Geradores são um tipo especial de função. Enquanto uma função padrão produz no máximo um único valor enquanto executa seu código do início ao fim, os geradores produzem vários valores, por solicitação, enquanto suspendem sua execução entre essas solicitações. Embora novos em JavaScript, os geradores já existem há algum tempo em Python, PHP e C #.

Os geradores são frequentemente considerados um recurso de linguagem quase estranho ou marginal que não costuma ser usado pelo programador médio. Embora a maioria dos exemplos deste capítulo sejam projetados para ensinar como funcionam as funções do gerador, também exploraremos alguns aspectos práticos dos geradores. Você verá como usar geradores para simplificar loops complicados e como tirar proveito da capacidade dos geradores de suspender e retomar sua execução, o que pode ajudá-lo a escrever um código assíncrono mais simples e elegante.

Promessas, por outro lado, são um novo tipo de objeto integrado que o ajuda a trabalhar com código assíncrono. Uma promessa é um espaço reservado para um valor que ainda não temos, mas que teremos em algum momento posterior. Eles são especialmente bons para trabalhar com várias etapas assíncronas.

Neste capítulo, você verá como funcionam os geradores e as promessas, e terminaremos explorando como combiná-los para simplificar muito nossas negociações com o código assíncrono. Mas antes de entrar em detalhes, vamos dar uma espiada em como nosso código assíncrono pode ser muito mais elegante.

Você sabe?

Quais são alguns usos comuns para uma função de gerador? Por que as promessas são melhores do que simples retornos de chamada para asyn- código crônico?

Você inicia uma série de tarefas de longa duração com Promise . raça. Quando a promessa se resolve? Quando isso não resolveria?

6,1 Tornando nosso código assíncrono elegante com geradores e promessas

Imagine que você seja um desenvolvedor trabalhando no freelanceninja.com, um popular site de recrutamento de ninjas freelance que permite aos clientes contratar ninjas para missões furtivas. Sua tarefa é implementar uma funcionalidade que permita aos usuários obter detalhes sobre a missão mais bem avaliada realizada pelo ninja mais popular. Os dados que representam os ninjas, os resumos de suas missões, bem como os detalhes das missões são armazenados em um servidor remoto, codificado em JSON. Você pode escrever algo assim:

```
tentar {
    var ninjas = syncGetJSON ("ninjas.json");
    var missions = syncGetJSON (ninjas [0] .missionsUrl);
    var missionDetails = syncGetJSON (missões [0] .detailsUrl); // Estude a descrição da missão

}
pegar (e) {
  // Oh não, não conseguimos obter os detalhes da missão
}
```

Este código é relativamente fácil de entender e, se ocorrer um erro em qualquer uma das etapas, podemos detectá-lo facilmente no pegar quadra. Mas, infelizmente, esse código tem um grande problema. Obter dados de um servidor é uma operação de longa duração e, como o JavaScript depende de um modelo de execução de thread único, apenas bloqueamos nossa IU até que as operações de longa duração terminem. Isso leva a aplicativos que não respondem e usuários decepcionados. Para resolver esse problema, podemos reescrevê-lo com retornos de chamada, que serão invocados quando uma tarefa terminar, sem bloquear a IU:

```
getJSON ("ninjas.json", function (err, ninjas) {
    if (err) {
        console.log ("Erro ao buscar lista de ninjas", err); Retorna;

    }
    getJSON (ninjas [0] .missionsUrl, function (err, missões) {
        if (err) {
            console.log ("Erro ao localizar missões ninja", err); Retorna;

        }
    getJSON (missions [0] .detailsUrl, function (err, missionDetails) {
        if (err) {
            console.log ("Erro ao localizar detalhes da missão", err); Retorna;

        }
        // Estude o plano de inteligência});

    });
});
```

Embora esse código seja muito melhor recebido por nossos usuários, você provavelmente concordará que é confuso, adiciona muito código clichê de tratamento de erros e é totalmente feio. É aqui que os geradores e as promessas entram em ação. Ao combiná-los, podemos transformar o código de callback não bloqueador, mas estranho, em algo muito mais elegante:

Uma função geradora é definida colocando um asterisco logo após a palavra-chave da função. Podemos usar a nova palavra-chave yield em funções geradoras.

As promessas estão ocultas no método getJSON.

```
assíncrono (função * () {
    tentar {
        const ninjas = rendimento getJSON ("ninjas.json");
        missões const = rendimento getJSON (ninjas [0] .missionsUrl);
        const missionDescription = rendimento getJSON (missões [0] .detailsUrl); // Estude os detalhes da missão

    }
    pegar (e) {
        // Oh não, não conseguimos obter os detalhes da missão
    }
});
```

Não se preocupe se este exemplo não fizer muito sentido ou se você encontrar alguma sintaxe (como função* ou produzem) desconhecido. No final deste capítulo, você conhecerá todos

os ingredientes necessários. Por enquanto, é suficiente que você compare a elegância (ou a falta dela) do código de retorno de chamada não bloqueador e os geradores não bloqueadores e código de promessas.

Vamos começar devagar explorando as funções do gerador, o primeiro trampolim para um código assíncrono elegante.

6,2 Trabalhando com funções do gerador

Os geradores são um tipo de função completamente novo e são significativamente diferentes das funções padrão comuns. UMA *gerador* é uma função que gera uma sequência de valores, mas não todos de uma vez, como faria uma função padrão, mas por solicitação. Temos que pedir explicitamente ao gerador um novo valor, e o gerador responderá com um valor ou nos notificará de que não há mais valores para produzir. O que é ainda mais curioso é que depois que um valor é produzido, uma função geradora não termina sua execução, como faria uma função normal. Em vez disso, um gerador está simplesmente suspenso. Então, quando uma solicitação de outro valor chega, o gerador continua de onde parou.

A lista a seguir fornece um exemplo simples de uso de um gerador para gerar uma sequência de armas.

Listagem 6.1 Usando uma função geradora para gerar uma sequência de valores

```
função* WeaponGenerator () {
    produzem " Katana ";
    produzem " Wakizashi ";
    produzem " Kusarigama ";
}

para (deixar arma de WeaponGenerator ()) {
    afirmar (arma! == indefinido, arma);
}
```

Define uma função geradora colocando * após a palavra-chave da função

Gera valores individuais usando a nova palavra-chave de rendimento

Consome o gerado sequência com o novo loop for-of

Começamos definindo um gerador que produzirá uma sequência de armas. Criar uma função geradora é simples: acrescentamos um asterisco (*) após o função palavra-chave. Isso nos permite usar o novo produzem palavra-chave dentro do corpo do gerador para produzir valores individuais. A Figura 6.1 ilustra a sintaxe.

Neste exemplo, criamos um gerador chamado WeaponGenerator que produz uma sequência de armas: Katana, Wakizashi, e Kusarigama. Uma maneira de consumir essa sequência de armas é usar um novo tipo de loop, o para de ciclo:

```
para (deixar arma de WeaponGenerator ()) {
    afirmar (arma, arma);
}
```

O resultado de invocar este para de loop é mostrado na figura 6.2. (Por enquanto, não se preocupe muito com o para de loop, como iremos explorar mais tarde.)

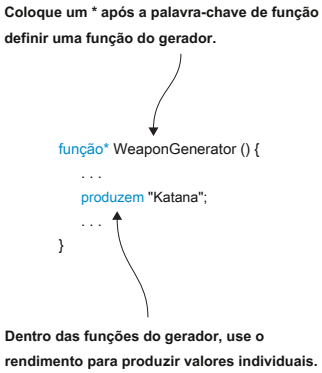


Figura 6.1 Adicione um asterisco (*) após a palavra-chave de função para definir um gerador.

Figura 6.2 O resultado da iteração em nosso **WeaponGenerator ()**

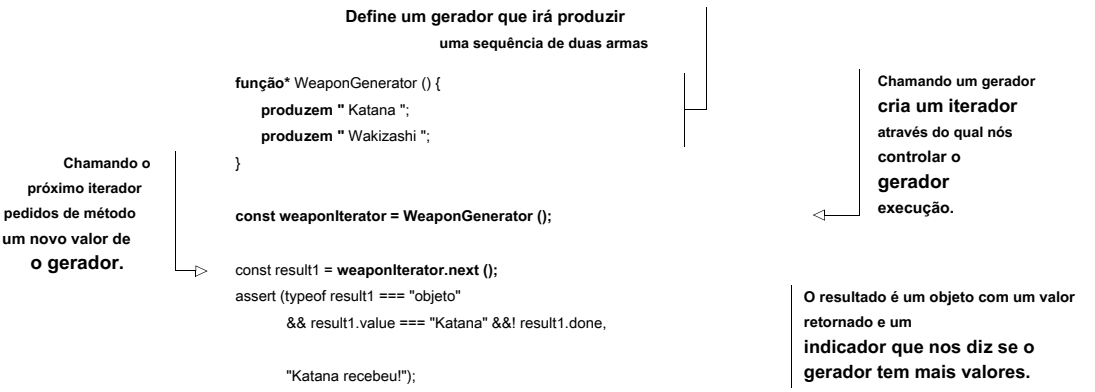
No lado direito do para de loop, colocamos o resultado de invocar nosso gerador. Mas se você der uma olhada mais de perto no corpo do WeaponGenerator função, você verá que não há Retorna demonstração. O que há com isso? Neste caso, o lado direito do loop for-of avaliar para Indefinido, como seria o caso se estivéssemos lidando com uma função padrão?

A verdade é que os geradores são bastante diferentes das funções padrão. Para começar, chamar um gerador não executa a função do gerador; em vez disso, ele cria um objeto chamado de *iterador*. Vamos explorar esse objeto.

6.2.1 Controlar o gerador por meio do objeto iterador

Fazer uma chamada para um gerador não significa que o corpo da função do gerador será executado. Em vez disso, um objeto iterador é criado, um objeto por meio do qual podemos nos comunicar com o gerador. Por exemplo, podemos usar o iterador para solicitar valores adicionais. Vamos ajustar nosso exemplo anterior para explorar como o objeto iterador funciona.

Listagem 6.2 Controlando um gerador por meio de um objeto iterador



```
const result2 = weaponIterator.next (); assert (typeof result2 === "objeto"
        && result2.value === "Wakizashi" &&! result2.done,
        "Wakizashi recebeu!");
```

Ligando em seguida novamente obtém outro valor do gerador.

```
const result3 = weaponIterator.next (); assert (typeof result3 === "objeto"
        && result3.value === undefined && result3.done,
        "Não há mais resultados!");
```

Quando não há mais código para executar, o gerador retorna "indefinido" e indica que está feito.

Como você pode ver, quando chamamos um gerador, um novo *iterador* é criado:

```
const weaponIterator = WeaponGenerator ();
```

O iterador é usado para controlar a execução do gerador. Uma das coisas fundamentais que o objeto iterador expõe é o Próximo método, que podemos usar para controlar o gerador, solicitando um valor dele:

```
const result1 = weaponIterator.next ();
```

Em resposta a essa chamada, o gerador executa seu código até atingir um produzem palavra-chave que produz um resultado intermediário (um item na sequência de itens gerada) e retorna um *novo* objeto que encapsula aquele resultado e nos diz se seu trabalho está concluído.

Assim que o valor atual é produzido, o gerador suspende sua execução sem bloquear e espera pacientemente por outra solicitação de valor. Este é um recurso incrivelmente poderoso que as funções padrão não possuem, um recurso que usaremos posteriormente com grande efeito.

Neste caso, a primeira chamada para o iterador Próximo método executa o código do gerador para o primeiro produzem expressão, rendimento "Katana", e retorna um objeto com a propriedade valor definido como Katana e a propriedade feito definido como falso, sinalizando que há mais valores a serem produzidos.

Mais tarde, solicitamos outro valor do gerador, fazendo outra chamada para o weaponIterator de Próximo método:

```
const result2 = weaponIterator.next ();
```

Isso tira o gerador da suspensão, e o gerador continua de onde parou, executando seu código até que outro valor intermediário seja alcançado: rendimento "Wakizashi". Isso suspende o gerador e produz um objeto carregando Wakizashi.

Finalmente, quando chamamos o Próximo pela terceira vez, o gerador retoma sua execução. Mas desta vez não há mais código para executar, então o gerador retorna um objeto com valor definido como Indefinido, e feito definido como verdade, sinalizando que seu trabalho está concluído.

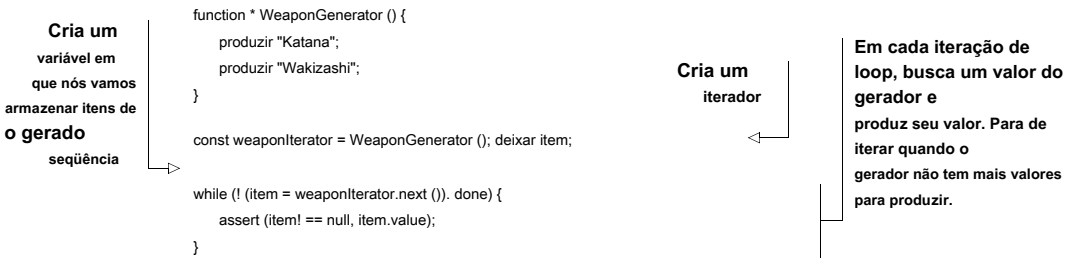
Agora que você viu como controlar geradores por meio de iteradores, está pronto para aprender como iterar sobre os valores produzidos.

ITERANDO O ITERADOR

O iterador, criado chamando um gerador, expõe um Próximo método através do qual podemos solicitar um novo valor do gerador. O Próximo método retorna um objeto que carrega o valor produzido pelo gerador, bem como as informações armazenadas no
feito propriedade que nos diz se o gerador tem valores adicionais para produzir.

Agora vamos aproveitar esses fatos para usar um velho e simples enquanto loop para iterar sobre os valores produzidos por um gerador. Veja a lista a seguir.

Listagem 6.3 Iterando sobre os resultados do gerador com um **enquanto** ciclo



Aqui, criamos novamente um objeto iterador chamando uma função geradora:

```
const weaponIterator = WeaponGenerator ();
```

Nós também criamos um item variável na qual armazenaremos valores individuais produzidos pelo gerador. Continuamos especificando um enquanto loop com uma condição de loop ligeiramente complicada, que vamos quebrar um pouco:

```
while (! (item = weaponIterator.next ()). done) {
    afirmar (item! == null, item.value)
}
```

Em cada iteração de loop, buscamos um valor do gerador chamando o Próximo método do nosso weaponIterator, e armazenamos esse valor no item variável. Como todos esses objetos, o objeto referenciado pelo item variável tem um valor propriedade que armazena o valor retornado do gerador, e um feito propriedade que sinaliza se o gerador terminou de produzir valores. Se o gerador não terminar seu trabalho, vamos para outra iteração do loop; e se for, paramos o loop.

E é assim que para de loop, do nosso primeiro exemplo de gerador, funciona. O para de loop é um açúcar sintático para iteração sobre iteradores:

```
para (item var de WeaponGenerator ()) {
    afirmar (item! == nulo, item);
}
```

Em vez de chamar manualmente o Próximo método do iterador correspondente e sempre verificando se terminamos, podemos usar o para de loop para fazer exatamente a mesma coisa, apenas nos bastidores.

RENDENDO A OUTRO GERADOR

Assim como frequentemente chamamos uma função padrão de outra função padrão, em certos casos queremos ser capazes de delegar a execução de um gerador a outro. Vamos dar uma olhada em um exemplo que gera guerreiros e ninjas.

Listagem 6.4 usando **produzem*** para delegar a outro gerador

```
function * WarriorGenerator () {
    produzir "Sun Tzu";
    produzem* NinjaGenerator ();          <-- yield * delega para outro gerador.
    produzir "Genghis Khan";
}

function * NinjaGenerator () {
    produzir "Hattori";
    produzir "Yoshi";
}

for (let warrior of WarriorGenerator ()) {
    afirmar (guerreiro! == null, guerreiro);
}
```

Se você executar este código, verá que a saída é Sun Tzu, Hattori, Yoshi, Genghis Khan.
Gerando Sun Tzu provavelmente não o pega desprevenido; é o primeiro valor produzido pelo WarriorGenerator. Mas a segunda saída, Hattori, merece uma explicação.

