


```
    element.attachEvent ("on" + tipo, identificador);  
  }  
}
```



Vincula-se usando um
API proprietária

Neste exemplo, preparamos nosso código para o futuro, sabendo (ou tendo esperanças) de que algum dia a Microsoft colocará o Internet Explorer em linha com os padrões DOM. Se o navegador for compatível com a API compatível com os padrões, usamos a detecção de recursos para inferir isso e usamos a API padrão, o `addEventListener` método. Caso contrário, verificamos se o método proprietário do IE `attachEvent` está disponível e use-o. Se tudo mais falhar, não fazemos nada.

A maioria das mudanças futuras da API, infelizmente, não são fáceis de prever e não há como prever os próximos bugs. Esse é apenas um dos motivos importantes pelos quais enfatizamos os testes ao longo deste livro. Em face das mudanças imprevisíveis que afetarão nosso código, o melhor que podemos esperar é ser diligentes no monitoramento de nossos testes para cada versão do navegador e resolver rapidamente os problemas que as regressões podem apresentar.

Ter um bom conjunto de testes e acompanhar de perto os próximos lançamentos do navegador é absolutamente a melhor maneira de lidar com futuras regressões dessa natureza. Não precisa ser desgastante para o seu ciclo normal de desenvolvimento, que já deve incluir testes de rotina. A execução desses testes em novas versões do navegador deve sempre ser considerada no planejamento de qualquer ciclo de desenvolvimento.

Você pode obter informações sobre os próximos lançamentos de navegador nos seguintes locais:

- Microsoft Edge (um sucessor do IE): <http://blogs.windows.com/msedgedev/>
- Raposa de fogo: <http://ftp.mozilla.org/pub/firefox/nightly/latest-trunk/>
- WebKit (Safari): <https://webkit.org/nightly/>
- Ópera: <https://dev.opera.com/>
- Cromada: <http://chrome.blogspot.hr/>

Diligência é importante. Como nunca podemos prever totalmente os bugs que serão introduzidos por um navegador, é melhor garantir que permaneçamos no controle de nosso código e evitemos rapidamente quaisquer crises que possam surgir.

Felizmente, os fornecedores de navegadores estão fazendo muito para garantir que regressões dessa natureza não ocorram, e os navegadores costumam ter suítes de teste de várias bibliotecas JavaScript integradas em sua suíte de teste de navegador principal. Isso garante que nenhuma regressão futura será introduzida que afete essas bibliotecas diretamente. Embora isso não capture todas as regressões (e certamente não o fará em todos os navegadores), é um ótimo começo e mostra um bom progresso por parte dos fornecedores de navegadores na prevenção do maior número de problemas possível.

Nesta seção, examinamos quatro pontos principais de preocupação para o desenvolvimento de JavaScript reutilizável: bugs do navegador, correções de bugs do navegador, código externo e regressões do navegador. O quinto ponto - recursos ausentes nos navegadores - merece uma menção especial, portanto, o cobriremos na próxima seção, junto com outras estratégias de implementação relevantes para aplicativos da web entre navegadores.

14.3 Estratégias de implementação

Saber de quais questões estar atento é apenas metade da batalha. Descobrir soluções eficazes e usá-las para implementar código robusto para vários navegadores é outra questão.

Uma ampla variedade de estratégias está disponível e, embora nem todas as estratégias funcionem em todas as situações, a variedade apresentada nesta seção cobre a maioria das preocupações que precisaremos abordar em nossas robustas bases de código. Vamos começar com algo que seja fácil e quase sem problemas.

14.3.1 *Correções seguras para navegadores diferentes*

As classes mais simples (e mais seguras) de correções entre navegadores são aquelas que exibem duas características importantes:

- Eles não têm efeitos negativos ou colaterais em outros navegadores.
- Eles não usam nenhuma forma de navegador ou detecção de recursos.

As instâncias para aplicar essas correções podem ser raras, mas elas são uma tática pela qual devemos sempre nos empenhar em nossos aplicativos.

Vejamos um exemplo. O seguinte snippet de código representa uma mudança (retirada do jQuery) que ocorreu ao trabalhar com o Internet Explorer:

```
// ignore os valores negativos de largura e altura
if ((key == 'width' || key == 'height') && parseFloat (value) <0)
    valor = indefinido;
```

Algumas versões do IE lançam uma exceção quando um valor negativo é definido no altura ou largura propriedades de estilo. Todos os outros navegadores ignoram a entrada negativa. Esta solução alternativa ignora todos os valores negativos em *todo* navegadores. Essa alteração evita que uma exceção seja lançada no Internet Explorer e não tem efeito em nenhum outro navegador. Esta é uma mudança indolor que fornece uma API unificada para o usuário (porque lançar exceções inesperadas nunca é desejado).

Outro exemplo desse tipo de correção (também do jQuery) aparece no código de manipulação de atributos. Considere isto:

```
if (nome == "tipo" &&
    elem.nodeName.toLowerCase () == "input" &&
    elem.parentNode)
    lançar "atributo de tipo não pode ser alterado";
```

O Internet Explorer não nos permite manipular o modelo atributo de elementos de entrada que já fazem parte do DOM; tentativas de alterar esse atributo resultam em uma exceção proprietária lançada. jQuery chegou a uma solução intermediária: não permite *todo* tentativas de manipular o modelo atributo em elementos de entrada injetados em todos os navegadores, lançando uma exceção informativa uniforme.

Esta mudança na base de código jQuery não requer navegador ou detecção de recurso; ele unifica a API em todos os navegadores. A ação ainda resulta em uma exceção, mas essa exceção é uniforme em todos os tipos de navegador.

Esta abordagem particular pode ser considerada controversa. Limita propositalmente os recursos da biblioteca em todos os navegadores por causa de um bug que existe em apenas um. A equipe da jQuery avaliou essa decisão com cuidado e decidiu que era melhor ter uma API unificada que funcionasse de forma consistente do que uma API que quebrasse inesperadamente ao desenvolver código para vários navegadores. Você pode se deparar com situações como essa ao desenvolver suas próprias bases de código reutilizáveis e precisará considerar cuidadosamente se uma abordagem de limitação como essa é apropriada para seu público.

O importante a ser lembrado para esses tipos de alterações de código é que eles fornecem uma solução que funciona perfeitamente em todos os navegadores sem a necessidade de detecção de navegador ou recurso, tornando-os efetivamente imunes a alterações futuras. Você deve sempre buscar soluções que funcionem dessa maneira, mesmo que as instâncias aplicáveis sejam poucas e distantes entre si.

14.3.2 Detecção de recursos e polyfills

Como discutimos anteriormente, *detecção de recursos* é uma abordagem comumente usada ao escrever código para vários navegadores. Essa abordagem não é apenas simples, mas também geralmente eficaz. Ele funciona determinando se um determinado objeto ou propriedade de objeto existe e, em caso afirmativo, presumindo que ele fornece a funcionalidade implícita. (Na próxima seção, veremos o que fazer nos casos em que essa suposição falha.)

Mais comumente, a detecção de recursos é usada para escolher entre várias APIs que fornecem partes duplicadas de funcionalidade. Por exemplo, o capítulo 10 explorou o `Array.prototype.find`, acessível a todos os arrays, um método que podemos usar para encontrar o primeiro item do array que satisfaça uma determinada condição. Infelizmente, o método é acessível apenas em navegadores que suportam totalmente ES6. Então, o que fazemos quando estamos presos a navegadores que ainda não oferecem suporte a esse recurso? Em geral, como lidamos com os recursos ausentes nos navegadores?

A resposta é polyfilling! Um polyfill é um substituto do navegador. Se um navegador não oferece suporte a uma funcionalidade específica, fornecemos nossa própria implementação. Por exemplo, a Mozilla Developer Network (MDN) fornece polyfills para uma ampla gama de funcionalidades ES6. Entre outras coisas, isso inclui a implementação de JavaScript do `Array.prototype.find` método (<http://mng.bz/d9IU>), conforme mostrado na lista a seguir.

Listagem 14.2 Um polyfill para o `Array.prototype.find` método

```
if (! Array.prototype.find) {
  Array.prototype.find = function (predicado) {
    if (this === null) {
      lance new TypeError ('find chamado em null ou undefined');
    }
    if (predicado typeOf! == 'function') {
      // ...
    }
  }
}
```

**Especifica
nosso próprio
implementação**

← **Fornecer um polyfill apenas se o navegador atual não implementar o método**

```

        lance novo TypeError ('predicado deve ser uma função');
    }
    var list = Object (this);
    var length = list.length >>> 0; var thisArg = argumentos [1];
    valor var;

    para (var i = 0; i < comprimento; i++) {
        valor = lista [i];
        if (predicate.call (thisArg, value, i, list)) {
            valor de retorno;
        }
    }
    retornar indefinido;
};
}

```

← Certifica-se de que o comprimento é um número inteiro não negativo

Encontra o primeiro item da matriz que satisfaz um predicado

Neste exemplo, primeiro usamos a detecção de recursos para verificar se o navegador atual tem suporte integrado para o `achar método`:

```

if (! Array.prototype.find) {
    ...
}

```

Sempre que possível, devemos adotar a forma padrão de realizar qualquer ação. Conforme mencionado antes, isso ajudará a tornar nosso código o mais à prova de futuro possível. Por este motivo, se o navegador já suporta o método, não fazemos nada. Se estamos lidando com um navegador que ainda não alcançou o ES6, fornecemos nossa própria implementação.

Acontece que o núcleo do método é direto. Percorremos o array, chamando a função de predicado passada, que verifica se um item do array satisfaz nossos critérios. Se isso acontecer, nós o devolvemos.

Uma técnica interessante é apresentada nesta listagem:

```
var length = list.length >>> 0;
```

O operador `>>>` é o *operador zero-fill shift right*, que desloca o primeiro operando o número especificado de bits para a direita, enquanto descarta os bits em excesso. Neste caso, este operador é usado para converter o comprimento propriedade para um número inteiro não negativo. Isso é feito porque os índices de array em JavaScript devem ser inteiros sem sinal.

Um uso importante da detecção de recursos é descobrir os recursos fornecidos pelo ambiente do navegador no qual o código está sendo executado. Isso nos permite fornecer recursos que usam esses recursos em nosso código ou determinar se precisamos fornecer um fallback.

O snippet de código a seguir mostra um exemplo básico de detecção da presença de um recurso do navegador usando a detecção de recurso, para determinar se devemos fornecer a funcionalidade completa do aplicativo ou um fallback de experiência reduzida:

```
if (typeof document! == "undefined" &&
    document.addEventListener &&
    document.querySelector &&
    document.querySelectorAll) {
    // Temos API suficiente para trabalhar para construir nosso aplicativo
}
outro {
    // Provide Fallback
}
```

Aqui, testamos se

- O navegador tem um documento carregado
- O navegador fornece um meio de vincular manipuladores de eventos
- O navegador pode encontrar elementos com base em um seletor

A falha em qualquer um desses testes nos leva a recorrer a uma posição de recuo. O que é feito no fallback depende das expectativas dos consumidores do código e dos requisitos colocados no código. Algumas opções podem ser consideradas:

- Poderíamos realizar mais detecção de recursos para descobrir como fornecer uma experiência reduzida que ainda usa algum JavaScript.
- Poderíamos optar por não executar nenhum JavaScript, voltando para o HTML sem script na página.
- Poderíamos redirecionar o usuário para uma versão mais simples do site. O Google faz isso com o Gmail, por exemplo.

