnica como um meio de atravessar string, em vez de fazer o exec () - técnica de loop interno que vimos anteriormente neste capítulo.

Por exemplo, digamos que estamos procurando uma string de consulta e convertê-la em um formato alternativo que atenda aos nossos objetivos. Viraríamos uma string de consulta como

```
foo = 1 & foo = 2 & blah = a & blah = b & foo = 3
```

em um que se parece com este

```
foo = 1,2,3 & blah = a, b "
```

Uma solução usando expressões regulares e substituir pode resultar em algum código especialmente conciso, conforme mostrado na próxima listagem.

Listagem 10.10 Uma técnica para compactar uma string de consulta

```
função compress (fonte) {
    chaves const = {};
    source.replace (
        / ([^ = &] +) = ([^ &] *) / g,
        função (completa, chave, valor) {
            chaves [chave] =
                (teclas [tecla]? teclas [tecla] + ",": "") + valor; Retorna "";

        }
```

Armazena chaves localizadas

Extrai informações de valor-chave

```
    );
    resultado const = [];
    para (deixe digitar as chaves) {
        result.push (key + "=" + keys [key]);
    }
    return result.join ("&");
}

assert (compress ("foo = 1 & foo = 2 & blah = a & blah = b & foo = 3") ===
                "foo = 1,2,3 & blah = a, b",
                "A compressão está OK!");
```

Coleta
informação chave

Une resultados com &

O aspecto mais interessante deste exemplo é o uso da string substituir método como um meio de percorrer uma string para obter valores, em vez de um mecanismo de pesquisa e substituição. O truque é duplo: passar uma função como o argumento do valor de substituição e, em vez de retornar um valor, usá-lo como meio de pesquisa.

O código de exemplo primeiro declara um hash chave em que armazenamos as chaves e os valores que encontramos na string de consulta de origem. Então chamamos o substituir na string de origem, passando um regex que corresponderá aos pares de valores-chave e capturará a chave e o valor. Também passamos uma função que receberá a correspondência completa, a captura de chave e a captura de valor. Esses valores capturados são armazenados no hash para referência posterior. Observe que retornamos a string vazia porque não nos importamos com quais substituições acontecem na string de origem - estamos apenas usando os efeitos colaterais em vez do resultado.

Depois de substituir retorna, declaramos uma matriz na qual agregaremos os resultados e iteraremos por meio das chaves que encontramos, adicionando cada uma à matriz. Por fim, juntamos cada um dos resultados que armazenamos na matriz usando & como delimitador e retornamos o resultado:

```
resultado const = [];
para (deixe digitar as chaves) {
    result.push (key + "=" + keys [key]);
}
return result.join ("&");
```

Usando esta técnica, podemos cooptar o Fragmento do objeto substituir método como nosso próprio mecanismo de pesquisa de string. O resultado não é apenas rápido, mas também simples e eficaz. O nível de poder que essa técnica fornece, especialmente em vista da pequena quantidade de código necessária, não deve ser subestimado.

Todas essas técnicas de expressão regular podem ter um grande impacto em como escrevemos o script em nossas páginas. Vamos ver como aplicar o que você aprendeu para resolver alguns problemas comuns que podemos encontrar.

10.6 Resolvendo problemas comuns com expressões regulares

Em JavaScript, alguns idiomas tendem a ocorrer repetidamente, mas suas soluções nem sempre são óbvias. O conhecimento das expressões regulares pode definitivamente vir em nosso auxílio e, nesta seção, veremos alguns problemas comuns que podemos resolver com um regex ou dois.