Usando o produzem* operador em um iterador, cedemos a outro gerador. Neste exemplo, a partir de WarriorGenerator estamos cedendo a um novo NinjaGenerator; todas as chamadas para o atual WarriorGenerator iterador Próximo método são redirecionados para o NinjaGenerator. Isso dura até o NinjaGenerator não tem trabalho a fazer. Então, em nosso exemplo, após Sun Tzu, o programa gera Hattori e Yoshi. Somente quando o NinjaGenerator é feito com o seu trabalho; a execução do iterador original continuará gerando Genghis Khan. Observe que isso está acontecendo de forma transparente com o código que chama o gerador original. O para de loop não se importa que o WarriorGenerator cede a outro gerador; continua ligando Próximo até que esteja feito.

Agora que você tem uma ideia de como os geradores funcionam em geral e como funciona a delegação para outro gerador, vamos dar uma olhada em alguns exemplos práticos.

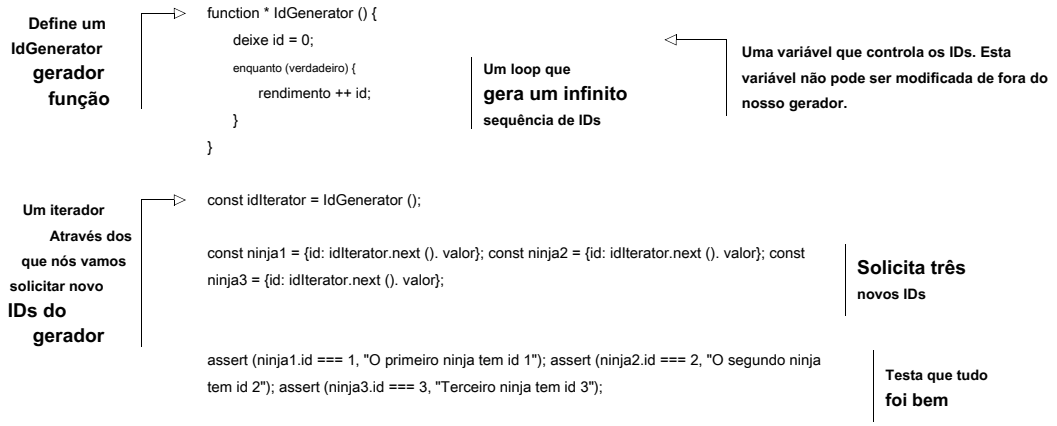
6.2.2 Usando geradores

Gerar sequências de itens é muito bom e elegante, mas vamos ser mais práticos, começando com um caso simples de geração de IDs únicos.

USANDO GERADORES PARA GERAR IDS

Ao criar certos objetos, muitas vezes precisamos atribuir um ID exclusivo para cada objeto. A maneira mais fácil de fazer isso é por meio de uma variável de contador global, mas isso é meio feio porque a variável pode ser acidentalmente bagunçada de qualquer lugar em nosso código. Outra opção é usar um gerador, conforme mostrado na lista a seguir.

Listagem 6.5 Usando geradores para gerar IDs



Este exemplo começa com um gerador que possui uma variável local, eu ia, que representa nosso contador de ID. O eu ia variável é local para nosso gerador; não há medo de que alguém o modifique acidentalmente em algum outro lugar do código. Isso é seguido por um infinito enquanto loop, que a cada iteração produz um novo eu ia valor e suspende sua execução até que uma solicitação de outro ID apareça:

```
function * IdGenerator () {
    deixe id = 0;
    enquanto (verdadeiro) {
        rendimento ++ id;
    }
}
```

NOTA Escrever loops infinitos não é algo que geralmente queremos fazer em uma função padrão. Mas com geradores está tudo bem! Sempre que o gerador encontra um produzem declaração, a execução do gerador é suspensa até o Próximo método é chamado novamente. Então todo Próximo chamada executa apenas uma iteração de nosso infinito enquanto loop e envia de volta o próximo valor de ID.

Depois de definir o gerador, criamos um objeto iterador:

```
const idIterator = IdGenerator ();
```

Isso nos permite controlar o gerador com chamadas para o idIterator.next () método. Isso executa o gerador até um produzem é encontrado, retornando um novo valor de ID que podemos usar para nossos objetos:

```
const ninja1 = {id: idIterator.next (). valor};
```

Veja como isso é simples? Nenhuma variável global confusa cujo valor pode ser alterado acidentalmente. Em vez disso, usamos um iterador para solicitar valores de um gerador. Além disso, se

mais tarde, precisamos de outro iterador para rastrear os IDs de, por exemplo, samurai, podemos inicializar um novo gerador para isso.

USANDO GERADORES PARA TRAVESSAR O DOM

Como você viu no capítulo 2, o layout de uma página da web é baseado no DOM, uma estrutura em forma de árvore de nós HTML, em que cada nó, exceto o raiz, tem exatamente um pai e pode ter zero ou mais filhos . Como o DOM é uma estrutura tão fundamental no desenvolvimento da web, muito do nosso código é baseado em atravessá-lo. Uma maneira relativamente fácil de fazer isso é implementando uma função recursiva que será executada para cada nó visitado. Veja o seguinte código.

Listagem 6.6 Traversal de DOM recursivo

```
<div id = "subTree">
    <form>
        <input type = "text" />
    </form>
    <p> Parágrafo </p>
    <span> Span </span>
</div>
<script>
    function traverseDOM (elemento, retorno de chamada) {
        retorno de chamada (elemento);
        element = element.firstElementChild;
        while (elemento) {
            traverseDOM (elemento, retorno de chamada);
            element = element.nextElementSibling;
        }
    }
    const subTree = document.getElementById ("subTree"); traverseDOM (subárvore, função
    (elemento) {
        assert (element! == null, element.nodeName); });
</script>
```

Processa o atual nó com um retorno de chamada

Percorre o DOM de cada elemento filho

Inicia todo o processo chamando o traverseDOM função para o nosso elemento raiz

Neste exemplo, usamos uma função recursiva para percorrer todos os descendentes do elemento com o subárvore id, no processo de registro de cada tipo de nó que visitamos. Neste caso, o código produz DIV, FORM, INPUT, P, e PERÍODO.

Já faz um tempo que escrevemos esse código de travessia do DOM e ele nos serviu perfeitamente. Mas agora que temos geradores à nossa disposição, podemos fazer isso de forma diferente; veja o código a seguir.

Listagem 6.7 Iterando em uma árvore DOM com geradores

```
function * DomTraversal (elemento) {
    elemento de rendimento;
    element = element.firstElementChild;
    while (elemento) {
        produzem* DomTraversal (elemento);
        element = element.nextElementSibling;
```

Usa yield * para transferir o controle de iteração para outra instância do gerador DomTraversal

```
    }
}

const subTree = document.getElementById ("subTree"); para (elemento let de DomTraversal
(subárvore)) {
    assert (element! == null, element.nodeName);
}
```

Itera sobre os nós usando o loop for-of

Esta listagem mostra que podemos obter travessias de DOM com geradores, tão facilmente quanto com recursão padrão, mas com o benefício adicional de não ter que usar a sintaxe um pouco estranha de callbacks. Em vez de processar a subárvore de cada nó visitado recorrendo a outro nível, criamos uma função geradora para cada nó visitado e nos rendemos a ela. Isso nos permite escrever o que é código conceitualmente recursivo de maneira iterável. O benefício é que podemos consumir a sequência gerada de nós com um simples para de loop, sem recorrer a retornos de chamada desagradáveis.

Este exemplo é particularmente bom, porque também mostra como usar geradores para separar o código que está produzindo valores (neste caso, nós HTML) do código que está consumindo a sequência de valores gerados (neste caso, o
para de loop que registra os nós visitados), sem ter que recorrer a callbacks. Além disso, usar iterações é, em certos casos, muito mais natural do que recursão, por isso é sempre bom ter nossas opções abertas.

Agora que exploramos alguns aspectos práticos dos geradores, vamos voltar a um tópico um pouco mais teórico e ver como trocar dados com um gerador em execução.

6.2.3 Comunicando-se com um gerador

Nos exemplos apresentados até agora, você viu como retornar vários valores *a partir de* um gerador usando produzem expressões. Mas os geradores são ainda mais poderosos do que isso! Nós também podemos enviar dados *para* um gerador, conseguindo assim comunicação bidirecional! Com um gerador, podemos produzir um resultado intermediário, usar esse resultado para calcular outra coisa de fora do gerador e, então, quando estivermos prontos, enviar dados completamente novos de volta para o gerador e retomar sua execução. Usaremos esse recurso com grande efeito no final do capítulo para lidar com o código assíncrono, mas, por enquanto, vamos mantê-lo simples.

ENVIANDO VALORES COMO ARGUMENTOS DE FUNÇÃO DO GERADOR

A maneira mais fácil de enviar dados para um gerador é tratando-os como qualquer outra função e usando argumentos de chamada de função. Dê uma olhada na lista a seguir.

Listagem 6.8 Enviando dados e recebendo dados de um gerador

Um gerador pode receber argumentos padrão, como qualquer outra função.

O valor enviado o próximo se torna o valor da expressão produzida, então nosso impostor é Hanzo.

```
function * NinjaGenerator (ação) {
    impostor const = rendimento ("Hattori" + ação);

assert (impostor === "Hanzo",
            "O gerador foi infiltrado");
```

A mágica acontece. Ao produzir um valor, o gerador retorna um cálculo intermediário. Ao chamar o próximo método do iterador com um argumento, enviamos os dados de volta ao gerador.

```
    rendimento ("Yoshi (" + impostor + ")" + ação);
}

const ninjaIterator = NinjaGenerator ("skulk");

const result1 = ninjaIterator.next ();
assert (result1.value === "Hattori skulk", "Hattori está se escondendo");

const result2 = ninjaIterator.next ("Hanzo"); assert (result2.value === "Yoshi (Hanzo)
skulk",
        "Temos um impostor!");
```

Normal argumento passagem

Inicia a execução do gerador e verifica se obtemos o valor correto

Envia dados para o gerador como um argumento para o próximo método e verifica se o valor foi transferido corretamente

Uma função que recebe dados não é nada especial; as funções antigas fazem isso o tempo todo. Mas lembre-se, os geradores têm esse poder incrível; eles podem ser suspensos e reiniciados. E acontece que, ao contrário das funções padrão, os geradores podem até receber dados *depois de* sua execução é iniciada, sempre que os retomamos solicitando o próximo valor.

USANDO O PRÓXIMO MÉTODO PARA ENVIAR VALORES PARA UM GERADOR

Além de fornecer dados ao invocar o gerador pela primeira vez, podemos enviar dados *em* um gerador passando argumentos para o Próximo método. Nesse processo, despertamos o gerador da suspensão e retomamos sua execução. Este valor passado é usado pelo gerador como o valor de todo produzem expressão, na qual o gerador estava atualmente suspenso, conforme mostrado na figura 6.3.

Neste exemplo, temos duas chamadas para o ninjaIterator de Próximo método. A primeira chamada, ninjaIterator.next (), solicita o primeiro valor do gerador. Como nosso gerador não começou a ser executado, esta chamada inicia o gerador, que calcula o

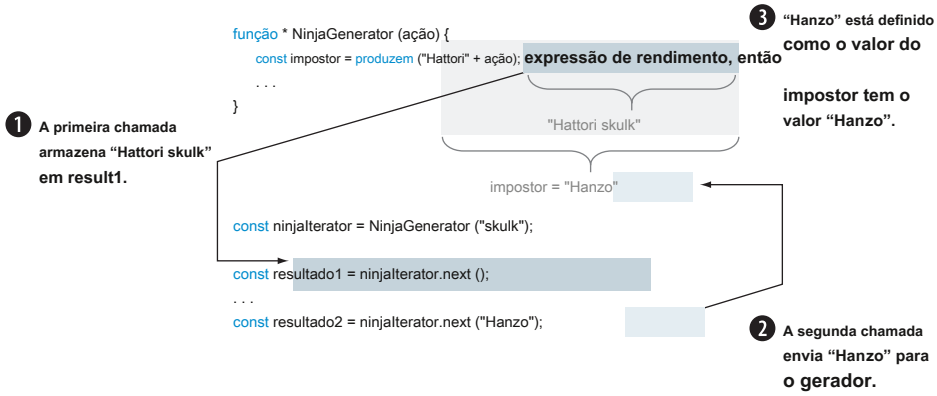


Figura 6.3 A primeira chamada para **ninjaIterator.next ()** solicita um novo valor do gerador, que retorna **Hattori skulk** e suspende a execução do gerador no **produzem** expressão. A segunda chamada para **ninjaIterator.next ("Hanzo")** solicita um novo valor, mas também envia o argumento **Hanzo** no gerador. Este valor será usado como o valor do todo **produzem** expressão, e a variável **impostor** agora carregará o valor **Hanzo.**

valor da expressão " Hattori "+ ação, produz o Hattori skulk valor e suspende a execução do gerador. Não há nada de especial nisso; fizemos algo semelhante várias vezes ao longo deste capítulo.

O interessante acontece na segunda chamada para o ninjaIterator de Próximo método: ninjaIterator.next ("Hanzo"). Desta vez, estamos usando o Próximo método para passar os dados de volta para o gerador. Nossa função geradora está esperando pacientemente, suspensa na expressão rendimento ("Hattori" + ação), então o valor Hanzo, passado como o argumento para Próximo(), é usado como o valor do todo produzem expressão. Na nossa caso, isso significa que a variável impostor no impostor = rendimento ("Hattori" + açao) vai acabar com o valor Hanzo.

É assim que alcançamos a comunicação bidirecional com um gerador. Nós usamos produzem para retornar dados de um gerador, e o iterador de Próximo() método para passar os dados de volta para o gerador.

NOTA O Próximo método fornece o valor para a espera produzem expressão, então se não houver produzem expressão esperando, não há nada para fornecer o valor. Por este motivo, nós *não pode* fornecer valores durante a primeira chamada para o Próximo método. Mas lembre-se, se você precisa fornecer um valor inicial para o gerador, você pode fazê-lo ao chamar o próprio gerador, como fizemos com NinjaGenerator ("skulk").

EXCEÇÕES DE LANÇAMENTO

Existe outra maneira, um pouco menos ortodoxa, de fornecer um valor a um gerador: lançando uma exceção. Cada iterador, além de ter um Próximo método, tem um lançar método que podemos usar para lançar uma exceção de volta para o gerador. Novamente, vamos ver um exemplo simples.

Listagem 6.9 Lançando exceções para geradores

```
function * NinjaGenerator () {
    tentar{
        produzir "Hattori";
        fail ("A exceção esperada não ocorreu");
    }
    pegar (e) {
        assert (e === "Catch this!", "Aha! Pegamos uma exceção");
    }
}

const ninjaIterator = NinjaGenerator ();

const result1 = ninjaIterator.next ();
assert (result1.value === "Hattori", "Temos Hattori");

ninjaIterator.throw ("Pegue isso!");
```

Esta falha não deveria ser alcançado.

Captura exceções e testa se nós recebemos o exceção esperada

Puxa um valor de o gerador

Lança uma exceção para o gerador

A Listagem 6.9 começa de forma semelhante à Listagem 6.8, especificando um gerador chamado NinjaGenerator. Mas, desta vez, o corpo do gerador é ligeiramente diferente. Rodeamos todo o código do corpo da função com um tentar pegar quadra:

```
function * NinjaGenerator () {
    tentar{
        produzir "Hattori";
        fail ("A exceção esperada não ocorreu");
    }
    pegar (e) {
        assert (e === "Catch this!", "Aha! Pegamos uma exceção");
    }
}
```

Em seguida, continuamos criando um iterador e obtendo um valor do gerador:

```
const ninjaIterator = NinjaGenerator (); const result1 = ninjaIterator.next ();
```

Finalmente, usamos o lançar método, disponível em todos os iteradores, para lançar uma exceção de volta ao gerador:

```
ninjaIterator.throw ("Pegue isso!");
```

Ao executar essa listagem, podemos ver que nosso lançamento de exceção funciona conforme o esperado, conforme mostrado na figura 6.4.