Como a detecção de recursos tem pouca sobrecarga (é apenas uma pesquisa de propriedade / objeto) e é relativamente simples em sua implementação, é uma boa maneira de fornecer níveis básicos de fallback, tanto no nível da API quanto do aplicativo. É uma boa escolha para a primeira linha de defesa em sua criação de código reutilizável.

14.3.3 Problemas não testáveis do navegador

Infelizmente, o JavaScript e o DOM têm várias áreas de problemas possíveis que são impossíveis ou proibitivamente caras para testar. Felizmente, essas situações são raras, mas quando as encontramos, sempre vale a pena gastar tempo investigando para ver se há algo que podemos fazer a respeito.

As seções a seguir discutem alguns problemas conhecidos que são impossíveis de testar usando qualquer interação JavaScript convencional.

EVENT HANDLER BINDINGS

Um dos lapsos irritantes dos navegadores é a incapacidade de determinar programaticamente se um manipulador de eventos foi vinculado. Os navegadores não fornecem nenhuma maneira de determinar se alguma função foi associada a um ouvinte de evento em um elemento. Não há como remover todos os manipuladores de eventos vinculados de um elemento, a menos que tenhamos mantido referências a todos os manipuladores vinculados à medida que os criamos.

EVENT FIRING

Outro agravante é determinar se um evento será disparado. Embora seja possível determinar se um navegador oferece suporte a um meio de vincular um evento, é *não* possível saber se um navegador irá disparar um evento. Isso se torna problemático em alguns lugares.

Primeiro, se um script for carregado dinamicamente após a página já ter sido carregada, o script pode tentar vincular um ouvinte para aguardar o carregamento da janela quando, na verdade, esse evento já aconteceu. Como não há como determinar se o evento já ocorreu, o código pode ficar esperando uma eternidade para ser executado.

A segunda situação ocorre se um script deseja usar eventos personalizados fornecidos por um navegador como alternativa. Por exemplo, o Internet Explorer fornece `mouseenter` e `mouseleave` eventos, que simplificam o processo de determinar quando o mouse de um usuário entra ou sai dos limites de um elemento. Estes são frequentemente usados como alternativas para o `mouseover` e `mouseout` eventos, porque agem um pouco mais intuitivamente do que os eventos padrão. Mas, como não há como determinar se esses eventos serão disparados sem primeiro vinculá-los e aguardar a interação do usuário com eles, é difícil usá-los em código reutilizável.

EFEITOS DE PROPRIEDADE CSS

Outro ponto problemático é determinar se a alteração de certas propriedades CSS afeta a apresentação. Algumas propriedades CSS afetam apenas a representação visual da tela e nada mais; eles não mudam os elementos circundantes ou afetam outras propriedades no elemento. Exemplos são `color`, `backgroundColor`, e `opacity`.

Não há como determinar programaticamente se a alteração dessas propriedades de estilo gerará os efeitos desejados. A única maneira de verificar o impacto é por meio de um exame visual da página.

BROWSER CRASHES

Testar script que faz com que o navegador trave é outro aborrecimento. O código que causa o travamento de um navegador é especialmente problemático, porque, ao contrário das exceções que podem ser facilmente capturadas e tratadas, elas sempre farão com que o navegador falhe.

Por exemplo, em versões mais antigas do Safari (consulte <http://bugs.jquery.com/ticket/1331>), criar uma expressão regular que usasse intervalos de caracteres Unicode sempre faria com que o navegador travasse, como no exemplo a seguir:

```
novo RegExp ("[\ w \ u0128- \ uFFFF * _] +");
```

O problema com isso é que não é possível testar se esse problema existe, porque o próprio teste sempre produzirá uma falha no navegador mais antigo.

Além disso, os bugs que causam travamentos para sempre ficam envolvidos em dificuldades, porque embora possa ser aceitável que o JavaScript seja desabilitado em algum segmento da população que usa seu navegador, nunca é aceitável travar completamente o navegador desses usuários.

APIS INCONGRUOSOS

Um tempo atrás, vimos que o jQuery decidiu proibir a capacidade de alterar o modelo atributo em todos os navegadores devido a um bug no Internet Explorer. Nós *poderia* teste esse recurso e desative-o apenas no IE, mas isso configuraria uma incongruência, já que a API funcionaria de maneira diferente de navegador para navegador. Nessas situações, quando um bug é tão grave que causa a quebra de uma API, a única opção é contornar a área afetada e fornecer uma solução diferente.

Além de problemas impossíveis de testar, alguns problemas são *possível* para testar, mas são proibitivamente difíceis de testar com eficácia. Vejamos alguns deles.

DESEMPENHO API

Às vezes, APIs específicas são mais rápidas ou mais lentas em navegadores diferentes. Ao escrever código reutilizável e robusto, é importante tentar usar as APIs que fornecem bom desempenho. Mas nem sempre é óbvio qual API é.

A realização eficaz da análise de desempenho de um recurso geralmente envolve o lançamento de uma grande quantidade de dados nele, e isso normalmente leva um tempo relativamente longo. Portanto, não é algo que podemos fazer sempre que nossa página é carregada.

Recursos não testáveis são um aborrecimento significativo que atrapalham a escrita de JavaScript reutilizável, mas frequentemente podemos contorná-los com um pouco de esforço e inteligência. Usando técnicas alternativas ou construindo nossas APIs para evitar esses problemas em primeiro lugar, provavelmente seremos capazes de construir um código eficaz, apesar das probabilidades contra nós.

14.4 Reduzindo suposições

Escrever código reutilizável em vários navegadores é uma batalha de suposições, mas usando detecção e criação inteligentes, podemos reduzir o número de suposições que fazemos em nosso código. Quando fazemos suposições sobre o código que escrevemos, podemos encontrar problemas mais adiante.

Por exemplo, supor que um problema ou bug sempre existirá em um navegador específico é uma suposição enorme e perigosa. Em vez disso, testar o problema (como fizemos ao longo deste capítulo) prova ser muito mais eficaz. Em nossa codificação, devemos sempre nos esforçar para reduzir o número de suposições, reduzindo efetivamente a margem de erro e a probabilidade de que algo vai voltar e nos deixar por trás.

A área mais comum para fazer suposições em JavaScript é a detecção do agente do usuário - especificamente, analisar o agente do usuário fornecido por um navegador (navegador

. agente de usuário) e usá-lo para fazer uma suposição sobre como o navegador se comportará (em outras palavras, detecção do navegador). Infelizmente, a maioria das análises de string de agente de usuário prova ser uma excelente fonte de erros induzidos no futuro. Assumir que um bug, problema ou recurso proprietário sempre estará vinculado a um navegador específico é uma receita para o desastre.

Mas a realidade intervém quando se trata de minimizar suposições: é virtualmente impossível remover todas elas. Em algum ponto, teremos que assumir que um navegador fará o que deve fazer. Descobrir onde encontrar esse equilíbrio depende totalmente do desenvolvedor, e é o que “separa os homens dos meninos”, como eles dizem (com desculpas às nossas leitoras).

Por exemplo, vamos reexaminar o código de anexação de evento que já vimos neste capítulo:

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

Sem olhar para frente, veja se você consegue identificar três suposições feitas por este código. Vá em frente, vamos esperar. (*Perigo* o tema joga ...)

Como você está? O código anterior tem pelo menos estas três suposições:

- As propriedades que estamos verificando são funções que podem ser chamadas.
- São as funções corretas, realizando as ações que esperamos.
- Esses dois métodos são as únicas maneiras possíveis de vincular um evento.

Poderíamos facilmente nos livrar da primeira suposição adicionando verificações para ver se as propriedades são, de fato, funções. Lidar com os dois pontos restantes é muito mais difícil.

Neste código, sempre precisamos decidir quantas premissas são ideais para nossos requisitos, nosso público-alvo e nós. Frequentemente, reduzir o número de suposições também aumenta o tamanho e a complexidade da base de código. É totalmente possível, e bastante fácil, tentar reduzir as suposições ao ponto da completa insanidade, mas em algum ponto temos que parar e fazer um balanço do que temos, dizer “bom o suficiente” e trabalhar a partir daí. Lembre-se de que mesmo o código que menos supõe ainda está sujeito a regressões introduzidas por um navegador.

14.5 Resumo

- Embora a situação tenha melhorado consideravelmente, os navegadores infelizmente não estão isentos de bugs e geralmente não suportam os padrões da web de forma consistente.
- Ao escrever aplicativos JavaScript, a escolha de quais navegadores e plataformas suportar é uma consideração importante.
- Como não é possível oferecer suporte a todas as combinações, a qualidade nunca deve ser sacrificada pela cobertura!
- Os maiores desafios para escrever código JavaScript que pode ser executado em vários navegadores são correções de bugs, regressões, bugs do navegador, recursos ausentes e código externo.
- O desenvolvimento reutilizável em vários navegadores envolve vários fatores:
 - *Tamanho do código* —Mantendo o tamanho do arquivo pequeno
 - *Sobrecarga de desempenho* —Mantendo o nível de desempenho acima de um mínimo palatável
 - *Qualidade API* —Certificando-se de que as APIs funcionam uniformemente em todos os navegadores

- Não existe uma fórmula mágica para determinar o equilíbrio correto desses fatores. Os fatores de
- desenvolvimento são algo que deve ser equilibrado por cada desenvolvedor em seus esforços de desenvolvimento individual.
- Usando técnicas inteligentes, como detecção de recursos, podemos nos defender contra as inúmeras direções a partir das quais o código reutilizável será atacado sem fazer sacrifícios indevidos.

14.6 Exercícios

- 1 O que devemos levar em consideração ao decidir quais navegadores oferecer suporte?
- 2 Explique o problema de IDs gananciosos.
- 3 O que é detecção de recurso?
- 4 O que é um polyfill do navegador?

apêndice A

Recursos ES6 adicionais

Este apêndice cobre

- Literais de modelo
- Destruição
- Aprimoramentos literais de objeto

Este apêndice cobre alguns dos recursos “menores” do ES6 que não se encaixam perfeitamente nos capítulos anteriores. *Literais de modelo* permitir a interpolação de strings e strings de várias linhas, *desestruturação* nos permite extrair facilmente dados de objetos e matrizes, e

literals de objeto aprimorados melhorar o relacionamento com, bem, literais de objeto.

Literais de modelo

Literais de modelo são um novo recurso do ES6 que torna a manipulação de strings muito mais agradável do que antes. Basta pensar no passado; quantas vezes você foi forçado a escrever algo tão feio quanto isso?

```
const ninja = {
  nome: "Yoshi",
  ação: "subterfúgio"
};

const concatMessage = "Nome:" + ninja.name + ""
  + "Ação:" + ninja.action;
```

Neste exemplo, temos que construir uma string com dados inseridos dinamicamente. Para conseguir isso, temos que recorrer a algumas concatenações complicadas. Mas não mais! No ES6, podemos obter o mesmo resultado com literais de modelo; basta dar uma olhada na lista a seguir.

Listagem A.1 Literais de modelo

```
const ninja = {
  nome: "Yoshi",
  ação: "subterfúgio"
};

const concatMessage = "Nome:" + ninja.name + ""
  + "Ação:" + ninja.action;

const templateMessage = `Nome: ${ninja.name} Ação: ${ninja.action}`;

assert (concatMessage === templateMessage,
  "Nossas mensagens correspondem");
```

Usa crases para criar literais de modelo que podem conter expressões JavaScript encapsuladas em \$ {}

Como você pode ver, o ES6 fornece um novo tipo de string que usa crases (`), uma string que pode conter marcadores de posição, denotados com a sintaxe \$ {}. Dentro desses espaços reservados, podemos colocar qualquer expressão JavaScript: uma variável simples, um acesso de propriedade de objeto (como fizemos com `ninja.action`), e até mesmo chamadas de função.