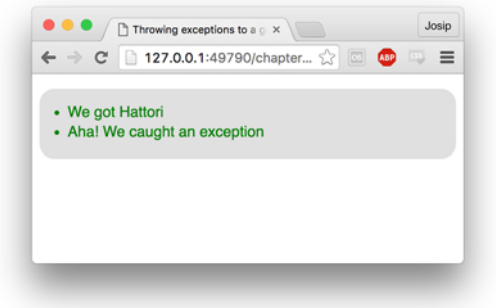


Figura 6.4 Podemos lançar exceções para geradores de fora de um gerador.

Esse recurso que nos permite lançar exceções de volta aos geradores pode parecer um pouco estranho no início. Por que faríamos isso? Não se preocupe; não vamos mantê-lo no escuro por muito tempo. No final deste capítulo, usaremos esse recurso para melhorar a comunicação assíncrona do lado do servidor.

Apenas seja paciente um pouco mais.

Agora que você viu vários aspectos dos geradores, estamos prontos para dar uma olhada nos bastidores para ver como funcionam os geradores.

6.2.4 Explorando geradores sob o capô

Até agora sabemos que chamar um gerador não o executa. Em vez disso, ele cria um novo iterador que podemos usar para solicitar valores do gerador. Depois que um gerador produz (ou fornece) um valor, ele suspende sua execução e aguarda a próxima solicitação. Então, de certa forma, um gerador funciona quase como um pequeno programa, uma máquina de estado que se move entre estados:

- *Início suspenso* —Quando o gerador é criado, ele inicia neste estado. Nenhum código do gerador é executado.
- *Executando* —O estado em que o código do gerador é executado. A execução continua desde o início ou de onde o gerador foi o último

suspenso. Um gerador se move para este estado quando o iterador correspondente Próximo método é chamado e existe código a ser executado.

- *Rendimento suspenso* —Durante a execução, quando um gerador atinge um produzem expressão, ele cria um novo objeto carregando o valor de retorno, o produz e suspende sua execução. Este é o estado em que o gerador está pausado e está esperando para continuar sua execução.

- *Concluído* —Se durante a execução o gerador funcionar em um Retorna instrução ou fica sem código para executar, o gerador passa para este estado.

A Figura 6.5 ilustra esses estados.

Agora vamos complementar isso em um nível ainda mais profundo, vendo como a execução dos geradores é rastreada com contextos de execução.

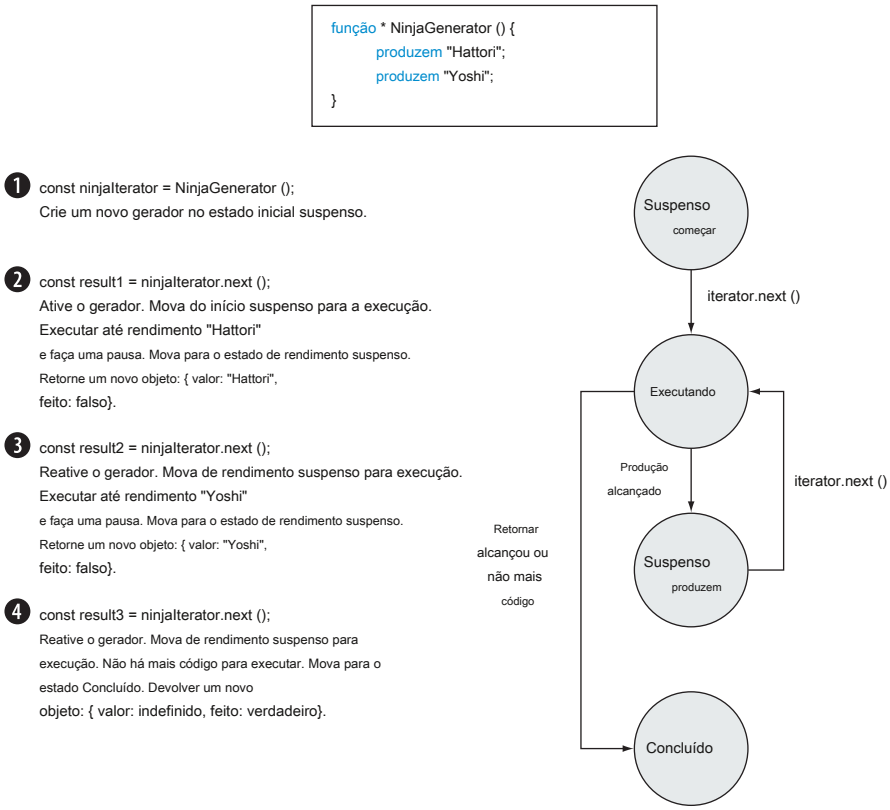


Figura 6.5 Durante a execução, um gerador se move entre os estados acionados por chamadas para o iterador correspondente **Próximo** método.

GERADORES DE RASTREIO COM CONTEXTOS DE EXECUÇÃO

No capítulo anterior, apresentamos o contexto de execução, um mecanismo interno de JavaScript usado para rastrear a execução de funções. Embora um tanto especiais, os geradores ainda são funções, portanto, vamos examinar mais de perto explorando a relação entre eles e os contextos de execução. Começaremos com um fragmento de código simples:

```
function * NinjaGenerator (ação) {
    rendimento "Hattori" + ação; retornar "Yoshi" +
    ação;
}

const ninjaIterator = NinjaGenerator ("skulk"); const result1 = ninjaIterator.next (); const
result2 = ninjaIterator.next ();
```

Aqui, reutilizamos nosso gerador que produz dois valores: Hattori skulk e Yoshi
skulk.

Agora, vamos explorar o estado do aplicativo, a pilha de contexto de execução em vários pontos da execução do aplicativo. A Figura 6.6 fornece um instantâneo em duas posições na execução do aplicativo. O primeiro instantâneo mostra o estado do aplicativo
execução *antes* chamando o NinjaGenerator função B . Porque estamos executando código global, a pilha de contexto de execução contém apenas o contexto de execução global,
que faz referência ao ambiente global no qual nossos identificadores são mantidos. Apenas o
NinjaGenerator identificador faz referência a uma função, enquanto os valores de todos os outros identificadores
fiers são Indefinido.

Quando fazemos a ligação para o NinjaGenerator função C

```
const ninjaIterator = NinjaGenerator ("skulk");
```

o fluxo de controle entra no gerador e, como acontece quando entramos em qualquer outra função, um novo NinjaGenerator O item de contexto de execução é criado (junto com o ambiente léxico correspondente) e colocado na pilha. Mas porque os geradores são especiais, *Nenhum* do código de função é executado. Em vez disso, um novo iterador, ao qual nos referiremos no código como ninjaIterator, é criado e retornado. Como o iterador é usado para controlar a execução do gerador, o iterador obtém uma referência ao contexto de execução no qual foi criado.

Uma coisa interessante acontece quando a execução do programa sai do gerador, como mostra a figura 6.7. Normalmente, quando a execução do programa retorna de uma função padrão, o contexto de execução correspondente é retirado da pilha e completamente descartado. Mas este não é o caso dos geradores.

```
função * NinjaGenerator (ação) {
    produzem "Hattori" + ação;
    Retorna "Yoshi" + ação;
}

const ninjaIterator = NinjaGenerator ("skulk");

const resultado1 = ninjaIterator.next ();
const resultado2 = ninjaIterator.next ();
```

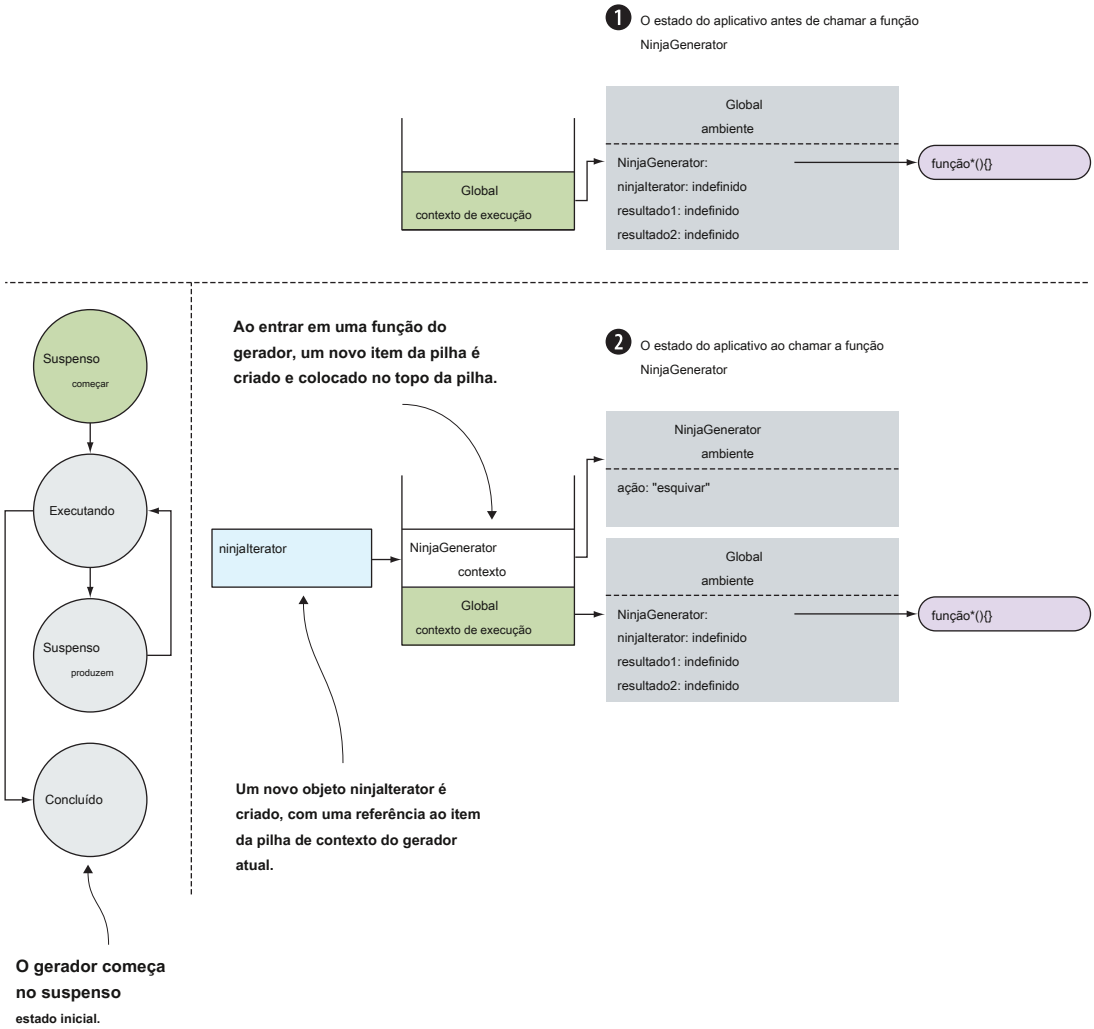


Figura 6.6 O estado da pilha de contexto de execução ***antes*** chamando o **NinjaGenerator** função B , e ***quando*** chamando o **NinjaGenerator** função C

```
função * NinjaGenerator (ação) {
    produzem "Hattori" + ação;
    Retorna "Yoshi" + ação;
}

const ninjaIterator = NinjaGenerator ("skulk");

const resultado1 = ninjaIterator.next ();
const resultado2 = ninjaIterator.next ();
```

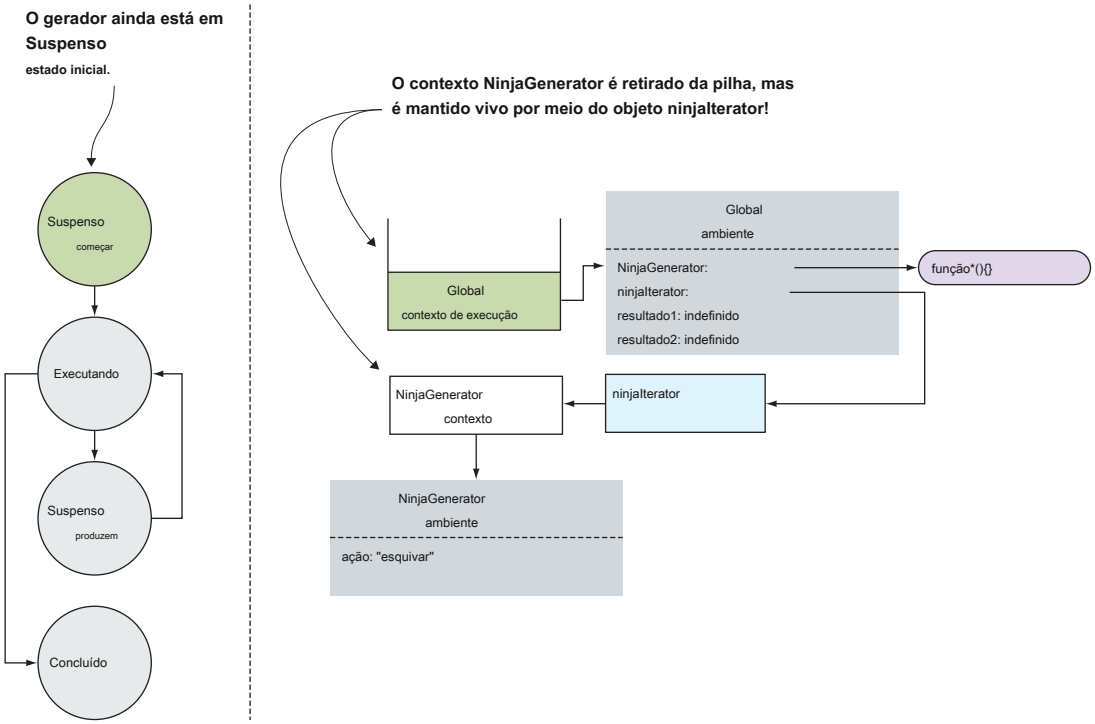


Figura 6.7 O estado do aplicativo ao retornar do **NinjaGenerator** chamar

A correspondência NinjaGenerator item de pilha *é* saiu da pilha, mas é *não* descartado, porque o ninjaIterator mantém uma referência a ele. Você pode vê-lo como um análogo aos fechamentos. Nos encerramentos, precisamos manter vivas as variáveis que estão vivas no momento em que o encerramento da função é criado, para que nossas funções mantenham uma referência ao ambiente no qual foram criadas. Desta forma, garantimos que o ambiente e suas variáveis estão vivos enquanto a própria função. Os geradores, por outro lado, devem ser capazes de retomar sua execução. Como a execução de todas as funções é tratada por contextos de execução, o iterador mantém uma referência ao seu contexto de execução, de modo que esteja vivo pelo tempo que o iterador precisar.

Outra coisa interessante acontece quando chamamos o Próximo método no iterador:

```
const result1 = ninjaIterator.next ();
```

Se esta fosse uma chamada de função direta padrão, isso causaria a criação de um *novo* Próximo() item de contexto de execução, que seria colocado na pilha. Mas como você deve ter notado, os geradores são tudo menos padrão, e uma chamada para o Próximo método de um iterador se comporta de maneira muito diferente. Ele reativa o contexto de execução correspondente, neste caso, o NinjaGenerator contexto e o coloca no topo da pilha, continuando a execução de onde parou, conforme mostrado na figura 6.8.

A Figura 6.8 ilustra uma diferença crucial entre funções padrão e geradores. As funções padrão só podem ser chamadas de novo, e cada chamada cria um *novo* contexto de execução. Em contraste, o contexto de execução de um gerador pode ser temporariamente suspenso e reiniciado à vontade.

Em nosso exemplo, porque esta é a primeira chamada para o Próximo método, e o gerador não começou a executar, o gerador inicia sua execução e passa para o estado Executando. A próxima coisa interessante acontece quando nossa função de gerador atinge este ponto:

rendimento "Hattori" + ação

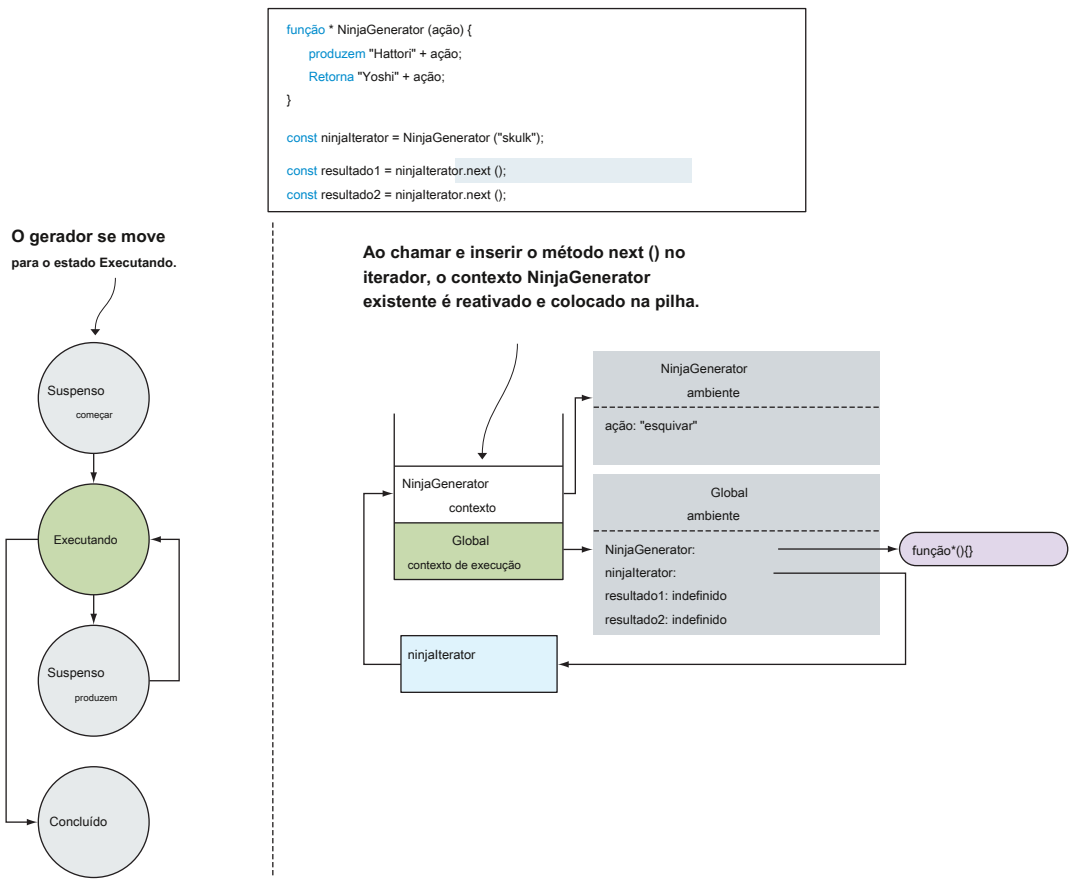


Figura 6.8 Chamando o iterador **Próximo** método reativa o item da pilha de contexto de execução do gerador correspondente, coloca-o na pilha e continua de onde parou da última vez.

O gerador determina que a expressão é igual Hattori skulk, e a avaliação chega ao produzem palavra-chave. Isso significa que Hattori skulk é o primeiro resultado intermediário do nosso gerador e que queremos suspender a execução do gerador e retornar esse valor. Em termos de estado do aplicativo, algo semelhante acontece como antes: o NinjaGenerator contexto é retirado da pilha, mas não é completamente descartado, porque ninjaIterator mantém uma referência a ele. O gerador agora está suspenso e mudou para o estado Suspended Yield, sem bloqueio. A execução do programa é retomada em código global, armazenando o valor gerado em resultado1.

O estado atual do aplicativo é mostrado na figura 6.9.

O código continua alcançando outra chamada de iterador:

```
const result2 = ninjaIterator.next ();
```

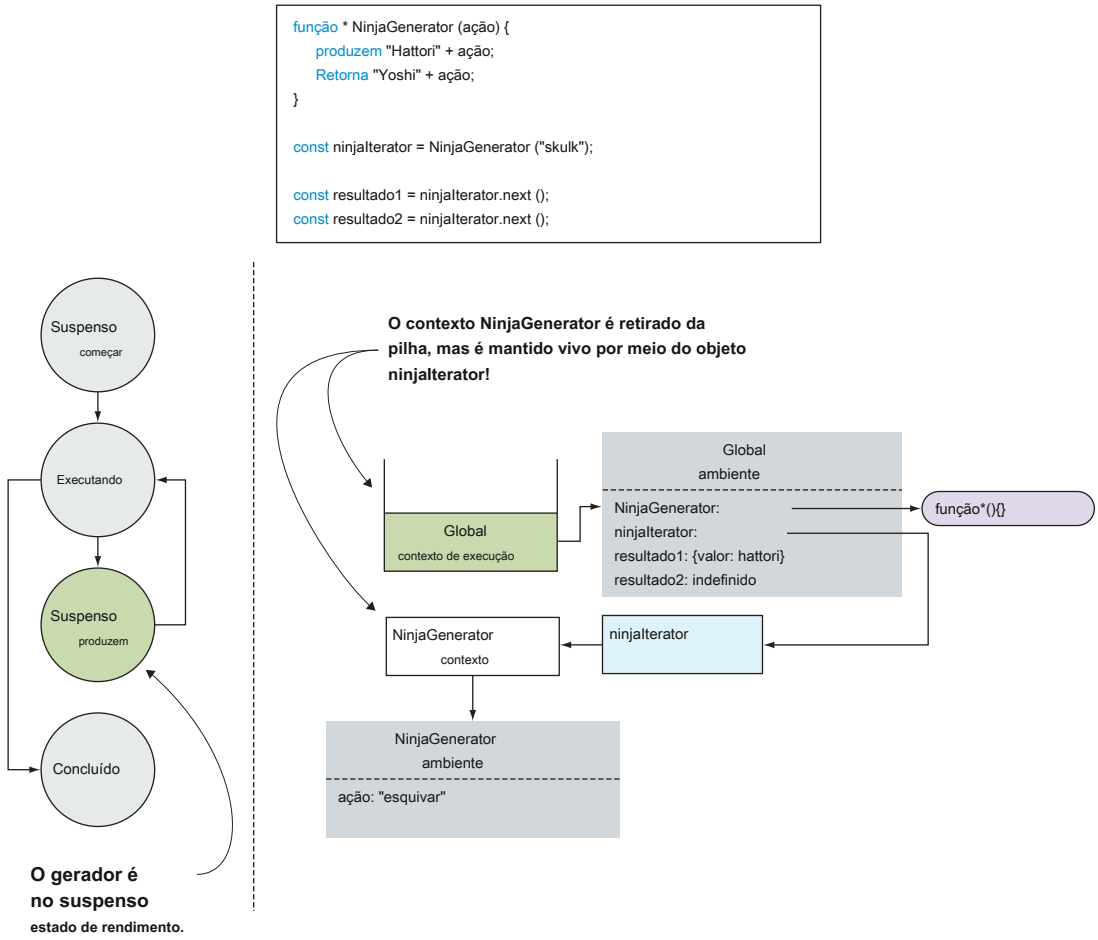


Figura 6.9 Depois de gerar um valor, o contexto de execução do gerador é retirado da pilha (mas não é descartado, porque **ninjalterator** mantém uma referência a ele), e a execução do gerador é suspensa (o gerador passa para o estado de rendimento suspenso).

Neste ponto, passamos por todo o procedimento mais uma vez: reativamos o NinjaGenerator contexto referenciado por ninjaIterator, coloque-o na pilha e continue a execução de onde paramos. Neste caso, o gerador avalia a expressão
"Yoshi" + ação. Mas desta vez não há produzem expressão e, em vez disso, o programa encontra uma Retorna demonstração. Isso retorna o valor Yoshi skulk e conclui a execução do gerador movendo o gerador para o estado Concluído.

Uff, isso era incrível! Nós nos aprofundamos em como os geradores funcionam sob o capô para mostrar que todos os benefícios maravilhosos dos geradores são um efeito colateral do fato de que o contexto de execução de um gerador é mantido vivo se produzirmos de um gerador, e não destruído como é o caso com valores de retorno e funções padrão.

Agora, recomendamos que você respire fundo antes de continuar com o segundo ingrediente-chave necessário para escrever um código assíncrono elegante: promessas.

6,3 Trabalhando com promessas

Em JavaScript, dependemos muito de cálculos assíncronos, cálculos cujos resultados ainda não temos, mas teremos em algum momento posterior. Portanto, o ES6 introduziu um novo conceito que torna o manuseio de tarefas assíncronas mais fácil: promessas.

UMA *promessa* é um espaço reservado para um valor que não temos agora, mas teremos mais tarde; é uma garantia de que, eventualmente, saberemos o resultado de uma computação assíncrona. Se cumprirmos nossa promessa, nosso resultado será um valor. Se ocorrer um problema, nosso resultado será um erro, uma desculpa para não podermos entregar. Um ótimo exemplo de uso de promessas é buscar dados de um servidor; prometemos que eventualmente obteremos os dados, mas sempre há uma chance de ocorrerem problemas.

Criar uma nova promessa é fácil, como você pode ver no exemplo a seguir.

Listagem 6.10 Criar uma promessa simples

Uma promessa é resolvido com sucesso chamando o resolução aprovada função (e rejeitado por ligar a função de rejeição).

Cria uma promessa chamando um construtor Promise integrado e passando uma função de retorno de chamada com dois parâmetros: resolver e rejeitar

```
const ninjaPromise = new Promise ((resolver, rejeitar) => {
    resolver ("Hattori");
    // rejeitar ("Um erro ao resolver uma promessa!"); });
```

```
ninjaPromise.then (ninja => {
    assert (ninja === "Hattori", "Foi-nos prometido Hattori!"); }, err => {

    fail ("Não deve haver um erro")});
```

E o segundo é chamado se um erro ocorre.

Ao usar o método then em uma promessa, podemos passar em dois retornos de chamada; o primeiro é chamado se um promessa foi resolvida com sucesso.

Para criar uma promessa, usamos o novo e integrado Promessa construtor, para o qual passamos uma função, neste caso uma função de seta (mas poderíamos facilmente usar uma expressão de função). Esta função, chamada de *executor* função, tem dois parâmetros:
resolver e rejeitar. O executor é chamado *imediatamente* ao construir o
Promessa objeto com duas funções integradas como argumentos: resolver, que chamamos manualmente se quisermos que a promessa seja resolvida com sucesso, e rejeitar, que chamamos se ocorrer um erro.

Este código usa a promessa chamando o integrado então método no Promessa
objeto, um método para o qual passamos duas funções de retorno de chamada: a *sucesso* retorno de chamada e um *fracasso* ligar de volta. O primeiro é chamado se a promessa for resolvida com sucesso (se o resolver
função é chamada na promessa), e o último é chamado se houver um problema (ocorre uma exceção não tratada ou o rejeitar função é chamada em uma promessa).

Em nosso código de exemplo, criamos uma promessa e a resolvemos imediatamente chamando o
resolver função com o argumento Hattori. Portanto, quando chamamos o então
método, o primeiro, sucesso, retorno de chamada é executado e o teste que produz fomos
prometeu Hattori! passes.

Agora que temos uma ideia geral do que são as promessas e como funcionam, vamos dar um passo atrás para ver alguns dos problemas que as promessas enfrentam.

6.3.1 Compreender os problemas com retornos de chamada simples

Usamos código assíncrono porque não queremos bloquear a execução de nosso aplicativo (decepcionando assim nossos usuários) durante a execução de tarefas de longa duração. Atualmente, resolvemos este problema com retornos de chamada: Para uma tarefa de longa execução, fornecemos uma função, um retorno de chamada que é invocado quando a tarefa finalmente é concluída.

Por exemplo, buscar um arquivo JSON em um servidor é uma tarefa de longa duração, durante a qual não queremos que o aplicativo pare de responder para nossos usuários. Portanto, fornecemos um retorno de chamada que será invocado quando a tarefa for concluída:

```
getJSON ("data / ninjas.json", function () {
    / * Manipular resultados * /
});
```

Naturalmente, durante essa tarefa de longa execução, podem ocorrer erros. E o problema com retornos de chamada é que você não pode usar construções de linguagem integradas, como tentar pegar declarações, da seguinte forma:

```
tentar {
    getJSON ("data / ninjas.json", function () {
        // Manipular resultados
    });
} catch (e) {/ * Manipular erros * /}
```

Isso acontece porque o código que invoca o retorno de chamada geralmente não é executado na mesma etapa do loop de evento que o código que inicia a tarefa de longa execução (você verá exatamente o que isso significa quando aprender mais sobre o loop de evento no capítulo 13).

Como consequência, os erros geralmente se perdem. Muitas bibliotecas, portanto, definem suas próprias convenções para relatar erros. Por exemplo, no mundo Node.js, callbacks costumam ter dois argumentos, errar e dados, Onde errar será um valor não nulo se ocorrer um erro em algum lugar ao longo do caminho. Isso leva ao primeiro problema com retornos de chamada: *difícil tratamento de erros.*

Depois de realizar uma tarefa de longa duração, geralmente queremos fazer algo com os dados obtidos. Isso pode levar ao início de outra tarefa de longa execução, que pode eventualmente acionar outra tarefa de longa duração e assim por diante - levando a uma série de etapas interdependentes, assíncronas e processadas por callback. Por exemplo, se quisermos executar um plano sorrateiro para encontrar todos os ninjas à nossa disposição, obter a localização do primeiro ninja e enviar-lhe algumas ordens, acabaremos com algo assim:

```
getJSON ("data / ninjas.json", function (err, ninjas) {
    getJSON (ninjas [0] .location, function (err, locationInfo) {
        sendOrder (locationInfo, function (err, status) {
          / * Status do processo * /
        })
    })
});
```

Você provavelmente acabou, pelo menos uma ou duas vezes, com código estruturado de forma semelhante - um monte de callbacks aninhados que representam uma série de etapas que precisam ser feitas. Você pode notar que esse código é difícil de entender, inserir novas etapas é uma dor e o tratamento de erros complica seu código significativamente. Você obtém essa “pirâmide da desgraça” que continua crescendo e é difícil de administrar. Isso nos leva ao segundo problema com retornos de chamada: *executar sequências de etapas é complicado.*

Às vezes, as etapas que temos que percorrer para chegar ao resultado final não dependem umas das outras, então não temos que executá-las em sequência. Em vez disso, para economizar milissegundos preciosos, podemos fazê-los em paralelo. Por exemplo, se quisermos colocar um plano em movimento que exija que saibamos quais ninjas temos à nossa disposição, o próprio plano e o local onde nosso plano será executado, poderíamos aproveitar as vantagens do jQuery obter método e escreva algo assim:

```
var ninjas, mapInfo, plano;

$ .get ("data / ninjas.json", function (err, data) {
    if (err) {processError (err); Retorna; } ninjas = dados;

    actionItemArrived ();
});

$ .get ("data / mapInfo.json", function (err, data) {
    if (err) {processError (err); Retorna; } mapInfo = dados;

    actionItemArrived ();
});
```

```
$ .get ("plan.json", função (err, dados) {
    if (err) {processError (err); Retorna; }

    plano = dados;
    actionItemArrived ();
});

function actionItemArrived () {
    if (ninjas! = null && mapInfo! = null && plan! = null) {
        console.log ("O plano está pronto para ser colocado em ação!");
    }
}

function processError (err) {
    alert ("Erro", err)
}
```

Neste código, executamos as ações de obter os ninjas, obter as informações do mapa e obter o plano em paralelo, porque essas ações não dependem umas das outras. Só nos preocupamos que, no final, tenhamos todos os dados à nossa disposição. Como não sabemos a ordem em que os dados são recebidos, sempre que obtemos alguns dados, temos que verificar se é a última peça do quebra-cabeça que está faltando. Finalmente, quando todas as peças estiverem no lugar, podemos colocar nosso plano em ação. Observe que temos que escrever muito código clichê apenas para fazer algo tão comum quanto executar uma série de ações em paralelo. Isso nos leva ao terceiro problema com retornos de chamada: *executar uma série de etapas em paralelo também é complicado.*

Ao apresentar o primeiro problema com retornos de chamada - lidar com erros - mostramos como não podemos usar algumas das construções fundamentais da linguagem, como trycatch declarações. Algo semelhante acontece com os loops: se você quiser realizar ações assíncronas para cada item em uma coleção, terá que passar por mais alguns obstáculos para fazê-lo.