Quando uma string de modelo é avaliada, os marcadores de posição são substituídos pelo resultado da avaliação da expressão JavaScript contida nesses marcadores.

Além disso, os literais de modelo não estão limitados a uma única linha (assim como os padrões entre aspas duplas e simples), e não há nada que nos impeça de torná-los multilinhas, como mostrado na lista a seguir.

Listagem A.2 Literais de modelo multilinha

```
nome const = "Yoshi", ação = "subterfúgio"; const multilineString =
`Nome: ${name}
Yoshi: ${action}`;
```

Strings de modelo não são limitado a uma única linha.

Agora que fornecemos uma breve introdução aos literais de template, vamos dar uma olhada em outro recurso do ES6: *desestruturação*.

Destruição

A desestruturação nos permite extrair facilmente dados de objetos e arrays usando padrões. Por exemplo, imagine que você tem um objeto cujas propriedades deseja atribuir a algumas variáveis, como na listagem a seguir.

Listagem A.3 Destruturando objetos

```
const ninja = {nome: "Yoshi", ação: "skulk", arma: "shuriken"};
```

```
const nameOld = ninja.name; const actionOld = ninja.action;  
const weaponOld = ninja.weapon;
```

Método antigo: temos que atribuir explicitamente cada propriedade do objeto a uma variável.

```
const {nome, ação, arma} = ninja;
```

```
afirmar (nome === "Yoshi", "Nosso ninja Yoshi"); assert (action === "skulk", "is skulk"); afirmar  
(arma === "shuriken", "com uma shuriken");
```

Desestruturação de objetos: nós pode atribuir cada propriedade a uma variável com o mesmo nome, tudo de uma vez.

```
const {nome: meuNome, ação: minhaAção, arma: minhaArme} = ninja;
```

```
assert (myName === "Yoshi", "Nosso ninja Yoshi"); assert (myAction === "skulk", "is skulk"); assert  
(myWeapon === "shuriken", "com um shuriken");
```

Se necessário, podemos nomear explicitamente as variáveis para as quais queremos atribuir valores.

Como mostra a listagem A.3, com a desestruturação do objeto, podemos facilmente extrair várias variáveis de um literal de objeto, tudo de uma vez. Considere a seguinte declaração:

```
const {nome, ação, arma} = ninja;
```

Isso cria três novas variáveis (nome, ação, e arma) cujos valores são os valores das propriedades correspondentes do objeto no lado direito da instrução (ninja.name, ninja.action, e ninja.weapon, respectivamente).

Quando não queremos usar os nomes das propriedades do objeto, podemos ajustá-los, como na seguinte instrução:

```
const {nome: meuNome, ação: minhaAção, arma: minhaArme} = ninja;
```

Aqui, criamos três variáveis (myName, myAction, e minha arma) e atribuir a eles valores das propriedades do objeto especificado.

Anteriormente, mencionamos que também podemos desestruturar arrays, pois os arrays são apenas um tipo especial de objeto. Dê uma olhada na lista a seguir.

Listagem A.4 Destructuring arrays

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

```
const [primeiroNinja, segundoNinja, terceiroNinja] = ninjas;
```

```
assert (firstNinja === "Yoshi", "Yoshi é nosso primeiro ninja"); assert (secondNinja === "Kuma", "Kuma o  
segundo");
```

Os itens da matriz são, em ordem, atribuídos a variáveis especificadas.

```
assert (thirdNinja === "Hattori", "E Hattori the terceiro");
```

```
const [, terceiro] = ninjas;
```

```
assert (terceiro === "Hattori", "Podemos pular itens");
```

```
const [primeiro, ... restantes] = ninjas;
```

```
afirmar (primeiro === "Yoshi", "Yoshi é novamente nosso primeiro ninja"); assert (restante.length === 2, "Existem dois ninjas restantes");
```

```
afirmar (restante [0] === "Kuma", "Kuma é o primeiro ninja restante"); afirmar (restante [1] === "Hattori", "Hattori o segundo ninja restante");
```

Podemos pular alguns
itens da matriz.

Podemos capturar
itens à direita.

Destruir matrizes é um pouco diferente de desestruturar objetos, principalmente na sintaxe, porque as variáveis são colocadas entre colchetes (ao contrário de colchetes, que são usados para desestruturar objetos), conforme mostrado no fragmento a seguir:

```
const [primeiroNinja, segundoNinja, terceiroNinja] = ninjas;
```

Nesse caso, Yoshi, o primeiro ninja, é atribuído à variável `firstNinja`. Kuma é atribuído à variável `secondNinja`. Hattori é atribuído à variável `thirdNinja`.

A desestruturação de array também tem alguns usos avançados. Por exemplo, se quisermos pular certos itens, podemos omitir os nomes das variáveis, mantendo as vírgulas, como na seguinte instrução:

```
const [, terceiro] = ninjas;
```

Neste caso, os dois primeiros ninjas serão ignorados, enquanto o valor do terceiro ninja, Hattori, será atribuído à variável `terceiro`.

Além disso, podemos extrair apenas alguns itens, ao atribuir os itens restantes a uma nova matriz:

```
const [primeiro, ... restantes] = ninjas;
```

O primeiro item, Yoshi, é atribuído à variável `primeiro`, e os ninjas restantes, Kuma e Hattori, são atribuídos à nova matriz, remanescente. Observe que, neste caso, os itens restantes são marcados da mesma forma que os parâmetros restantes (o operador `...`).

Literais de objeto aprimorados

Uma das grandes coisas sobre JavaScript é sua facilidade de criar objetos com literais de objeto: definimos algumas propriedades e as envolvemos entre chaves e voilà, criamos um novo objeto. No ES6, a sintaxe literal do objeto ganhou algumas novas extensões. Vejamos um exemplo. Digamos que queremos criar um ninja objeto e atribua a ele uma propriedade com base no valor de uma variável que está no escopo, uma propriedade cujo nome é calculado dinamicamente e um método, conforme mostrado na lista a seguir.

Listagem A.5 Literais de objeto aprimorados

```
const name = "Yoshi";
const oldNinja = {
  nome nome,
```

Cria uma propriedade com o mesmo nome de uma
variável no escopo e atribui o valor dessa variável
a ela

```

    getName: function () {
      return this.name;
    }
  };

  oldNinja ["antigo" + nome] = verdadeiro;
  assert (oldNinja.name === "Yoshi", "Yoshi aqui");
  assert (typeof oldNinja.getName === "function", "with a method"); assert ("oldYoshi" em oldNinja, "e uma propriedade
  dinâmica");

  const newNinja = {
    nome,
    getName () {
      return this.name;
    },
    ["novo" + nome]: verdadeiro
  };

  assert (newNinja.name === "Yoshi", "Yoshi aqui, novamente"); assert (typeof newNinja.getName === "function", "with a
  method"); assert ("newYoshi" em newNinja, "e uma propriedade dinâmica");

```

Define um método em um objeto

Cria uma propriedade cujo nome é calculado dinamicamente

Sintaxe abreviada do valor da propriedade; atribui o valor da mesma variável nomeada à propriedade

Abreviação da definição do método; não há necessidade de adicionar dois pontos e a palavra-chave de função. Usar parênteses após o nome da propriedade sinaliza que estamos lidando com um método.

Um nome de propriedade computado

Este exemplo começa criando um oldNinja objeto usando a antiga sintaxe literal de objeto pré-ES6:

```

nome const = "Yoshi"; const oldNinja = {

  nome nome,
  getName: function () {
    return this.name;
  }
};

oldNinja ["antigo" + nome] = verdadeiro;

```

Comparamos isso com literais de objeto aprimorados que alcançam exatamente o mesmo efeito, com menos confusão sintática:

```

const newNinja = {
  nome,
  getName () {
    return this.name;
  },
  ["novo" + nome]: verdadeiro
};

```

Isso completa nossa exploração de novos conceitos importantes introduzidos pelo ES6.

Apêndice B

Armando com teste e depuração

Este apêndice cobre

- Ferramentas para depurar código JavaScript
- Técnicas para gerar testes
- Construindo um conjunto de testes
- Pesquisando alguns dos frameworks de teste populares

Este apêndice apresenta algumas técnicas fundamentais no desenvolvimento de aplicativos da Web do lado do cliente: depuração e teste. Construir suítes de teste eficazes para nosso código é sempre importante. Afinal, se não testarmos nosso código, como saberemos se ele faz o que pretendemos? O teste nos dá um meio de garantir que nosso código não apenas execute, mas execute *corretamente*.

Além disso, tão importante quanto uma estratégia de teste sólida é para *todo* código, pode ser crucial quando fatores externos têm o potencial de afetar a operação do nosso código, que

é *exatamente* o caso que enfrentamos no desenvolvimento de JavaScript em vários navegadores. Não temos apenas os problemas típicos de garantir a qualidade do código (especialmente ao lidar com vários desenvolvedores trabalhando em uma única base de código) e nos proteger contra regressões que podem quebrar partes de uma API (problemas genéricos que todos os programadores precisam enfrentar), mas também temos o problema de determinar se nosso código funciona em todos os navegadores que escolhemos oferecer suporte.

Neste capítulo, veremos ferramentas e técnicas para depurar código JavaScript, gerar testes com base nesses resultados e construir um conjunto de testes para executar esses testes de maneira confiável. Vamos começar.

Ferramentas de desenvolvedor web

Por muito tempo, o desenvolvimento de aplicativos JavaScript foi prejudicado pela falta de uma infraestrutura básica de depuração. A única maneira de depurar o código JavaScript era espalhar alertas declarações que nos notificariam sobre o valor da expressão alertada, em torno do código que estava agindo de forma estranha. Como você pode imaginar, isso tornou a depuração (difícilmente uma atividade divertida) ainda mais difícil.

Felizmente, o Firebug, uma extensão do Firefox, foi desenvolvido em 2007. Firebug ocupa um lugar especial no coração de muitos desenvolvedores da web, porque foi a primeira ferramenta que forneceu uma experiência de depuração que se equiparou a depuração de última geração ambientes de desenvolvimento integrado (IDEs), como Visual Studio ou Eclipse. Além disso, o Firebug inspirou o desenvolvimento de ferramentas de desenvolvedor semelhantes para todos os principais navegadores: Ferramentas de desenvolvedor F12, incluídas no Internet Explorer e Microsoft Edge; WebKit Inspector, incluído no Safari; Ferramentas de desenvolvedor do Firefox, incluídas no Firefox; e Chrome DevTools incluídos no Chrome e Opera. Vamos explorá-los um pouco.

FIREBUG

Firebug, a primeira ferramenta avançada de depuração de aplicativos da web, está disponível exclusivamente para Firefox e pode ser acessada pressionando a tecla F12 (ou clicando com o botão direito em qualquer lugar da página e selecionando Inspecionar elemento com Firebug). Você pode instalar o Firebug abrindo a página no Firefox (<https://getfirebug.com/>) e seguindo as instruções. Figura

B.1 mostra Firebug.