É verdade que você pode fazer uma biblioteca para simplificar o tratamento de todos esses problemas (e muitas pessoas fizeram isso). Mas isso geralmente leva a muitas maneiras ligeiramente diferentes de lidar com os mesmos problemas, então as pessoas por trás do JavaScript nos deram *promessas,* uma abordagem padrão para lidar com computação assíncrona.

Agora que você entende a maioria das razões por trás da introdução de promessas, bem como tem um conhecimento básico delas, vamos dar um passo adiante.

6.3.2 Mergulhando em promessas

Uma promessa é um objeto que serve como um marcador para um resultado de uma tarefa assíncrona. Representa um valor que não temos, mas esperamos ter no futuro. Por essa razão, durante sua vida, uma promessa pode passar por alguns estados, como mostra a figura 6.10.

Uma promessa começa no *pendente* estado, no qual não sabemos nada sobre nosso valor prometido. É por isso que uma promessa no estado pendente também é chamada de não resolvido promessa. Durante a execução do programa, se a promessa resolver função é chamada, o

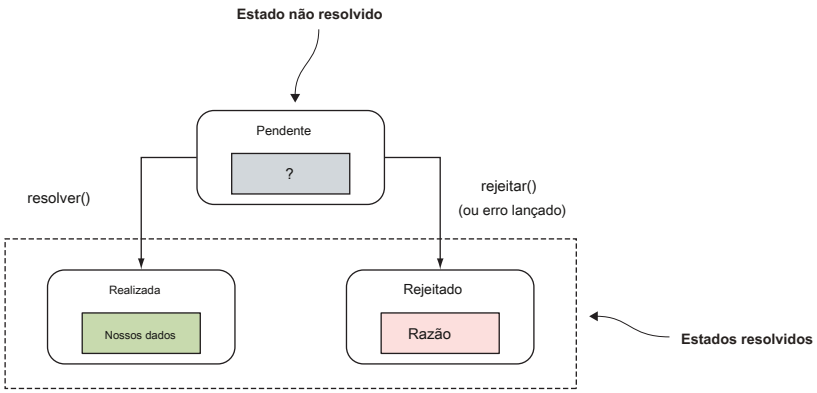


Figura 6.10 Estados de uma promessa

promessa move-se para o *realizada* estado, no qual obtivemos com sucesso o valor prometido. Por outro lado, se a promessa é rejeitar função é chamada, ou se uma exceção não tratada ocorrer durante o tratamento da promessa, a promessa muda para o *rejeitado* estado, no qual não conseguimos obter o valor prometido, mas no qual ao menos sabemos o porquê. Uma vez que uma promessa tenha alcançado o *realizada* estado ou o

rejeitado estado, ele não pode mudar (uma promessa não pode ir de cumprida para rejeitada ou vice-versa) e sempre permanece nesse estado. Dizemos que uma promessa é *resolvido (* com sucesso ou não).

A lista a seguir fornece uma visão mais detalhada do que está acontecendo quando usamos as promessas.

Listagem 6.11 Uma análise mais detalhada da ordem de execução da promessa

```
relatório ("No início do código");

var ninjaDelayedPromise = new Promise ((resolver, rejeitar) => {
    report ("executor ninjaDelayedPromise");
    setTimeout (() => {
        report ("Resolvendo ninjaDelayedPromise");
        resolver ("Hattori");
    }, 500);
});

assert (ninjaDelayedPromise! == null, "Depois de criar ninjaDelayedPromise");

ninjaDelayedPromise.then (ninja => {
    afirmar (ninja === "Hattori",
                "resolução de ninjaDelayedPromise tratada com Hattori");
});
```

Chamando a Promessa construtor imediatamente invoca o passado função.

Resolveremos essa promessa como bem-sucedida depois que um tempo limite de 500 ms expirar.

O método Promise then é usado para configurar um retorno de chamada que será chamado quando a promessa for resolvida, em nosso caso quando o tempo limite expirar.

```
const ninjaImmediatePromise = new Promise ((resolver, rejeitar) => {
    report ("executor ninjaImmediatePromise. Resolução imediata."); resolver ("Yoshi");

});

ninjaImmediatePromise.then (ninja => {
    afirmar (ninja === "Yoshi",
                "resolução de ninjaImmediatePromise tratada com Yoshi");
});

relatório ("No final do código");
```

Cria um novo prometa que Obtém imediatamente resolvido

Configura um retorno de chamada a ser invocado quando a promessa resolve. Mas nossa promessa já foi cumprida!

O código na listagem 6.11 produz os resultados mostrados na figura 6.11. Como você pode ver, o código começa registrando a mensagem "No início do código" usando nosso personalizado relatório função (apêndice C) que exibe a mensagem na tela. Isso nos permite rastrear facilmente a ordem de execução.

Em seguida, criamos uma nova promessa chamando o Promessa construtor. Isso invoca imediatamente a função do executor na qual configuramos um tempo limite:

```
setTimeout (() => {
    report ("Resolvendo ninjaDelayedPromise");
    resolver ("Hattori");
}, 500);
```

O tempo limite resolverá a promessa a seguir 500ms. Esta poderia ter sido qualquer outra tarefa assíncrona, mas escolhemos o humilde timeout devido à sua simplicidade.

Depois de ninjaDelayedPromise tem sido criado, ele ainda não sabe o valor que eventualmente terá, ou mesmo se terá sucesso. (Lembre-se de que ainda está esperando o tempo limite que o resolverá.) Assim, após a construção ção, o ninjaDelayedPromise está no primeiro estado de promessa, *pendente*.

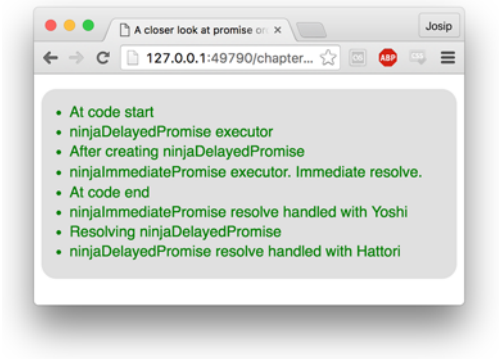


Figura 6.11 O resultado da execução da listagem 6.11

Em seguida, usamos o então método no ninjaDelayedPromise para agendar um callback a ser executado quando a promessa for resolvida com sucesso:

```
ninjaDelayedPromise.then (ninja => {
    afirmar (ninja === "Hattori",
                "resolução de ninjaDelayedPromise tratada com Hattori");
});
```

Este retorno de chamada irá *sempre* ser chamado de forma assíncrona, independentemente do estado atual da promessa.

Continuamos criando outra promessa, ninjaImmediatePromise, que é resolvido imediatamente durante sua construção, chamando o resolver função. Ao contrário do ninjaDelayedPromise, que após a construção está no *pendente* estado, o ninjaImmediatePromise termina a construção no *resolvido* estado, e a promessa já tem o valor Yoshi.

Depois, usamos o ninjaImmediatePromise de então método para registrar um retorno de chamada que será executado quando a promessa for resolvida com êxito. Mas nossa promessa já foi cumprida; isso significa que o retorno de chamada de sucesso será chamado imediatamente ou que será ignorado? A resposta é *nenhum.*

As promessas são projetadas para lidar com ações assíncronas, portanto, o mecanismo de JavaScript *sempre* recorre ao tratamento assíncrono para tornar previsível o comportamento da promessa. O motor faz isso executando o então callbacks depois que todo o código na etapa atual do loop de eventos for executado (mais uma vez, exploraremos exatamente o que isso significa no capítulo 13). Por este motivo, se estudarmos a saída na figura 6.11, veremos que primeiro registramos "No final do código" e, em seguida, registramos que o ninjaImmediatePromise foi resolvido. No final, após o tempo limite de 500 ms expirar, o ninjaDelayedPromise é resolvido, o que causa a execução da correspondência então ligar de volta.

Neste exemplo, por uma questão de simplicidade, trabalhamos apenas com o cenário otimista em que tudo vai bem. Mas o mundo real não é só raios de sol e arco-íris, então vamos ver como lidar com todos os tipos de problemas malucos que podem ocorrer.

6.3.3 *Rejeitando promessas*

Existem duas maneiras de rejeitar uma promessa: *explicitamente,* chamando o aprovado rejeitar método na função executora de uma promessa, e *implicitamente,* se durante o tratamento de uma promessa, ocorre uma exceção não tratada. Vamos começar nossa exploração com a seguinte lista.

Listagem 6.12 Rejeitando promessas explicitamente

```
promessa const = nova promessa ((resolver, rejeitar) => {
    rejeitar ("Rejeitar explicitamente uma promessa!");
});

promessa.então (
    () => falha ("Caminho feliz, não será chamado!"),
    error => pass ("Uma promessa foi rejeitada explicitamente!")
);
```

Uma promessa pode ser rejeitada explicitamente chamando a função de rejeição passada.

Se uma promessa for rejeitada, a segunda, erro, o retorno de chamada é invocado.

Podemos rejeitar explicitamente uma promessa, chamando o aprovado rejeitar método: rejeitar ("Rejeitar explicitamente uma promessa!"). Se uma promessa for rejeitada, ao registrar retornos de chamada através do então método, o segundo, erro, retorno de chamada sempre será invocado.

Além disso, podemos usar uma sintaxe alternativa para lidar com rejeições de promessa, usando o pegar método, conforme mostrado na lista a seguir.

Listagem 6.13 Encadeando um **pegar** método

```
var promessa = nova promessa ((resolver, rejeitar) => {
    rejeitar ("Rejeitar explicitamente uma promessa!"); });


promessa.then (() => falha ("Caminho feliz, não será chamado!"))
            . pegar (() => passar ("A promessa também foi rejeitada"));
```

Em vez de fornecer o segundo, erro, retorno de chamada, podemos encadear o método catch e passar para ele o callback de erro. O resultado final é o mesmo.

Como mostra a listagem 6.13, podemos encadear no pegar método após o então , para também fornecer um retorno de chamada de erro que será invocado quando uma promessa for rejeitada. Neste exemplo, é uma questão de estilo pessoal. Ambas as opções funcionam igualmente bem, mas mais tarde, ao trabalhar com cadeias de promessas, veremos um exemplo em que encadear o
pegar método é útil.

Além da rejeição explícita (por meio do rejeitar chamada), uma promessa também pode ser rejeitada implicitamente, se ocorrer uma exceção durante seu processamento. Dê uma olhada no exemplo a seguir.

Listagem 6.14 Exceções rejeitam implicitamente uma promessa

```
promessa const = nova promessa ((resolver, rejeitar) => {
    undeclaredVariable ++;
});

promessa.then (() => falha ("Caminho feliz, não será chamado!"))
            . catch (error => pass ("A terceira promessa também foi rejeitada"));
```

Uma promessa é rejeitada implicitamente se uma exceção não tratada ocorrer quando processamento da promessa.

Se ocorrer uma exceção, o segundo, erro, o retorno de chamada é invocado.

Dentro do corpo do executor da promessa, tentamos incrementar variável não declarada, uma variável que não está definida em nosso programa. Como esperado, isso resulta em uma exceção. Porque não há tentar pegar declaração dentro do corpo do executor, isso resulta em uma rejeição implícita da promessa atual, e o pegar o retorno de chamada é eventualmente invocado. Nesta situação, poderíamos ter facilmente fornecido o segundo retorno de chamada para o então método, e o efeito final seria o mesmo.

Essa maneira de tratar todos os problemas que acontecem ao trabalhar com as promessas de maneira uniforme é extremamente útil. Independentemente de como a promessa foi rejeitada, seja explicitamente chamando o rejeitar método ou mesmo implicitamente, se ocorrer uma exceção, todos os erros e motivos de rejeição são direcionados ao nosso callback de rejeição. Isso torna nossas vidas como desenvolvedores um pouco mais fáceis.

Agora que entendemos como as promessas funcionam e como agendar retornos de chamada de sucesso e falha, vamos pegar um cenário do mundo real, obtendo dados formatados em JSON de um servidor, e "prometê-los".

6.3.4 Criando nossa primeira promessa do mundo real

Uma das ações assíncronas mais comuns no cliente é buscar dados do servidor. Como tal, este é um excelente pequeno estudo de caso sobre o uso de promessas. Para a implementação subjacente, usaremos o integrado XMLHttpRequest objeto.

Listagem 6.15 Criando um **getJSON** promessa

Cria um XMLHttpRequest objeto

Cria e retorna uma nova promessa

Registra um carregamento manipulador que será chamado se o servidor respondeu

Inicializa o pedido

Mesmo se o servidor tiver respondeu, isso não significa que tudo correu como esperado. Use o resultado apenas se o servidor responder com status 200 (tudo OK).

Tente analisar a string JSON; se for bem-sucedido, resolva a promessa como bem-sucedida com o objeto analisado.

Se o servidor responder com um código de status diferente ou se houver uma exceção ao analisar o String JSON, rejeite a promessa.

Envia o solicitação

Se houver um erro durante a comunicação com o servidor, rejeite a promessa.

Usa a promessa criada pela função getJSON para registrar resolver e rejeitar callbacks

```
function getJSON (url) {
    retornar nova promessa ((resolver, rejeitar) => {
        solicitação const = novo XMLHttpRequest ();

        request.open ("GET", url);

        request.onload = function () {
            tentar {
                if (this.status === 200) {
                    resolver (JSON.parse (this.response));
                } outro{
                        rejeitar (this.status + "" + this.statusText);
                }
            } catch (e) {
                rejeitar (mensagem eletrônica);
            }
        };

        request.onerror = function () {
            rejeitar (this.status + "" + this.statusText);
        };

        request.send ();
    });
}

getJSON ("data / ninjas.json"). então (ninjas => {
    assert (ninjas! == null, "Ninjas obtidos!"); }). catch (e => fail ("Não deveria estar aqui:"
+ e));
```

NOTA Executar este exemplo, e todos os exemplos subsequentes que reutilizam esta função, requer um servidor em execução. Você pode, por exemplo, usar www.npmjs.com/ package / http-server .

Nosso objetivo é criar um getJSON função que retorna uma promessa que nos permitirá registrar retornos de chamada de sucesso e falha para obter dados formatados em JSON de maneira assíncrona do servidor. Para a implementação subjacente, usamos o integrado XMLHttpRequest
objeto que oferece dois eventos: carregando e onerror. O carregando evento é acionado quando o navegador recebe uma resposta do servidor, e onerror é acionado quando ocorre um erro na comunicação. Esses manipuladores de eventos serão chamados de forma assíncrona pelo navegador, à medida que ocorrem.

Se ocorrer um erro na comunicação, definitivamente não conseguiremos obter nossos dados do servidor, portanto, a coisa mais honesta a fazer é rejeitar nossa promessa:

```
request.onerror = function () {
    rejeitar (this.status + "" + this.statusText);
};
```

Se recebermos uma resposta do servidor, temos que analisar essa resposta e considerar a situação exata. Sem entrar em muitos detalhes, um servidor pode responder com várias coisas, mas neste caso, nos preocupamos apenas se a resposta foi bem-sucedida (status 200). Se não for, rejeitamos novamente a promessa.

Mesmo que o servidor tenha respondido com sucesso com dados, isso ainda não significa que estamos livres. Como nosso objetivo era obter objetos formatados em JSON do servidor, o código JSON sempre poderia ter erros de sintaxe. É por isso que, ao chamar o
JSON.parse método, envolvemos o código com um tentar pegar demonstração. Se ocorrer uma exceção ao analisar a resposta do servidor, também rejeitaremos a promessa. Com isso, cuidamos de todos os cenários ruins que podem acontecer.

Se tudo correr conforme o planejado, e obtivermos nossos objetos com sucesso, podemos cumprir a promessa com segurança. Finalmente, podemos usar nosso getJSON função para buscar ninjas do servidor:

```
getJSON ("data / ninjas.json"). então (ninjas => {
    assert (ninjas! == null, "Ninjas obtidos!"); }). catch (e => fail ("Não deveria estar aqui:"
+ e));
```

Nesse caso, temos três fontes potenciais de erros: erros no estabelecimento da comunicação entre o servidor e o cliente, o servidor respondendo com dados imprevistos (status de resposta inválido) e código JSON inválido. Mas da perspectiva do código que usa o getJSON função, não nos importamos com as especificações das fontes de erro. Nós fornecemos apenas um retorno de chamada que é acionado se tudo correr bem e os dados são recebidos corretamente, e um retorno de chamada que é acionado se ocorrer algum erro. Isso torna nossas vidas como desenvolvedores muito mais fáceis.

Agora vamos subir um degrau e explorar outra grande vantagem das promessas: sua composição elegante. Começaremos encadeando várias promessas em uma série de etapas distintas.

6.3.5 Encadeando promessas

Você já viu como lidar com uma sequência de etapas interdependentes leva à pirâmide da desgraça, uma sequência de callbacks profundamente aninhada e difícil de manter. As promessas são um passo para resolver esse problema, porque têm a capacidade de serem acorrentadas.

No início do capítulo, você viu como, usando o então método em uma promessa, podemos registrar um retorno de chamada que será executado se uma promessa for resolvida com sucesso. O que não dissemos é que chamar o então método também retorna uma nova promessa. Então

não há nada que nos impeça de acorrentar tantos então métodos como quisermos; veja o código a seguir.

Listagem 6.16 Encadeando promessas com **então**

```
getJSON ("data / ninjas.json")
  . então (ninjas => getJSON (ninjas [0] .missionsUrl))
  . então (missões => getJSON (missões [0] .detailsUrl))
  . então (missão => afirmar (missão! == null, "Missão Ninja obtida!"))
  . catch (erro => falha ("Ocorreu um erro"));
```

Capturas prometem rejeições em qualquer uma das etapas

Especifica vários etapas sequenciais por encadeando e depois liga

Isso cria uma sequência de promessas que serão, se tudo correr conforme o planejado, resolvidas uma após a outra. Primeiro, usamos o getJSON ("data / ninjas.json") método para obter uma lista de ninjas do arquivo no servidor. Após recebermos essa lista, pegamos as informações sobre o primeiro ninja e solicitamos uma lista de missões para as quais o ninja está atribuído:
getJSON (ninjas [0] .missionsUrl). Mais tarde, quando essas missões chegam, fazemos mais um pedido para os detalhes da primeira missão: getJSON (missões [0] .detailsUrl). Finalmente, registramos os detalhes da missão.

Escrever esse código usando callbacks padrão resultaria em uma sequência profundamente aninhada de callbacks. Identificar a sequência exata de etapas não seria fácil, e Deus nos livre se decidirmos adicionar uma etapa extra em algum lugar no meio.

PEGANDO ERROS EM PROMESSAS EM CADEIA

Ao lidar com sequências de etapas assíncronas, um erro pode ocorrer em qualquer etapa. Já sabemos que podemos fornecer um segundo retorno de chamada de erro para o então ligar ou fazer uma corrente em um pegar chamada que leva um retorno de chamada de erro. Quando nos preocupamos apenas com o sucesso / falha de toda a sequência de etapas, pode ser entediante fornecer a cada etapa um tratamento especial de erros. Portanto, conforme mostrado na listagem 6.16, podemos tirar proveito do

pegar método que você viu antes:

```
. . . catch (error => fail ("Ocorreu um erro:" + err));
```

Se ocorrer uma falha em qualquer uma das promessas anteriores, o pegar método pega isso. Se nenhum erro ocorrer, o fluxo do programa continuará por ele, sem obstruções.

Lidar com uma sequência de etapas é muito mais agradável com promessas do que com retornos de chamada regulares, você não concorda? Mas ainda não é tão elegante quanto poderia ser. Chegaremos a isso em breve, mas primeiro vamos ver como usar promessas para cuidar das etapas assíncronas paralelas.

6.3.6 *Esperando por uma série de promessas*

Além de nos ajudar a lidar com sequências de etapas assíncronas interdependentes, as promessas reduzem significativamente a carga de espera por várias tarefas assíncronas independentes. Vamos revisitar nosso exemplo no qual queremos, em paralelo, reunir informações sobre os ninjas à nossa disposição, os meandros do plano e o mapa do

local onde o plano será colocado em movimento. Com promessas, isso é tão simples quanto mostrado na lista a seguir.

Listagem 6.17 Esperando por uma série de promessas com **Promise.all**

```
Promise.all ([ getJSON ("data / ninjas.json"),
               getJSON ("data / mapInfo.json"),
               getJSON ("data / plan.json")]). então (resultados => {
  const ninjas = resultados [0], mapInfo = resultados [1], plano = resultados [2];

  afirmar (ninjas! == undefined
&& mapInfo! == undefined && plan! == undefined,
        "O plano está pronto para ser posto em prática!");
}). catch (erro => {
  falha ("Um problema em realizar nosso plano!"); });
```

O resultado é uma série de sucessos valores, no ordem de aprovação em promessas.

e cria uma nova promessa

O método Promise.all leva uma série de promessas, que tem sucesso se tudo as promessas são bem-sucedidas e fracassam se uma única promessa falhar.

Como você pode ver, não precisamos nos preocupar com a ordem em que as tarefas são executadas e se algumas delas foram concluídas e outras não. Afirmamos que queremos esperar por uma série de promessas usando o integrado Promise.all método. Este método aceita uma série de promessas e cria um *novo* promessa que é resolvida com sucesso quando todas as promessas passadas são resolvidas, e rejeita se até mesmo uma das promessas falhar. O retorno de chamada de sucesso recebe uma matriz de valores de sucesso, um para cada uma das promessas passadas, em ordem. Reserve um minuto para apreciar a elegância do código que processa várias tarefas assíncronas paralelas com promessas.

O Promise.all método espera por todas as promessas em uma lista. Mas às vezes temos inúmeras promessas, mas nos preocupamos apenas com a primeira que é bem-sucedida (ou falha). Conheçer a Promise.race método.

PROMESSAS DE CORRIDA

Imagine que temos um grupo de ninjas à nossa disposição e que queremos dar uma missão ao primeiro ninja que atender à nossa chamada. Ao lidar com promessas, podemos escrever algo como a lista a seguir.

Listagem 6.18 Promessas de corrida com **Promise.race**

```
Promise.race ([ getJSON ("data / yoshi.json"),
                getJSON ("data / hattori.json"),
                getJSON ("data / hanzo.json")])
  . então (ninja => {
    assert (ninja! == null, ninja.name + "respondeu primeiro"); }). catch (error => fail ("Failure!"));
```

É simples assim. Não há necessidade de rastrear tudo manualmente. Nós usamos o Promise.race método para pegar uma série de promessas e retornar um método completo *novo* promessa que resolve ou rejeita assim que a primeira das promessas resolve ou rejeita.

Até agora, você viu como as promessas funcionam e como podemos usá-las para simplificar muito o tratamento de uma série de etapas assíncronas, em série ou em paralelo. Embora

as melhorias, quando comparadas aos retornos de chamada antigos em termos de tratamento de erros e elegância do código, são ótimas, o código prometido ainda não está no mesmo nível de elegância que o código síncrono simples. Na próxima seção, os dois grandes conceitos que introduzimos neste capítulo, *geradores* e *promessas,* vêm juntos para fornecer a simplicidade do código síncrono com a natureza não bloqueadora do código assíncrono.

6,4 Combinando geradores e promessas

Nesta seção, combinaremos geradores (e sua capacidade de pausar e retomar sua execução) com promessas, a fim de obter um código assíncrono mais elegante. Usaremos o exemplo de uma funcionalidade que permite aos usuários obter detalhes da missão de alto nível realizada pelo ninja mais popular. Os dados que representam os ninjas, os resumos de suas missões, bem como os detalhes das missões são armazenados em um servidor remoto, codificado em JSON.

Todas essas subtarefas são de longa duração e mutuamente dependentes. Se tivéssemos que implementá-los de maneira síncrona, obteríamos o seguinte código simples:

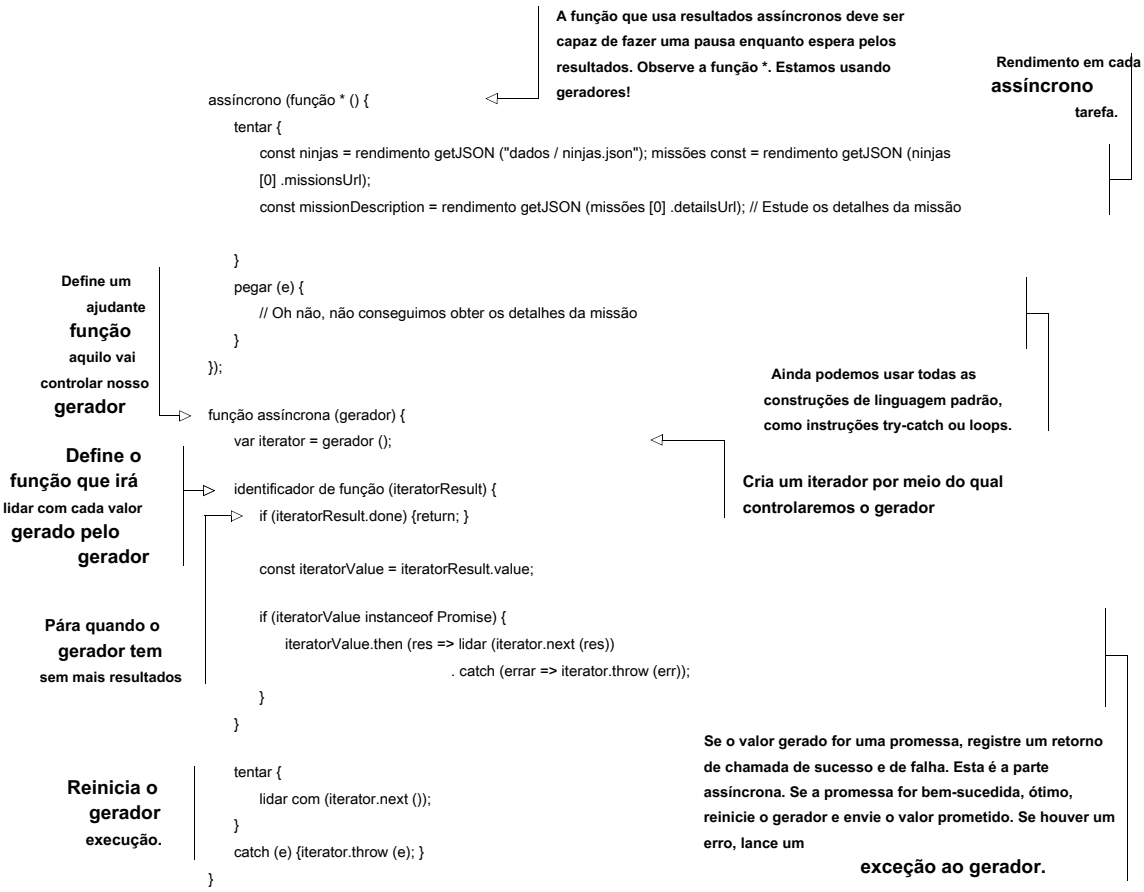
```
tentar {
    const ninjas = syncGetJSON ("data / ninjas.json"); missões const = syncGetJSON (ninjas [0]
    .missionsUrl);
    const missionDetails = syncGetJSON (missões [0] .detailsUrl); // Estude a descrição da missão

} catch (e) {
  // Oh não, não conseguimos obter os detalhes da missão
}
```

Embora esse código seja ótimo por sua simplicidade e tratamento de erros, ele bloqueia a IU, o que resulta em usuários insatisfeitos. Idealmente, gostaríamos de alterar esse código para que nenhum bloqueio ocorra durante uma tarefa de longa execução. Uma maneira de fazer isso é combinando geradores e promessas.

Como sabemos, ceder de um gerador suspende a execução do gerador sem bloqueio. Para acordar o gerador e continuar sua execução, temos que chamar o Próximo método no iterador do gerador. Promises, por outro lado, nos permitem especificar um callback que será acionado caso possamos obter o valor prometido e um callback que será acionado caso ocorra um erro.

A ideia, então, é combinar geradores e promessas da seguinte maneira: Colocamos o código que usa tarefas assíncronas em um gerador e executamos essa função do gerador. Quando alcançamos um ponto na execução do gerador que chama uma tarefa assíncrona, criamos uma promessa que representa o valor dessa tarefa assíncrona. Porque não temos ideia de quando essa promessa será resolvida (ou mesmo se será resolvida), neste ponto de execução do gerador, nós rendemos do gerador, para não causarmos bloqueio. Depois de um tempo, quando a promessa é cumprida, continuamos a execução de nosso gerador chamando o iterador Próximo método. Fazemos isso quantas vezes forem necessárias. Veja a lista a seguir para um exemplo prático.

Listagem 6.19 Combinando geradores e promessas



O assíncrono A função pega um gerador, o chama e cria um iterador que será usado para retomar a execução do gerador. Dentro de assíncrono função, nós declaramos um lidar
função que lida com um valor de retorno do gerador - uma “iteração” de nosso iterador. Se o resultado do gerador é uma promessa que é resolvida com sucesso, usamos o iterador Próximo método para enviar o valor prometido de volta ao gerador e retomar a execução do gerador. Se ocorrer um erro e a promessa for rejeitada, lançamos esse erro para o gerador usando o iterador lançar método (disse que seria útil). Continuamos fazendo isso até que o gerador diga que está pronto.

NOTA Este é um esboço, uma quantidade mínima de código necessária para combinar geradores e promessas. Não recomendamos que você use esse código na produção.

Agora vamos dar uma olhada mais de perto no gerador. Na primeira invocação do iterador
Próximo método, o gerador executa até o primeiro getJSON ("data / ninjas.json")

chamar. Esta chamada cria uma promessa que acabará por conter a lista de informações sobre o nosso ninjas. Como esse valor é buscado de forma assíncrona, não temos ideia de quanto tempo o navegador levará para obtê-lo. Mas sabemos uma coisa: não queremos bloquear a execução do aplicativo enquanto aguardamos. Por esse motivo, neste momento de execução, o gerador cede o controle, que pausa o gerador, e devolve o fluxo de controle à invocação do lidar função. Porque o valor gerado é um getJSON promessa, no lidar função, usando o então e

pegar métodos da promessa, registramos um sucesso e um retorno de chamada de erro e continuamos a execução. Com isso, o fluxo de controle sai da execução do lidar
função e o corpo do assíncrono função, e continua após a chamada para o
assíncrono função (em nosso caso, não há mais código depois, então ele fica inativo). Durante este tempo, nossa função gerador espera pacientemente suspensa, sem bloquear a execução do programa.

Muito, muito mais tarde, quando o navegador recebe uma resposta (positiva ou negativa), um dos retornos de chamada de promessa é invocado. Se a promessa foi resolvida com sucesso, o retorno de chamada de sucesso é invocado, o que, por sua vez, causa a execução do iterador Próximo método, que pede ao gerador outro valor. Isso tira o gerador da suspensão e envia para ele o valor passado pelo callback. Isso significa que entramos novamente no corpo do nosso gerador, após o primeiro produzem expressão, cujo valor se torna o ninjas lista que foi obtida de forma assíncrona do servidor. A execução da função do gerador continua, e o valor é atribuído ao

plano variável.

Na próxima linha do gerador, usamos alguns dos dados obtidos, ninjas [0]
. missionUrl, para fazer outro getJSON chamada que cria outra promessa que deve conter uma lista de missões feitas pelo ninja mais popular. Novamente, como essa é uma tarefa assíncrona, não temos ideia de quanto tempo vai demorar, então, novamente entregamos a execução e repetimos todo o processo.

Este processo é repetido enquanto houver tarefas assíncronas no gerador. Isso era um pouco complexo, mas gostamos deste exemplo porque ele combina muitas coisas que você aprendeu até agora:

- *Funciona como objetos de primeira classe* - Enviamos uma função como um argumento para o assíncrono função.
- *Funções geradoras* —Usamos sua capacidade de suspender e retomar a execução.
- *Promessas* —Eles nos ajudam a lidar com o código assíncrono.
- *Callbacks* —Nós registramos retornos de chamada de sucesso e fracasso em nossas promessas.
- *Funções de seta* - Devido à sua simplicidade, para retornos de chamada, usamos as funções de seta.
- *Fechamentos* —O iterador, por meio do qual controlamos o gerador, é criado no assíncrono função, e a acessamos, por meio de fechamentos, nos retornos de chamada de promessa.