O Firebug oferece funcionalidades avançadas de depuração, algumas das quais foram pioneiras. Por exemplo, podemos explorar facilmente o estado atual do DOM usando o painel HTML (o painel mostrado na figura B.1), executar o código JavaScript personalizado dentro do contexto da página atual usando o console (parte inferior da figura B.1), explore o estado do nosso código JavaScript usando o painel Script e até explore as comunicações de rede a partir do painel Net.

FIREFOX DEVELOPER TOOLS

Além do Firebug, se você for um usuário do Firefox, poderá usar o Firefox DevTools integrado, mostrado na figura B.2. Como você pode ver, a aparência geral das ferramentas de desenvolvedor do Firefox é semelhante ao Firebug (além de alguns layout e rótulo menores

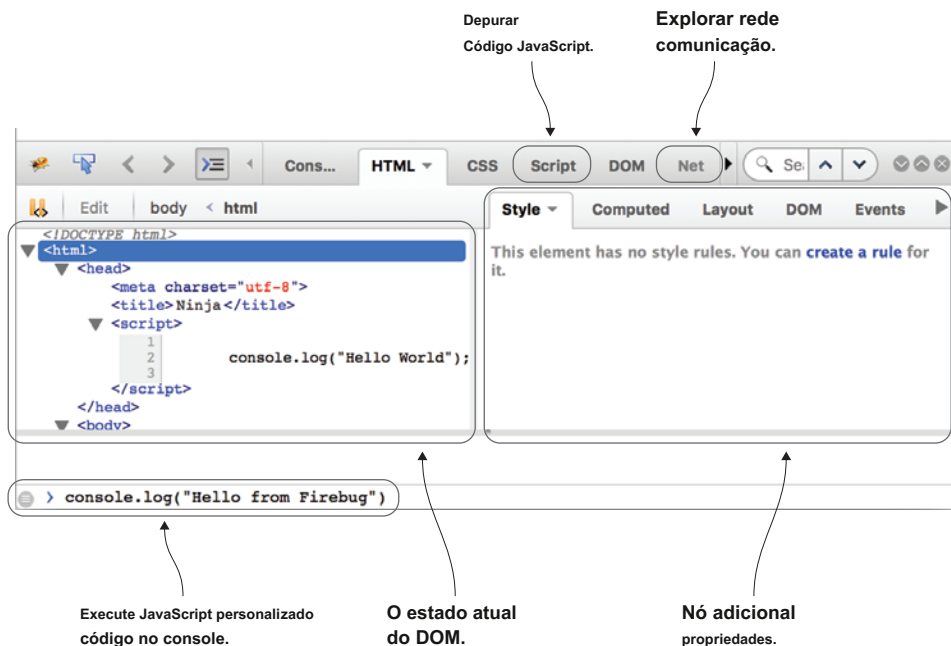


Figura B.1 O Firebug, disponível apenas no Firefox, foi a primeira ferramenta de depuração avançada para aplicativos da web.

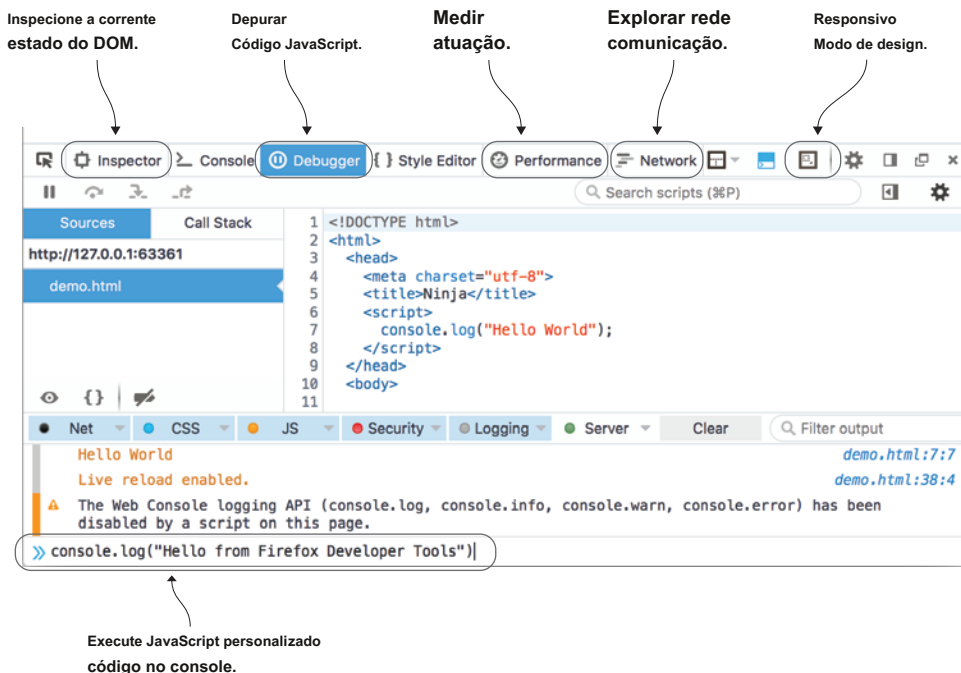


Figura B.2 As ferramentas de desenvolvedor do Firefox, integradas ao Firefox, oferecem todos os recursos do Firebug e mais alguns.

diferenças; por exemplo, o painel HTML do Firebug é denominado Inspector nas ferramentas de desenvolvedor do Firefox).

As ferramentas de desenvolvedor do Firefox são construídas pela equipe da Mozilla, que aproveitou essa integração com o Firefox trazendo alguns recursos úteis adicionais. O painel Desempenho, por exemplo, fornece uma visão detalhada sobre o desempenho de nossos aplicativos da web. Além disso, as ferramentas de desenvolvedor do Firefox são construídas com a web moderna em mente. Por exemplo, eles oferecem o modo de design responsivo, que nos ajuda a explorar a aparência de nossos aplicativos da web em diferentes tamanhos de tela, o que é algo com que devemos ter cuidado, porque hoje em dia os usuários acessam os aplicativos da web não apenas de seus PCs, mas também de dispositivos móveis, tablets e até TVs.

F12 DEVELOPER TOOLS

Se você está no campo do Internet Explorer (IE), ficará feliz em saber que o IE e o Microsoft Edge (o sucessor do IE) oferecem suas próprias ferramentas de desenvolvedor, as ferramentas de desenvolvedor F12. (Rapidamente, tente adivinhar qual tecla as ativa e desativa.) Essas ferramentas são mostradas na figura B.3.

Novamente, observe as semelhanças entre as ferramentas de desenvolvedor F12 e as ferramentas de desenvolvedor do Firefox (com apenas pequenas diferenças nos rótulos). As ferramentas F12 também nos permitem explorar o estado atual do DOM (o painel DOM Explorer, figura B.3), executar o código JavaScript personalizado por meio do console, depurar nosso código JavaScript (o painel Depurador), analisar o tráfego de rede (Rede), lidar com design responsivo (UI Responsiveness) e analisar o desempenho e o consumo de memória (Profiler e Memory).

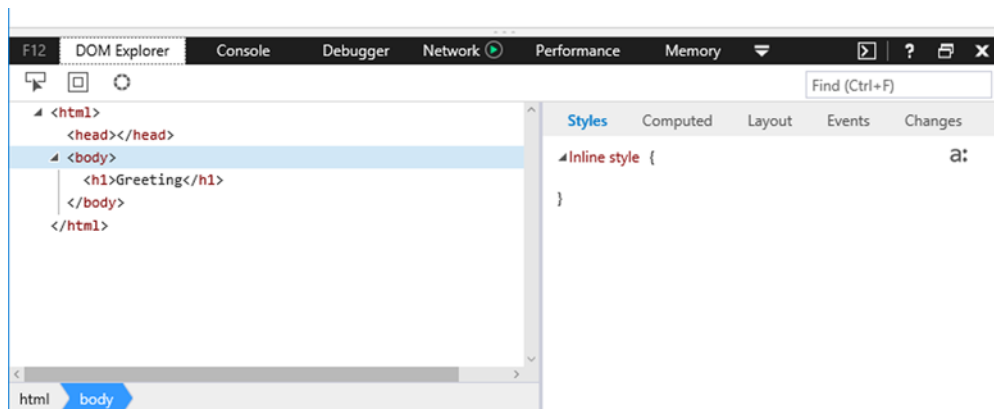


Figura B.3 As ferramentas de desenvolvedor F12 (alternadas pressionando F12) estão disponíveis no Internet Explorer e no Edge.

C EB K INSPETOR DE TI

Se você for um usuário do OS X, pode usar o WebKit Inspector, que é oferecido pelo Safari, conforme mostrado na figura B.4. Embora a IU do WebKit Inspector do Safari seja um pouco diferente

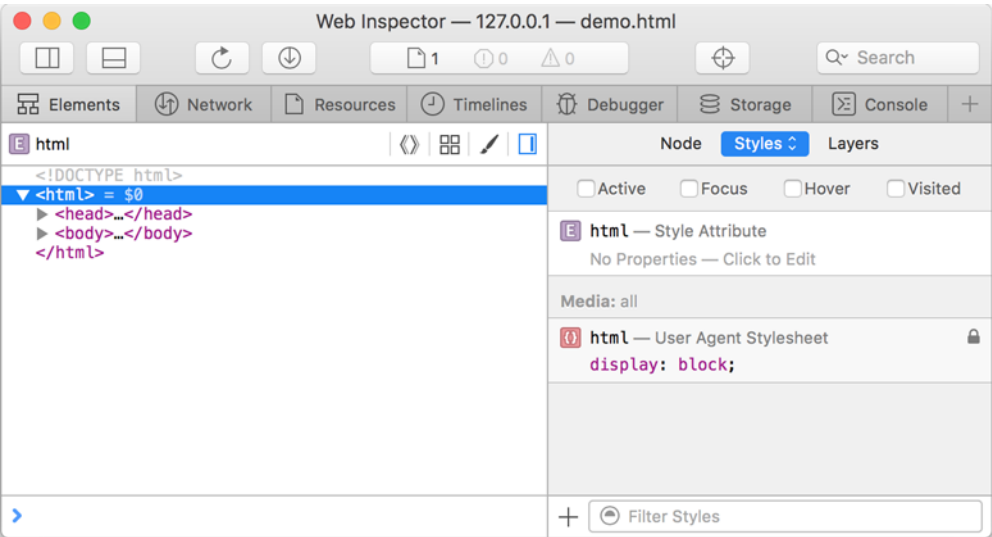


Figura B.4 WebKit Inspector, disponível no Safari

das ferramentas de desenvolvedor F12 ou ferramentas de desenvolvedor do Firefox, tenha certeza de que o WebKit Inspector também oferece suporte a todos os recursos de depuração importantes.

C HROME D EV T OOLS

Concluiremos nossa pequena pesquisa de ferramentas para desenvolvedores com o Chrome DevTools - em nossa opinião, o carro-chefe atual das ferramentas para desenvolvedores de aplicativos da web que está gerando muitas inovações ultimamente. Como você pode ver na figura B.5, a IU e os recursos básicos são semelhantes ao restante das ferramentas do desenvolvedor.