Agora que já passamos por todo o processo, vamos parar um minuto para avaliar como é muito mais elegante o código que implementa nossa lógica de negócios. Considere isto:

```
getJSON ("data / ninjas.json", (err, ninjas) => {
    if (err) {console.log ("Erro ao buscar ninjas", err); Retorna; }

    getJSON (ninjas [0] .missionsUrl, (err, missões) => {
        if (err) {console.log ("Erro ao localizar missões ninja", err); Retorna; } console.log (missões);

    })
});
```

Em vez de fluxo de controle e tratamento de erros mistos e código um pouco confuso, terminamos com algo assim:

```
assíncrono (função * () {
    tentar {
        const ninjas = rendimento getJSON ("dados / ninjas.json"); missões const = rendimento getJSON (ninjas
        [0] .missionsUrl);

        // Todas as informações recebidas
    }
    pegar (e) {
        //Ocorreu um erro
    }
});
```

Este resultado final combina as vantagens do código síncrono e assíncrono. Do código síncrono, temos a facilidade de compreensão e a capacidade de usar todos os mecanismos de controle de fluxo e tratamento de exceção padrão, como loops e tentar pegar
declarações. Do código assíncrono, obtemos a natureza não bloqueadora; a execução de nosso aplicativo não é bloqueada enquanto espera por tarefas assíncronas de longa duração.

6.4.1 Olhando para o futuro - a função assíncrona

Observe que ainda tivemos que escrever algum código clichê; tivemos que desenvolver um assíncrono
função que se encarrega de lidar com promessas e solicitar valores do gerador. Embora possamos escrever essa função apenas uma vez e, em seguida, reutilizá-la em todo o nosso código, seria ainda melhor se não tivéssemos que pensar sobre isso. As pessoas encarregadas de JavaScript estão bem cientes da utilidade da combinação de geradores e promessas, e querem tornar nossas vidas ainda mais fáceis construindo um suporte de linguagem direto para combinar geradores e promessas.

Para essas situações, o plano atual é incluir duas novas palavras-chave, assíncrono e aguardam, isso cuidaria desse código clichê. Em breve, poderemos escrever algo assim:

```
( assíncrono function () {
    tentar {
        const ninjas = aguardam getJSON ("data / ninjas.json"); missões const = aguardam getJSON (missões [0]
        .missionsUrl);

        console.log (missões);
    }
```

```
    pegar (e) {
        console.log ("Erro:", e);
    }
}) ()
```

Nós usamos o assíncrono palavra-chave na frente da palavra-chave de função para especificar que esta função depende de valores assíncronos, e em cada lugar onde chamamos uma tarefa assíncrona, colocamos o aguardam palavra-chave que diz ao mecanismo JavaScript, aguarde este resultado sem bloquear. No fundo, tudo acontece como discutimos anteriormente ao longo do capítulo, mas agora não precisamos nos preocupar com isso.

NOTA As funções assíncronas aparecerão na próxima parte do JavaScript. Atualmente nenhum navegador oferece suporte, mas você pode usar transpiladores como Babel ou Traceur se desejar usar async em seu código hoje.

6,5 Resumo

- Geradores são funções que geram sequências de valores - não todos de uma vez, mas por solicitação.
- Ao contrário das funções padrão, os geradores podem suspender e retomar sua execução. Depois que um gerador gera um valor, ele suspende sua execução sem bloquear o thread principal e espera pacientemente pela próxima solicitação.
- Um gerador é declarado colocando um asterisco (*) após a palavra-chave da função. Dentro do corpo do gerador, podemos usar o novo produzem palavra-chave que produz um valor e suspende a execução do gerador. Se quisermos ceder a outro gerador, usamos o produzem* operador.
- Chamar um gerador cria um objeto iterador por meio do qual controlamos a execução do gerador. Solicitamos novos valores do gerador usando o iterador Próximo método, e podemos até lançar exceções no gerador chamando o método do iterador lançar método. Além disso, o Próximo método pode ser usado para enviar valores para o gerador.
- Uma promessa é um espaço reservado para os resultados de um cálculo; é uma garantia de que eventualmente saberemos o resultado da computação, na maioria das vezes uma computação assíncrona. Uma promessa pode ter sucesso ou falhar e, depois de cumprida, não haverá mais mudanças.
- As promessas simplificam significativamente nossas negociações com tarefas assíncronas. Podemos facilmente trabalhar com sequências de etapas assíncronas interdependentes usando o
então método para encadear promessas. O tratamento paralelo de várias etapas assíncronas também é bastante simplificado; nós usamos o Promise.all método.
- Podemos combinar geradores e promessas para lidar com tarefas assíncronas com a simplicidade do código síncrono.

6,6 Exercícios

1 Depois de executar o código a seguir, quais são os valores das variáveis a1 para a4?

```
function * EvenGenerator () {
    deixe num = 2;
    enquanto (verdadeiro) {
        rendimento num;
        num = num + 2;
    }
}

let generator = EvenGenerator ();

deixe a1 = generator.next (). value; deixe a2 = generator.next
(). value;
deixe a3 = EvenGenerator (). next (). valor; deixe a4 = generator.next ().
value;
```

2 Qual é o conteúdo do ninjas array depois de executar o seguinte código? (Dica: pense sobre como o para de loop pode ser implementado com um enquanto ciclo.)

```
function * NinjaGenerator () {
    produzir "Yoshi";
    retornar "Hattori";
    produzir "Hanzo";
}

var ninjas = [];
para (deixe ninja de NinjaGenerator ()) {
    ninjas.push (ninja);
}

ninjas;
```

3 Quais são os valores das variáveis a1 e a2, depois de executar o seguinte código?

```
function * Gen (val) {
    val = rendimento val * 2; rendimento
    val;
}

deixe gerador = Gen (2);
deixe a1 = generator.next (3) .value; deixe a2 = generator.next
(4) .value;
```

4 Qual é a saída do código a seguir?

```
promessa const = nova promessa ((resolver, rejeitar) => {
    rejeitar ("Hattori");
});

promessa.então (val => alert ("Sucesso:" + val))
                . catch (e => alert ("Erro:" + e));
```

5 Qual é a saída do código a seguir?

```
promessa const = nova promessa ((resolver, rejeitar) => {
    resolver ("Hattori");
    setTimeout (() => rejeitar ("Yoshi"), 500);
});

promessa.então (val => alert ("Sucesso:" + val))
                . catch (e => alert ("Erro:" + e));
```

Parte 3

Cavando em objetos e fortalecendo seu código

Como você aprendeu os meandros das funções, continuaremos nossa exploração do JavaScript examinando mais de perto os fundamentos dos objetos, no capítulo 7.

No capítulo 8, estudaremos como controlar o acesso e monitorar nossos objetos com getters e setters, e com proxies, um tipo completamente novo de objeto em JavaScript.

Vamos dar uma olhada nas coleções no capítulo 9 - as tradicionais, como matrizes, bem como tipos completamente novos, como mapas e conjuntos.

A partir daí, passaremos para as expressões regulares no capítulo 10. Você aprenderá que muitas tarefas que costumavam exigir resmas de código para serem realizadas podem ser condensadas em um simples punhado de instruções por meio do uso adequado de expressões regulares JavaScript.

Finalmente, no capítulo 11, mostraremos como estruturar seus aplicativos JavaScript em unidades menores e bem organizadas de funcionalidade, chamadas de módulos.

7 Orientação do objeto com protótipos

Este capítulo cobre

- Explorando protótipos
- Usando funções como construtores
- Estendendo objetos com protótipos
- Evitando pegadinhas comuns
- Construindo classes com herança

Você aprendeu que as funções são objetos de primeira classe em JavaScript, que os encerramentos os tornam incrivelmente versáteis e úteis e que você pode combinar funções geradoras com promessas para resolver o problema do código assíncrono. Agora estamos prontos para lidar com outro aspecto importante do JavaScript: protótipos de objetos.

UMA *protótipo* é um objeto ao qual a pesquisa de uma propriedade específica pode ser delegada. Protótipos são um meio conveniente de definir propriedades e funcionalidades que estarão automaticamente acessíveis a outros objetos. Os protótipos têm um propósito semelhante ao das classes nas linguagens orientadas a objetos clássicas. Na verdade, o principal uso de protótipos em JavaScript é na produção de código escrito em um formato orientado a objetos

forma, semelhante, mas não exatamente como, código em linguagens baseadas em classe mais convencionais, como Java ou C #.

Neste capítulo, vamos nos aprofundar em como os protótipos funcionam, estudar sua conexão com funções construtoras e ver como imitar alguns dos recursos orientados a objetos frequentemente usados em outras linguagens orientadas a objetos mais convencionais. Também exploraremos uma nova adição ao JavaScript, o aula palavra-chave, que não traz exatamente classes completas para o JavaScript, mas nos permite simular classes e herança facilmente. Vamos começar a explorar.

Você sabe?

Como você testa se um objeto tem acesso a um determinado propriedade lar?

Por que uma cadeia de protótipos é importante para trabalhar com objetos em JavaScript?

As classes ES6 mudam a forma como o JavaScript funciona com objetos?

7,1 *Compreendendo os protótipos*

Em JavaScript, os objetos são coleções de propriedades nomeadas com valores. Por exemplo, podemos criar facilmente novos objetos com notação literal de objeto:

```
let obj = {
    prop1: 1,
    prop2: function () {},
    prop3: {}
}
```

Atribui um valor simples

Atribui uma função

Atribui outro objeto

Como podemos ver, as propriedades do objeto podem ser valores simples (como números ou strings), funções e até mesmo outros objetos. Além disso, JavaScript é uma linguagem altamente dinâmica e as propriedades atribuídas a um objeto podem ser facilmente alteradas, modificando e excluindo as propriedades existentes:

```
obj.prop1 = 1;
obj.prop1 = [];
delete obj.prop2;
```

prop1 armazena um número simples.

Atribui um valor de um completamente diferente tipo, aqui uma matriz

Remove a propriedade do objeto

Podemos até adicionar propriedades completamente novas:

```
obj.prop4 = "Olá";
```

Adiciona uma propriedade completamente nova

No final, todas essas modificações deixaram nosso objeto simples no seguinte estado:

```
{
    prop1: [],
    prop3: {},
    prop4: "Olá"
};
```

Ao desenvolver software, nos esforçamos para não reinventar a roda, portanto, queremos reutilizar o máximo de código possível. Uma forma de reutilização de código que também ajuda a organizar nossos programas é *herança,* estendendo as características de um objeto para outro. Em JavaScript, a herança é implementada com prototipagem.

A ideia de prototipagem é simples. Cada objeto pode ter uma referência ao seu *protótipo,* um objeto ao qual a busca por uma propriedade particular pode ser delegada, se o próprio objeto não tiver essa propriedade. Imagine que você está participando de um teste de jogo com um grupo de pessoas e que o apresentador do programa faz uma pergunta. Se você sabe a resposta, você a dá imediatamente; se não souber, pergunte à pessoa ao seu lado. É simples assim.

Vamos dar uma olhada na lista a seguir.

Listagem 7.1 Com protótipos, objetos podem acessar propriedades de outros objetos

```
const yoshi = {skulk: verdadeiro}; const hattori = {sneak: true};
const kuma = {creep: true};

assert ("skulk" em yoshi, "Yoshi pode skulk"); assert (! ("sneak" em yoshi)), "Yoshi não pode
sneak"); assert (! ("rastejar" em yoshi)), "Yoshi não pode rastejar");

Object.setPrototypeOf (yoshi, hattori);

assert ("sneak" em yoshi, "Yoshi can now sneak"); assert (! ("creep" in hattori)), "Hattori não pode creep");

Object.setPrototypeOf (hattori, kuma);
assert ("creep" em hattori, "Hattori pode agora creep"); assert ("creep" em yoshi, "Yoshi pode
creep");
```

Cria três objetos, cada um com sua própria propriedade

Use o objeto . setPrototypeOf método para definir um objeto como o protótipo de outro objeto.

yoshi tem acesso apenas a sua própria propriedade, oculta.

Ao definir hattori como protótipo de yoshi, yoshi agora tem acesso aos Hattori's propriedades.

Atualmente, Hattori não pode rastejar.

Define kuma como um protótipo de Hattori

Agora hattori tem acesso assustar.

yoshi também tem acesso a creep, por meio de hattori.

Neste exemplo, começamos criando três objetos: yoshi, hattori, e kuma. Cada um tem uma propriedade específica acessível apenas para aquele objeto: Apenas yoshi posso esquivar-se, só Hattori posso esgueirar-se, e somente Kuma posso rastejar. Veja a figura 7.1.

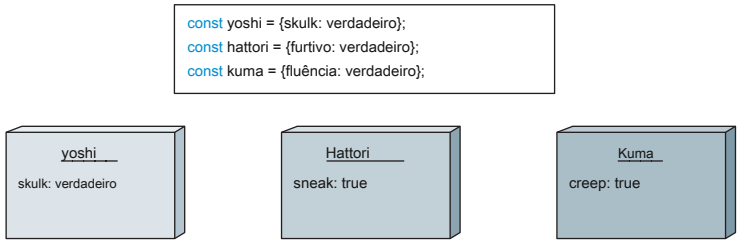


Figura 7.1 Inicialmente, cada objeto tem acesso apenas às suas próprias propriedades.

Para testar se um objeto tem acesso a uma propriedade particular, podemos usar o no operador. Por exemplo, executando skulk em yoshi retorna verdade, Porque yoshi tem acesso ao esconder-se propriedade; enquanto executando esgueirar-se em yoshi retorna falso.

Em JavaScript, a propriedade prototype do objeto é uma propriedade interna que não pode ser acessada diretamente (então a marcamos com [[protótipo]]). Em vez disso, o método integrado Object.setPrototypeOf recebe dois argumentos de objeto e define o segundo objeto como o protótipo do primeiro. Por exemplo, ligando Object.setPrototypeOf (yoshi, hattori); estabelece Hattori como um protótipo de yoshi.

Como resultado, sempre que perguntamos yoshi por uma propriedade que não tem, yoshi delegados que procuram hattori. Podemos acessar Hattori de esgueirar-se propriedade através de yoshi.
Veja a figura 7.2.

Podemos fazer algo semelhante com Hattori e kuma. Usando o Object.setPrototypeOf método, podemos definir Kuma como o protótipo de hattori. Se então perguntarmos Hattori por uma propriedade que ele não possui, essa pesquisa será delegada a kuma. Nesse caso, Hattori agora tem acesso a Kuma de rastejar propriedade. Veja a figura 7.3.

É importante enfatizar que todo objeto pode ter um protótipo, e o protótipo de um objeto também pode ter um protótipo, e assim por diante, formando um *cadeia de protótipo.* A delegação de pesquisa para uma propriedade específica ocorre em toda a cadeia e só para quando não há mais protótipos para explorar. Por exemplo, como mostrado na figura 7.3, perguntando yoshi pelo valor do rastejar propriedade aciona a pesquisa pela propriedade primeiro em yoshi. Como a propriedade não foi encontrada, yoshi protótipo de, hattori, é pesquisado. De novo, Hattori não tem uma propriedade chamada rastejar, então Hattori protótipo de, kuma, é pesquisado e a propriedade é finalmente encontrada.

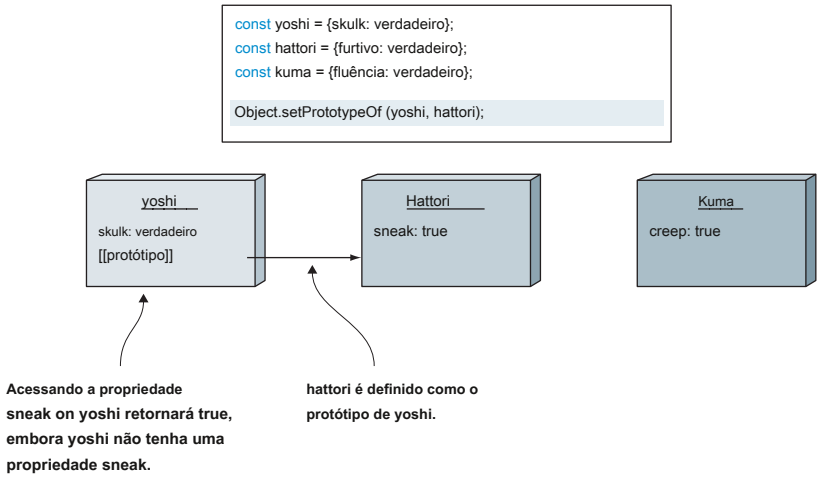


Figura 7.2 Quando acessamos uma propriedade que o objeto não possui, o protótipo do objeto é pesquisado por essa propriedade. Aqui, podemos acessar **Hattori** de **esgueirar-se** propriedade através de **yoshi,** Porque **yoshi** é **Hattori** protótipo de.