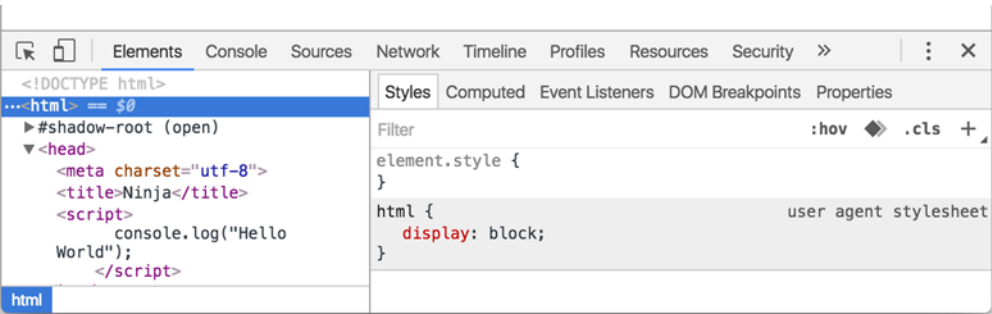


Figura B.5 Chrome DevTools, disponível no Chrome e Opera

Ao longo deste livro, usamos Chrome DevTools, por uma questão de convenção. Mas, como você viu ao longo desta seção, a maioria das ferramentas do desenvolvedor oferece recursos semelhantes (e se uma delas oferece algo novo, as outras o atualizam rapidamente). Você pode usar facilmente as ferramentas de desenvolvedor oferecidas pelo navegador de sua escolha.

Agora que você teve uma introdução às ferramentas que pode usar para depurar código, vamos explorar algumas técnicas de depuração.

Código de depuração

Uma parte significativa do tempo de desenvolvimento de software é gasta na remoção de bugs irritantes. Embora isso às vezes possa ser interessante, quase como resolver um mistério polêmico, normalmente queremos nosso código funcionando corretamente e sem bugs o mais rápido possível.

Depurar JavaScript tem dois aspectos importantes:

- *Exploração madeireira*, que imprime o que está acontecendo enquanto nosso código está sendo executado
- *Breakpoints*, que nos permitem pausar temporariamente a execução de nosso código e explorar o estado atual do aplicativo

Ambos são úteis para responder à pergunta importante, "O que está acontecendo em meu código?" mas cada um o aborda de um ângulo diferente. Vamos começar examinando o registro.

Exploração madeireira

Exploração madeireira instruções são usadas para enviar mensagens durante a execução do programa, sem impedir o fluxo normal do programa. Quando adicionamos instruções de log ao nosso código (por exemplo, usando o `console.log`), nos beneficiamos de ver as mensagens no console do navegador. Por exemplo, se quisermos saber o valor de uma variável chamada `x` em certos pontos da execução do programa, podemos escrever algo como a lista a seguir.

Listagem B.1 Registrando o valor da variável `x` em vários pontos de execução do programa

```
<!DOCTYPE html>
1: <html>
2:   <head>
3:     <title> Registro </title>
4:     <script>
5:       var x = 213;
6:       console.log ("O valor de x é:", x);
7:
8:       x = "Olá" + "Mundo";
9:       console.log ("O valor de x agora é:", x); </script>
10:
11:   </head>
12: <body> </body>
13: </html>
```

A Figura B.6 mostra o resultado da execução desse código no navegador Chrome com o console JavaScript habilitado.

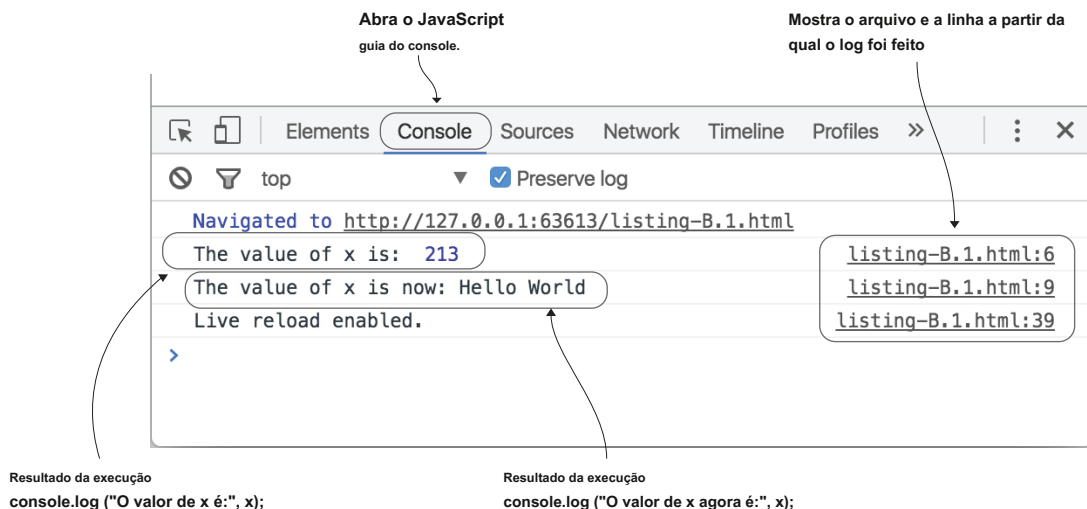


Figura B.6 O registro nos permite ver o estado de nosso código enquanto ele está sendo executado. Nesse caso, podemos ver que o valor de 213 é registrado na linha 6 e o valor de "Hello World" na linha 9 da listagem B.1. Todas as ferramentas de desenvolvedor, incluindo Chrome DevTools mostradas aqui, têm uma guia Console para fins de registro.

Como você pode ver, o navegador registra as mensagens diretamente no console JavaScript, mostrando a mensagem registrada e a linha na qual a mensagem foi registrada.

Este é um exemplo simples de registro de um valor de uma variável em diferentes pontos de execução do programa. Mas, em geral, você pode usar o registro para explorar várias facetas de seus aplicativos em execução, como a execução de funções importantes, a alteração de uma propriedade de objeto importante ou a ocorrência de um evento específico.

O registro é muito bom para ver o estado das coisas enquanto o código está sendo executado, mas às vezes queremos parar a ação e dar uma olhada. É aí que entram os pontos de interrupção.

Breakpoints

Usando *pontos de interrupção* podem ser mais complexos do que o registro, mas possuem uma vantagem notável: eles interrompem a execução de um script em uma linha de código específica, parando o navegador. Isso nos permite investigar vagarosamente o estado de todos os tipos de coisas no momento do intervalo.

Digamos que temos uma página que registra uma saudação a um ninja famoso, conforme mostrado na lista a seguir.

Listagem B.2 Uma página simples de "cumprimentar um ninja"

```
<!DOCTYPE html>
<html>
  <head>
    <title> Saudação Ninja </title>
  <script>
    function logGreeting (name) {
      console.log ("Saudações ao grande" + nome);
```

```

    }
    var ninja = "Hattori Hanzo";
    logGreeting (ninja);
  </script>
</head>
<body>
</body>
</html>

```

← Linha onde quebraremos

Digamos que definimos um ponto de interrupção usando o Chrome DevTools na linha anotada que chama o logGreeting função na listagem B.2 (clcando na medianiz do número da linha no painel Depurador) e atualize a página para fazer com que o código seja executado. O depurador então pararia a execução nessa linha e nos mostraria a tela na figura B.7.

O painel à direita mostra o estado do aplicativo no qual nosso código está sendo executado, incluindo o valor do ninja variável (Hattori Hanzo). O depurador quebra em uma linha *antes* a linha de breakpoint é executada; neste exemplo, a chamada para o logGreeting função ainda não foi executada.

ENTRANDO EM UMA FUNÇÃO

Se estivermos tentando depurar um problema com nosso logGreeting função, podemos querer *entrar* essa função para ver o que está acontecendo dentro dela. Enquanto nossa execução está pausada no logGreeting chamada (com um ponto de interrupção que definimos anteriormente), clicamos no botão Step Into (mostrado como uma seta apontando para um ponto na maioria dos depuradores) ou pressionamos F11, o que fará com que o depurador seja executado até a primeira linha de nosso logGreeting função. A Figura B.8 mostra o resultado.

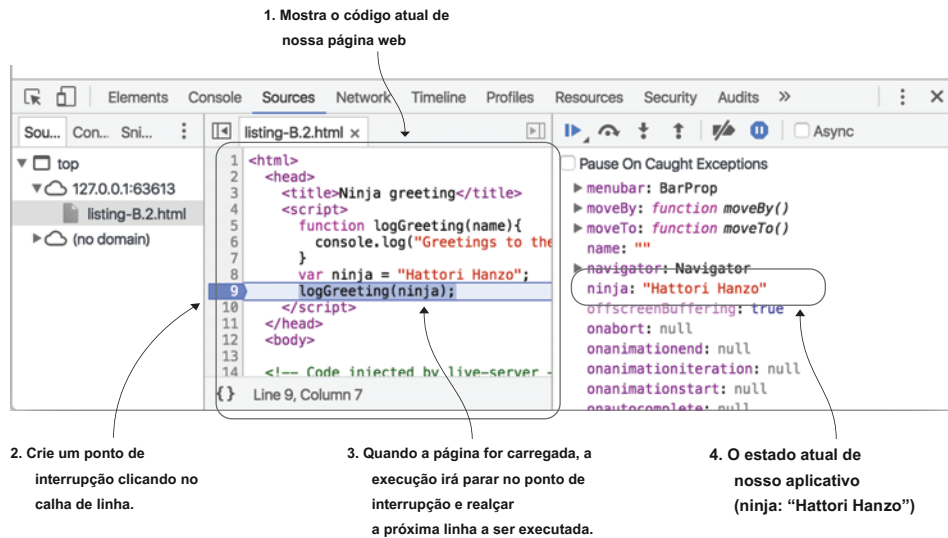


Figura B.7 Quando definimos um ponto de interrupção em uma linha de código (clcando na medianiz da linha) e carregamos a página, o navegador para de executar o código JavaScript antes que essa linha seja executada. Em seguida, você pode explorar vagorosamente o estado atual do aplicativo no painel à direita.

Observe que a aparência do Chrome DevTools mudou um pouco (em comparação com a figura B.7) para nos permitir vasculhar o estado do aplicativo em que o `logGreeting` a função é executada. Por exemplo, agora podemos explorar facilmente as variáveis locais de nosso `logGreeting` função e ver se temos um nome variável com o valor `Hattori Hanzo` (os valores das variáveis são mesmo mostrados em linha, com o código-fonte à esquerda). Observe também que no canto superior direito está um painel `Call Stack`, que mostra que estamos atualmente no `logGreeting` função, que foi chamada pelo código global.

Avançar e sair

Além do comando `Step Into`, podemos usar `Step Over` e `Step Out`.

O comando `Step Over` executa nosso código linha por linha. Se o código na linha executada contiver uma chamada de função, o depurador passa por cima da função (a função será executada, mas o depurador não saltará para seu código).

Se tivermos pausado a execução de uma função, clicar no botão `Step Out` executará o código até o final da função, e o depurador fará uma nova pausa logo após a execução ter deixado essa função.

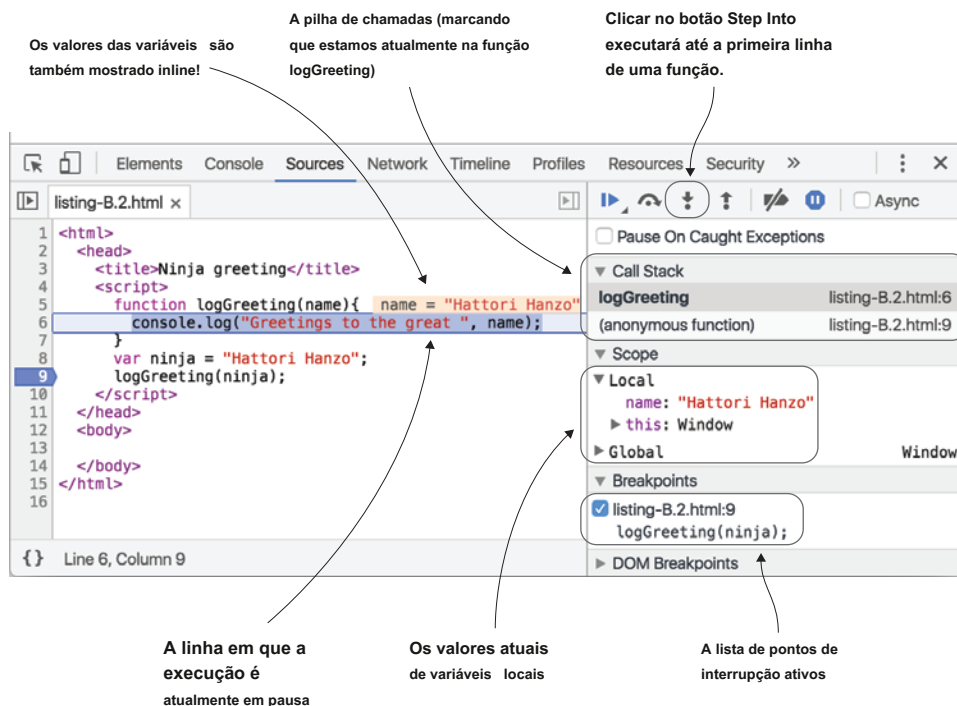


Figura B.8 Entrar em uma função nos permite ver o novo estado em que a função é executada. Podemos explorar a posição atual estudando o `Call Stack` e os valores atuais das variáveis locais.

C PONTOS DE INTERVALO ONDICIONAIS

Os pontos de interrupção padrão fazem com que o depurador pare a execução do aplicativo sempre que um depurador atingir esse ponto específico na execução do programa. Em certos casos, isso pode ser cansativo. Considere a seguinte lista.

Listagem B.3 Contando Ninjas e pontos de interrupção condicionais

```
<!DOCTYPE html>
<html>
  <head>
    <script>
      para (var i = 0; i < 100; i++) {
        console.log("Ninjas:" + i);
      }
    </script>
  </head>
  <body>
  </body>
</html>
```

E se quisermos explorar o estado do aplicativo ao contar o 50° ninja? Devemos ter que passar tediosamente pelos primeiros 49?

Imagine que queremos explorar o estado do aplicativo ao contar o 50° ninja. Seria cansativo ter que visitar todos os 49 ninjas antes de finalmente chegar ao que queremos?

Bem-vindo aos pontos de interrupção condicionais! Ao contrário dos pontos de interrupção tradicionais, que param toda vez que a linha do ponto de interrupção é executada, um *pontos de interrupção condicional* faz com que o depurador seja interrompido apenas se uma expressão associada aos pontos de interrupção condicionais for satisfeita. Você pode definir um ponto de interrupção condicional clicando com o botão direito do mouse na sarjeta do número da linha e escolhendo Adicionar (consulte a figura B.9 para saber como isso é feito no Chrome).

Ao associar a expressão: `i == 49` com um ponto de interrupção condicional, o depurador irá parar apenas quando essa condição for satisfeita. Dessa forma, podemos pular imediatamente para o ponto da execução do aplicativo em que estamos interessados e ignorar os menos interessantes.

Até agora, você viu como usar várias ferramentas de desenvolvedor de diferentes navegadores para depurar nosso código com registro e pontos de interrupção. Todas essas são ótimas ferramentas que

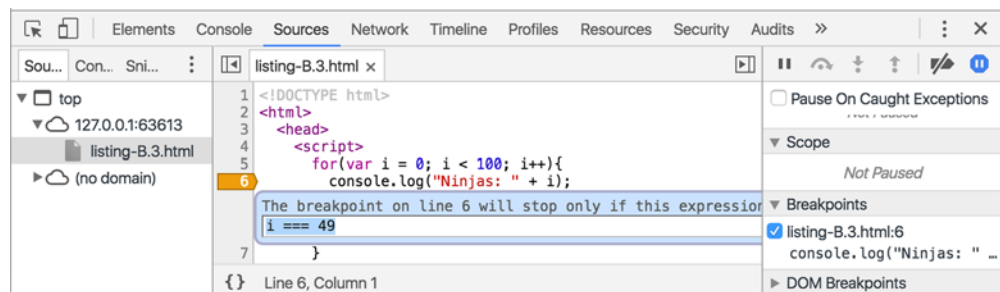


Figura B.9 Clique com o botão direito na margem do número da linha para definir um ponto de interrupção como condicional; observe que eles são mostrados em uma cor diferente, geralmente laranja.

ajude-nos a localizar bugs específicos e a compreender melhor a execução de um aplicativo específico. Mas, além disso, queremos ter uma infraestrutura que nos ajude a detectar bugs o mais rápido possível. Isso pode ser alcançado com testes.

Criação de testes

Robert Frost escreveu que “boas cercas fazem bons vizinhos”, mas no mundo dos aplicativos da web e, na verdade, em qualquer disciplina de programação, bons testes são bons códigos. Observe a ênfase na palavra *Boa*. É possível ter um conjunto de testes extenso que não ajuda em nada a qualidade do nosso código se os testes forem mal construídos.

Bons testes exibem três características importantes:

- *Repetibilidade* —Nossos resultados de teste devem ser altamente reproduzíveis. Os testes executados repetidamente sempre devem produzir exatamente os mesmos resultados. Se os resultados do teste não forem determinísticos, como saberemos quais resultados são válidos e quais são inválidos? Além disso, a reprodutibilidade garante que nossos testes não dependam de fatores externos, como cargas de rede ou CPU.
- *Simplicidade* —Nossos testes devem se concentrar em testar *1* coisa. Devemos nos esforçar para remover o máximo de marcação HTML, CSS ou JavaScript possível, sem interromper a intenção do caso de teste. Quanto mais removemos, maior a probabilidade de que o caso de teste seja influenciado apenas pelo código específico que estamos testando.
- *Independência* —Nossos testes devem ser executados isoladamente. Devemos evitar tornar os resultados de um teste dependentes de outro. Dividir os testes nas menores unidades possíveis nos ajudará a determinar a origem exata de um bug quando ocorre um erro.

Podemos usar várias abordagens para construir testes. As duas abordagens principais são desconstrutivas e construtivas:

- *Casos de teste desconstrutivo* —O código existente é reduzido (desconstruído) para isolar um problema, eliminando qualquer coisa que não seja pertinente ao problema. Isso ajuda a atingir as três características listadas anteriormente. Podemos começar com um site completo, mas depois de remover a marcação extra, CSS e JavaScript, chegaremos a um caso menor que reproduz o problema.
- *Casos de teste construtivos* —Começamos de um caso reduzido e bom conhecido e continuamos até sermos capazes de reproduzir o bug em questão. Para usar esse estilo de teste, precisaremos de alguns arquivos de teste simples a partir dos quais construir testes e uma maneira de gerar esses novos testes com uma cópia limpa de nosso código.

Vejamos um exemplo de teste construtivo.

Ao criar casos de teste reduzidos, podemos começar com alguns arquivos HTML com funcionalidade mínima já incluída neles. Podemos até ter arquivos iniciais diferentes para várias áreas funcionais; por exemplo, um para manipulação de DOM, um para testes Ajax, um para animações e assim por diante.

Por exemplo, a lista a seguir mostra um caso de teste DOM simples usado para testar o jQuery.

Listagem B.4 Um caso de teste DOM reduzido para jQuery

```
<style>
#teste {largura: 100px; altura: 100px; fundo: vermelho; } </style>

<div id = "teste"> </div>
<script src = "dist / jquery.js"> </script>
<script>
$(document).ready(function () {
    $("#teste").append("teste");
});
</script>
```

Outra alternativa é usar um serviço pré-construído projetado para criar casos de teste simples, por exemplo JSFiddle (<http://jsfiddle.net/>), CodePen (<http://codepen.io/>) ou JS Bin (<http://jsbin.com/>) Todos têm funcionalidade semelhante; eles nos permitem construir casos de teste que se tornam disponíveis em um URL exclusivo. (E você pode até incluir cópias de bibliotecas populares.) Um exemplo no JSFiddle é mostrado na figura B.10.

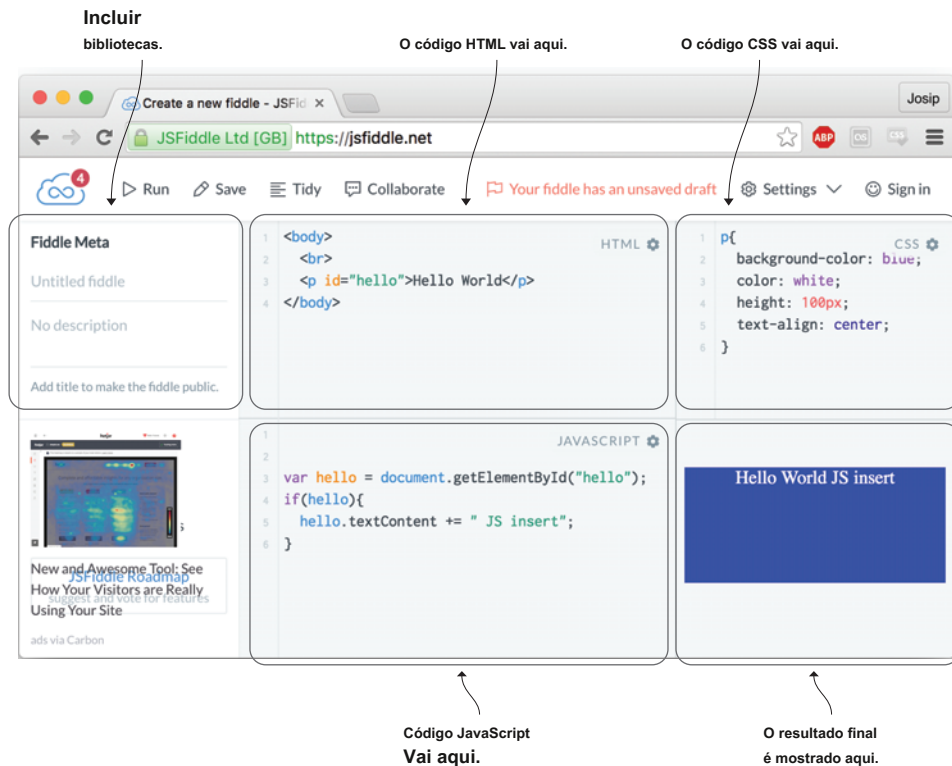


Figura B.10 O JSFiddle nos permite testar combinações de trechos de HTML, CSS e JavaScript em uma sandbox para ver se tudo funciona como planejado.

Usar o JSFiddle (ou ferramentas semelhantes) é muito bom e prático quando temos que fazer um teste rápido de um determinado conceito, especialmente porque você pode compartilhá-lo facilmente com outras pessoas, e talvez até obter algum feedback útil. Infelizmente, a execução de tais testes requer que você abra manualmente o teste e verifique seu resultado, o que pode ser bom se você tiver apenas alguns testes, mas normalmente devemos ter muitos e muitos testes que verificam cada canto e fissura de nosso código. Por este motivo, queremos automatizar nossos testes o máximo possível. Vamos ver como fazer isso.