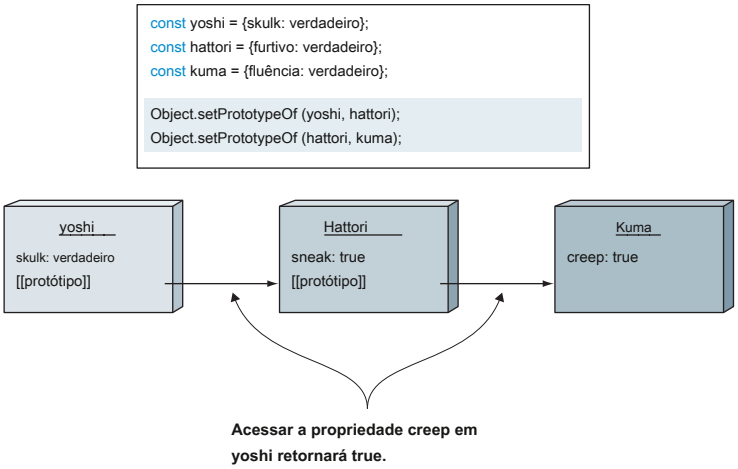


Figura 7.3 A busca por uma determinada propriedade pára quando não há mais protótipos para explorar. Acessando **yoshi.creep** aciona a pesquisa primeiro em **yoshi,** então em **hattori,** e finalmente em **kuma.**

Agora que temos uma ideia básica de como a busca por uma propriedade específica ocorre por meio da cadeia de protótipos, vamos ver como os protótipos são usados na construção de novos objetos com funções construtoras.

7,2 Construção de objetos e protótipos

A maneira mais simples de criar um novo objeto é com uma instrução como esta:

```
guerreiro const = {};
```

Isso cria um objeto novo e vazio, que podemos preencher com propriedades por meio de instruções de atribuição:

```
guerreiro const = {}; warrior.name = 'Saito';

warrior.occupation = 'atirador';
```

Mas aqueles que vêm de um background orientado a objetos podem perder o encapsulamento e a estruturação que vem com um construtor de classe, uma função que serve para inicializar um objeto a um estado inicial conhecido. Afinal, se vamos criar várias instâncias do mesmo tipo de objeto, atribuir as propriedades individualmente não é apenas tedioso, mas também altamente sujeito a erros. Gostaríamos de poder consolidar o conjunto de propriedades e métodos para uma classe de objetos em um só lugar.

JavaScript fornece esse mecanismo, embora em uma forma diferente da maioria das outras linguagens. Como as linguagens orientadas a objetos, como Java e C ++, JavaScript emprega o novo operador para instanciar novos objetos por meio de construtores, mas não há definição de classe verdadeira em JavaScript. Em vez disso, o novo operador, aplicado a uma função construtora (como você viu no capítulo 3), dispara a criação de um objeto recém-alocado.

O que não aprendemos nos capítulos anteriores é que cada função tem um objeto protótipo que é definido automaticamente como o protótipo dos objetos criados com aquela função. Vamos ver como isso funciona na lista a seguir.

Listagem 7.2 Criando um **novo** instância com um método prototipado

```
função Ninja () {}
Ninja.prototype.swingSword = function () {
    return true;
};

const ninja1 = Ninja (); assert (ninja1 === undefined,
                "Nenhuma instância de Ninja criada.");

const ninja2 = novo Ninja (); afirmar (ninja2 &&
                ninja2.swingSword &&
                ninja2.swingSword (),
                "A instância existe e o método pode ser chamado." );
```

Define uma função que não faz nada e não retorna nada

Cada função tem um objeto de protótipo embutido, que podemos modificar livremente.

Chama a função como uma função. O teste confirma que nada parece acontecer.

Chama a função como um construtor. O teste confirma que não apenas uma nova instância de objeto é criada, mas possui o método do protótipo da função.

Neste código, definimos uma função aparentemente não fazer nada chamada Ninja que invocaremos de duas maneiras: como uma função "normal", const ninja1 = Ninja (); e como uma construção
tor, const ninja2 = novo Ninja () ;.

Quando a função é criada, ela imediatamente obtém um novo objeto atribuído ao seu objeto protótipo, um objeto que podemos estender como qualquer outro objeto. Neste caso, adicionamos um swingSword método para isso:

```
Ninja.prototype.swingSword = function () {
    return true;
};
```

Em seguida, testamos a função. Primeiro chamamos a função normalmente e armazenamos seu resultado na variável ninja1. Olhando para o corpo da função, vemos que ela não retorna nenhum valor, então esperamos ninja1 para testar como Indefinido, que afirmamos ser verdade. Como uma função simples, Ninja não parece ser tão útil.

Em seguida, chamamos a função por meio do novo operador, invocando-o como um *construtor,* e algo completamente diferente acontece. A função é chamada mais uma vez, mas desta vez um objeto recém-alocado foi criado e definido como o contexto da função (e está acessível através do esta palavra-chave). O resultado retornou do novo operador é uma referência a este novo objeto. Nós então testamos isso ninja2 tem uma referência ao objeto recém-criado, e que esse objeto tem um swingSword método que podemos chamar. Veja a figura 7.4 para uma visão geral do estado atual do aplicativo.

```
função Ninja () {}
Ninja.prototype.swingSword = function () {
    Retorna verdade;
};
...
const ninja2 = novo Ninja ();
```

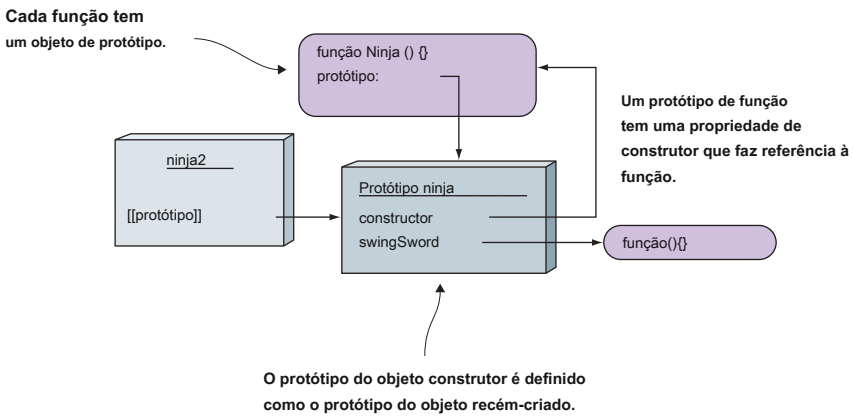


Figura 7.4 Cada função, quando criada, obtém um novo objeto de protótipo. Quando usamos uma função como construtor, o protótipo do objeto construído é definido como o protótipo da função.

Como você pode ver, uma função, quando criada, obtém um novo objeto que é atribuído à sua propriedade prototype. O objeto de protótipo inicialmente tem apenas uma propriedade, construtor, que faz referência à função (vamos revisitar o construtor propriedade mais tarde).

Quando usamos uma função como construtor (por exemplo, chamando novo Ninja ()), o protótipo do objeto recém-construído é definido como o objeto referenciado pelo protótipo da função construtora.

Neste exemplo, estendemos o Ninja.prototype com o swingSword método, e quando o ninja2 objeto é criado, seu protótipo propriedade está definida para Ninja protótipo de. Portanto, quando tentamos acessar o swingSword propriedade em ninja2, a busca por essa propriedade é delegada ao Ninja objeto de protótipo. Notar que *todo* objetos criados com o Ninja construtor terá acesso ao swingSword método. Agora isso é reutilização de código!

O swingSword método é uma propriedade do Ninja protótipo de, e não uma propriedade de ninja instâncias. Vamos explorar essa diferença entre propriedades de instância e propriedades de protótipo.

7.2.1 Propriedades da instância

Quando a função é chamada como um construtor por meio do novo operador, seu contexto é definido como a nova instância do objeto. Além de expor propriedades por meio do protótipo, podemos

inicializar valores dentro da função construtora por meio do esta parâmetro. Vamos examinar a criação de tais propriedades de instância na próxima listagem.

Listagem 7.3 Observando a precedência das atividades de inicialização

```
function Ninja () {
  this.swung = false;
  this.swingSword = function () {
      return! this.swung;
  };
}
Ninja.prototype.swingSword = function () {
    return this.swung;
};

const ninja = novo Ninja (); assert
(ninja.swingSword (),
          "Chamado o método de instância, não o método de protótipo.");
```

Cria uma instância variável que contém um valor booleano inicializado como falso

Cria um método de instância que retorna o inverso do valor da variável de instância oscilada

Define um método de protótipo com o mesmo nome do método de instância. Qual terá precedência?

Constrói uma instância Ninja e afirma que o método da instância substituirá o método do protótipo de mesmo nome

A Listagem 7.3 é semelhante ao exemplo anterior em que definimos um swingSword método adicionando-o ao protótipo propriedade do construtor:

```
Ninja.prototype.swingSword = function () {
    return this.swung;
};
```

Mas também adicionamos um método com nome idêntico dentro da própria função construtora:

```
function Ninja () {
    this.swung = false;
    this.swingSword = function () {
        return! this.swung;
    };
}
```

Os dois métodos são definidos para retornar resultados opostos para que possamos dizer qual será chamado.

NOTA Isso não é algo que aconselharíamos fazer no código do mundo real; muito pelo contrário. Estamos fazendo isso aqui apenas para demonstrar a precedência das propriedades.

Quando você executa o teste, vê que ele passa! Isso mostra que os membros da instância irão ocultar propriedades com o mesmo nome definido no protótipo. Veja a figura 7.5.

```
função Ninja () {
    this.swung = false;
    this.swingSword = function () {
        Retorna ! this.swung;
    };
}
Ninja.prototype.swingSword = função () {
    Retorna this.swung;
};
const ninja = novo Ninja ();
```

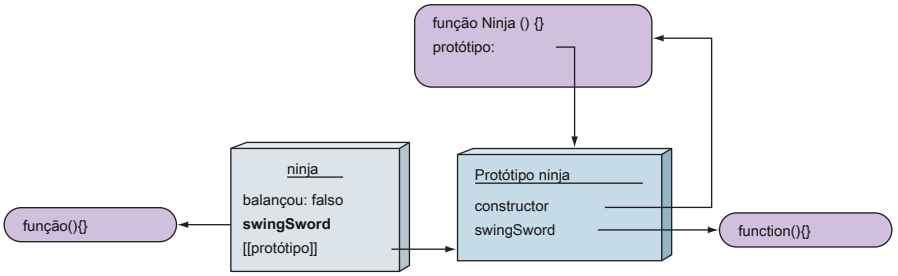


Figura 7.5 Se uma propriedade puder ser encontrada na própria instância, o protótipo nem mesmo é consultado!

Dentro da função construtora, o esta palavra-chave refere-se ao objeto recém-criado, portanto, as propriedades adicionadas dentro do construtor são criadas diretamente no novo ninja
instância. Mais tarde, quando acessarmos a propriedade swingSword sobre ninja, não há necessidade de atravessar a cadeia de protótipo (conforme mostrado na figura 7.4); a propriedade criada no construtor é imediatamente encontrada e retornada (consulte a figura 7.5).

Isso tem um efeito colateral interessante. Dê uma olhada na figura 7.6, que mostra o estado do aplicativo se criarmos três ninja instâncias.

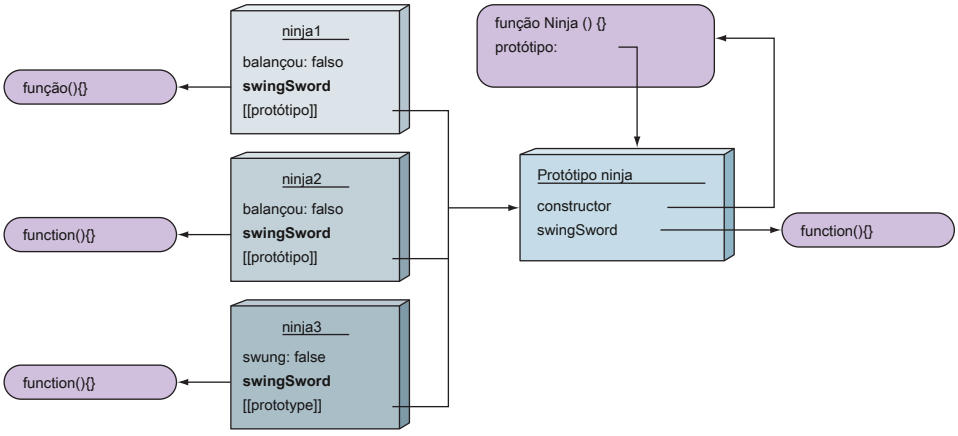


Figura 7.6 Cada instância obtém sua própria versão das propriedades criadas no construtor, mas todas elas têm acesso às mesmas propriedades do protótipo.

Como você pode ver, cada ninja instância obtém sua própria versão das propriedades que foram criadas dentro do construtor, enquanto todas têm acesso às mesmas propriedades do protótipo. Isso é bom para propriedades de valor (por exemplo, balançado) que são específicos para cada instância do objeto. Mas, em certos casos, pode ser problemático para métodos.

Neste exemplo, teríamos três versões do swingSword método que todos executam a mesma lógica. Isso não é um problema se criarmos alguns objetos, mas é algo a que prestar atenção se planejamos criar um grande número de objetos. Como cada cópia de método se comporta da mesma forma, a criação de várias cópias geralmente não faz sentido, porque apenas consome mais memória. Claro, em geral, o mecanismo JavaScript pode realizar algumas otimizações, mas isso não é algo em que se confiar. Dessa perspectiva, faz sentido colocar métodos de objeto apenas no protótipo da função, pois dessa forma temos um único método compartilhado por todas as instâncias de objeto.

NOTA Lembre-se do capítulo 5 sobre encerramentos: os métodos definidos nas funções construtoras nos permitem imitar variáveis de objetos privados. Se isso for algo de que precisamos, especificar métodos dentro dos construtores é o único caminho a percorrer.

7.2.2 Efeitos colaterais da natureza dinâmica do JavaScript

Você já viu que JavaScript é uma linguagem dinâmica na qual as propriedades podem ser facilmente adicionadas, removidas e modificadas à vontade. A mesma coisa vale para protótipos, tanto protótipos de função quanto protótipos de objeto. Veja a lista a seguir.

Listagem 7.4 Com protótipos, tudo pode ser alterado em tempo de execução

```
function Ninja () {
    this.swung = true;
}

const ninja1 = novo Ninja ();

Ninja.prototype.swingSword = function () {
    return this.swung;
};
assert (ninja1.swingSword (),
            "O método existe, mesmo fora de serviço.");

Ninja.prototype = {
  pierce: function () {
      return true;
  }
}

assert (ninja1.swingSword (),
            "Nosso ninja ainda pode balançar!");

const ninja2 = novo Ninja ();
assert (ninja2.pierce (), "Ninjas recém-criados podem perfurar"); assert (! ninja2.swingSword, "Mas eles não
podem balançar!");
```

Define um construtor que cria um Ninja com uma única propriedade booleana

Cria uma instância de Ninja chamando a função do construtor via o "novo" operador

Adiciona um método ao protótipo após o objeto foi criado

Mostra que o método existe no objeto

Completamente substitui o do Ninja protótipo com um novo objeto através do método de perfuração

Embora tenhamos substituído completamente o protótipo do construtor Ninja, nosso Ninja ainda pode brandir uma espada, porque mantém uma referência ao antigo protótipo Ninja.

Ninjas recém-criados referenciar o novo protótipo, para que possam perfurar, mas não balançar.