Os fundamentos de uma estrutura de teste

O objetivo principal de uma estrutura de teste é permitir-nos especificar testes individuais que podem ser agrupados em uma única unidade, para que possam ser executados em massa, fornecendo um único recurso que pode ser executado de forma fácil e repetida.

Para entender melhor como funciona uma estrutura de teste, faz sentido observar como ela é construída. Talvez surpreendentemente, as estruturas de teste de JavaScript são fáceis de construir.

Você teria que perguntar, porém, "Por que eu iria querer construir uma nova estrutura de teste?" Na maioria dos casos, escrever sua própria estrutura de teste de JavaScript não é necessário, porque muitas outras de boa qualidade já estão disponíveis (como você verá em breve). Mas construir sua própria estrutura de teste pode servir como uma boa experiência de aprendizado.

A afirmação

O núcleo de uma estrutura de teste de unidade é o seu método de asserção, normalmente denominado afirmar. Este método geralmente leva um *valor* - uma expressão cuja premissa é *afirmou* - e uma descrição do propósito da afirmação. Se o valor for avaliado como verdade, a afirmação passa; caso contrário, é considerado um fracasso. A mensagem associada geralmente é registrada com um indicador de aprovação / reprovação apropriado.

Uma implementação simples desse conceito pode ser vista na lista a seguir.

Listagem B.5 Uma implementação simples de uma asserção JavaScript

```
<!DOCTYPE html>
<html>
  <head>
    <title> Test Suite </title>
    <script>
      função assert (valor, desc) {
        var li = document.createElement ("li");
        li.className = valor? "passa": "falha";
        li.appendChild (document.createTextNode (desc));
        document.getElementById ("resultados"). appendChild (li);
      }
      window.onload = function () {
        assert (true, "The test suite is running."); assert (false, "Fail!");
      };
    </script>
  </style>
```

**Define a afirmação
método**

**Executa testes
usando asserções**

```
# resultados li.pass {color: green; }
# resultados li.fail {color: red; } </style>
```

Define estilos
para resultados

```
</head>
<body>
  <ul id = "results"> </ul>
</body>
</html>
```

← Retém os resultados do teste

A função chamada afirmar é quase surpreendentemente simples. Isso cria um novo `` elemento contendo a descrição, atribui uma classe chamada passar ou falhou, dependendo do valor do parâmetro de asserção (valor), e anexa o novo elemento a um elemento de lista no corpo do documento.

O conjunto de testes consiste em dois testes triviais: um que sempre terá sucesso e outro que sempre falhará:

```
assert (true, "The test suite is running."); // Sempre passará
assert (false, "Fail!"); // Sempre falhará
```

Regras de estilo para o passar e falhou classes indicam visualmente sucesso ou fracasso usando cores.

O resultado da execução de nosso conjunto de testes no Chrome é mostrado na figura B.11.

GORJETA Se você está procurando algo rápido, você pode usar o integrado `console.assert ()` método (ver figura B.12).

Agora que construímos nossa própria estrutura de teste rudimentar, vamos conhecer algumas das estruturas de teste mais populares e amplamente disponíveis.

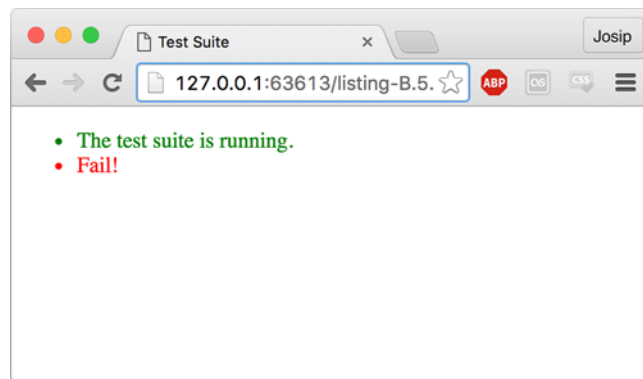


Figura B.11 O resultado da execução de nosso primeiro conjunto de testes

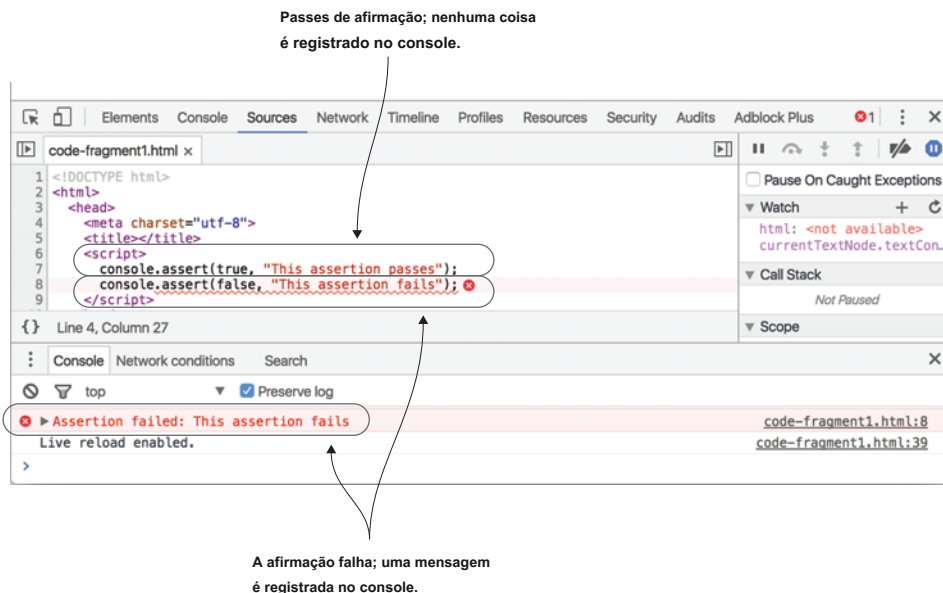


Figura B.12 Você pode usar o integrado **console.assert** como uma maneira rápida de testar o código. A mensagem de falha é registrada no console apenas se uma assertão falhar.

Estruturas de teste populares

Uma estrutura de teste deve ser uma parte fundamental de seu fluxo de trabalho de desenvolvimento, portanto, você deve escolher uma estrutura que funcione particularmente bem para seu estilo de codificação e sua base de código. Uma estrutura de teste de JavaScript deve atender a uma única necessidade: exibir os resultados dos testes e tornar mais fácil determinar quais testes foram aprovados ou reprovados. Os frameworks de teste podem nos ajudar a alcançar esse objetivo sem ter que nos preocupar com nada além de criar os testes e organizá-los em coleções chamadas *suites de teste*.

Existem vários recursos que podemos querer procurar em uma estrutura de teste de unidade JavaScript, dependendo das necessidades dos testes. Alguns desses recursos incluem o seguinte:

- A capacidade de simular o comportamento do navegador (cliques, pressionamentos de tecla e assim por diante)
- Controle interativo de testes (pausar e retomar testes)
- Tratamento de tempos limite de teste assíncrono
- A capacidade de filtrar quais testes devem ser executados

Vamos conhecer as duas estruturas de teste mais populares atualmente: QUnit e Jasmine.

QUNIT

QUnit é a estrutura de teste de unidade construída originalmente para testar o jQuery. Desde então, ele se expandiu além de seus objetivos iniciais e agora é uma estrutura de teste de unidade autônoma.

QUnit foi projetado principalmente para ser uma solução simples para testes de unidade, fornecendo uma API mínima, mas fácil de usar. As características distintivas do QUnit são as seguintes:

- API simples
- Suporta teste assíncrono
- Não limitado a código jQuery ou jQuery Especialmente
- adequado para testes de regressão

Vamos veja um exemplo de teste QUnit na lista a seguir que testa se nós desenvolvemos uma função que diz "Oi" com precisão para um ninja.

Listagem B.6 exemplo de teste QUnit

```
<!DOCTYPE html>
<html>
  <head>
    <link rel = "stylesheet" href = "qunit / qunit-git.css" />
    <script src = "qunit / qunit-git.js"> </script>
  </head>
  <body>
    <div id = "qunit"> </div>
  </body>
</html>
```

Inclui QUnit
código e estilos

Cria um elemento HTML que o QUnit
preenche com os resultados do teste

```
function sayHiToNinja (ninja) {
  retornar "Oi" + ninja;
}
```

Declara a função
que queremos testar

```
QUnit.test ("Ninja hello test", function (assert) {
  assert.ok (sayHiToNinja ("Hatori") == "Oi Hatori", "Aprovado"); assert.ok (falso, "Falha");
});
```

Especifica um
Caso de teste QUnit

Testa uma afirmação de passagem

Testa uma afirmação com falha

Ao abrir este exemplo em um navegador, você deve obter os resultados mostrados na figura B.13, com uma afirmação de passagem de execução da linha digaHiToNinja ("Hatori"), e uma afirmação falha de assert.ok (falso, "Falha").

Mais informações sobre o QUnit podem ser encontradas em <http://qunitjs.com/>.

J COMO O MEU

Jasmine é outra estrutura de teste popular, construída em bases ligeiramente diferentes do QUnit. As principais partes da estrutura são as seguintes:

- O descrever função, que descreve suítes de teste
- O isto função, que especifica testes individuais
- O Espero função, que verifica as afirmações individuais

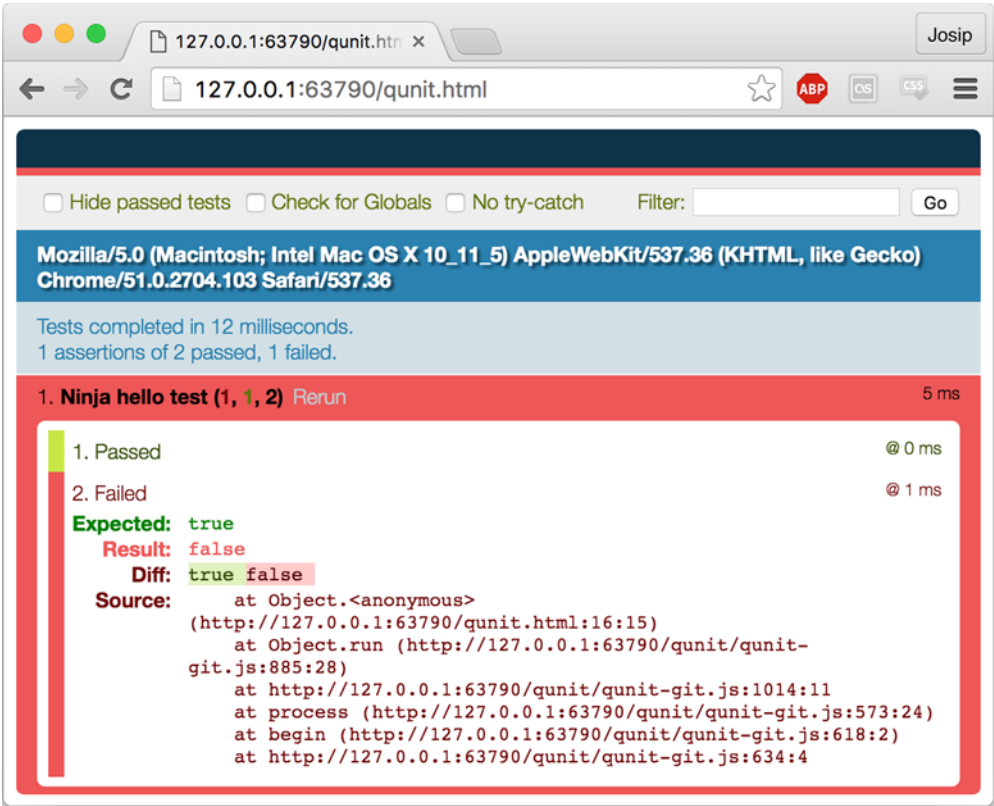


Figura B.13 Um exemplo de execução de teste QUnit. Como parte do nosso teste, temos uma afirmação que passa e uma que falha (uma das duas afirmações passou, uma falhou). Os resultados exibidos colocam uma ênfase muito maior no teste com falha, para garantir que o consertemos o mais rápido possível.

A combinação e a nomenclatura dessas funções são voltadas para tornar o conjunto de testes de natureza quase coloquial. Por exemplo, a lista a seguir mostra como testar o diga HiToNinja função usando Jasmine.

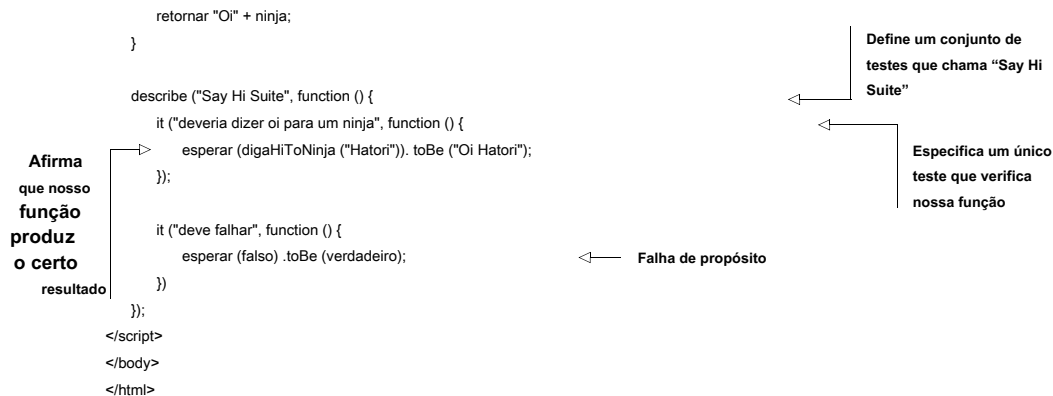
Listagem B.7 Exemplo de teste Jasmine

```
<!DOCTYPE html>
<html>
<head>
  <link rel = "stylesheet" href = "lib / jasmine-2.2.0 / jasmine.css">

  <script src = "lib / jasmine-2.2.0 / jasmine.js"> </script>
  <script src = "lib / jasmine-2.2.0 / jasmine-html.js"> </script> <script src = "lib / jasmine-2.2.0 / boot.js">
  </script>
</head>
<body>
<script>
  function sayHiToNinja (ninja) {
```

Inclui
Arquivos Jasmine

Declara a função
que queremos testar



O resultado da execução deste conjunto de testes Jasmine no navegador é mostrado na figura B.14.

Mais informações sobre Jasmine podem ser encontradas em <http://jasmine.github.io/>.

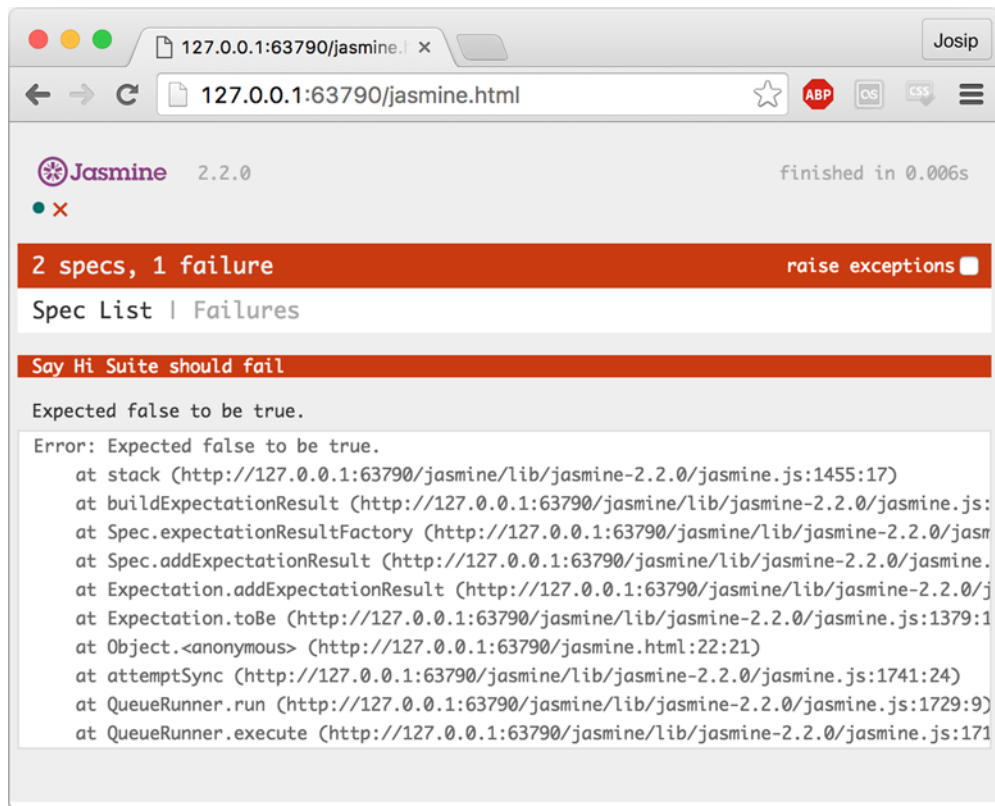


Figura B.14 O resultado da execução de um conjunto de testes Jasmine no navegador. Temos dois testes: um aprovado e outro reprovado (duas especificações, uma falha).

M COBERTURA DE CÓDIGO DE FACILIDADE

É difícil dizer o que torna um determinado conjunto de testes *Boa*. Idealmente, devemos testar todos os caminhos de execução possíveis de nossos programas. Infelizmente, exceto nos casos mais triviais, isso não é possível. Um passo na direção certa é tentar testar o máximo de código possível, e uma métrica que nos diz o grau em que um conjunto de testes cobre nosso código é chamada *Cobertura de código*.

Por exemplo, dizer que uma suíte de teste tem 80% de cobertura de código significa que 80% do código do nosso programa é executado pela suíte de teste, enquanto 20% do nosso código não é. Embora não possamos ter certeza absoluta de que 80% do código não contém bugs (podemos ter perdido um caminho de execução que leva a um), estamos completamente no escuro sobre os 20% que nem foram executados. É por isso que devemos medir a cobertura de código de nossos conjuntos de teste.

No desenvolvimento de JavaScript, podemos usar várias bibliotecas para medir a cobertura de nossos conjuntos de teste, mais notavelmente Blanket.js (<https://github.com/alex-seville/blanket>) e Istanbul (<https://github.com/gotwarlost/istanbul>) . A configuração dessas bibliotecas vai além do escopo deste livro, mas suas respectivas páginas da web oferecem todas as informações de que podemos precisar para configurá-las adequadamente.

apêndice C

Respostas de exercícios

Capítulo 2. Construindo a página no tempo de execução

- 1 Quais são as duas fases do ciclo de vida de um aplicativo da web do lado do cliente?

R: As duas fases do ciclo de vida de um aplicativo da Web do lado do cliente são a construção de páginas e o tratamento de eventos. Na fase de construção da página, a interface do usuário de nossa página é construída processando o código HTML e executando o código JavaScript da linha principal. Depois que o último nó HTML é processado, a página entra na fase de tratamento de eventos, na qual vários eventos são processados.

- 2 Qual é a principal vantagem de usar o `addEventListener` método para registrar um manipulador de eventos versus atribuir um manipulador a uma propriedade de elemento específico?

R: Ao atribuir manipuladores de eventos a propriedades de elementos específicos, podemos registrar apenas um manipulador de eventos; `addEventListener`, por outro lado, nos permite registrar quantos manipuladores de eventos forem necessários.

- 3 Quantos eventos podem ser processados de uma vez?

R: JavaScript é baseado em um modelo de execução de thread único, no qual os eventos são processados um de cada vez.

- 4 Em que ordem os eventos da fila de eventos são processados?

R: Os eventos são processados na ordem em que foram gerados: primeiro a entrar, primeiro a sair.

Capítulo 3. Funções de primeira classe para o novato: definições e argumentos

- 1 No trecho de código a seguir, quais funções são funções de retorno de chamada?

```
// sortAsc é um retorno de chamada porque o mecanismo JavaScript // o chama para comparar itens da matriz
numbers.sort (função sortAsc ( a, b) {
```

```

    retornar a - b;
  });

  // Não é um retorno de chamada; ninja é chamado como uma função de função padrão ninja () {}

  ninja ();

  var myButton = document.getElementById ("myButton"); // handleClick é um callback, a função é
  chamada // sempre que myButton é clicado

  myButton.addEventListener ("click", função handleClick () {
    alerta ("clicado");
  });

```

- 2 No snippet a seguir, categorize as funções de acordo com seu tipo (função declaração, expressão de função ou função de seta).

```

  // expressão da função como argumento para outra função numbers.sort (função sortAsc ( a, b ) {

    retornar a - b;
  });

  // função de seta como argumento para outra função numbers.sort (( a, b) => b - a);

  // expressão de função como o receptor em uma expressão de chamada ( função(){}());

  // declaração de função
  function outer () {
    // declaração de função
    função interna () {}
    retorno interno;
  }

  // chamada de expressão de função envolvida em uma expressão ( função(){}());

  // função de seta como callee (() => " Yoshi ") ();

```

- 3 Depois de executar o seguinte trecho de código, quais são os valores das variáveis samurai e ninja?

```

// "Tomoe", o valor do corpo da expressão da função seta var samurai = (() => "Tomoe") ();

// indefinido, no caso do corpo de uma função de seta ser uma instrução de bloco // o valor é o valor da instrução de retorno.

// Como não há instrução de retorno, o valor é indefinido. var ninja = (() => {"Yoshi"}) ();

```

- 4 Dentro do corpo do teste função, quais são os valores dos parâmetros a, b, e c para as duas chamadas de função?

```
teste de função (a, b, ... c) {/ * a, b, c * /}

// a = 1; b = 2; c = [3, 4, 5] teste (1, 2, 3, 4, 5);

// a = undefined; b = indefinido; c = [] teste ();
```

- 5 Depois de executar o seguinte snippet de código, quais são os valores do mensagem1 e mensagem2 variáveis?

```
function getNinjaWieldingWeapon (ninja, weapon = "katana") {
  retornar ninja + "" + katana;
}

// "Yoshi katana" - há apenas um argumento na chamada // então o padrão da arma é "katana"

var message1 = getNinjaWieldingWeapon ("Yoshi");

// "Yoshi wakizashi" - enviamos dois argumentos, o valor // padrão não é levado em consideração

var message2 = getNinjaWieldingWeapon ("Yoshi", "wakizashi");
```

Capítulo 4. Funções para o jornalista: entendendo a invocação de função

- 1 A função a seguir calcula a soma dos argumentos passados usando o argumentos objeto. Usando os parâmetros restantes introduzidos no capítulo anterior, reescreva o soma função para que não use o argumentos objeto.

```
function sum () {
  var sum = 0;
  para (var i = 0; i < argument.length; i++) {
    soma + = argumentos [i];
  }
  soma de retorno;
}

assert (sum (1, 2, 3) === 6, 'A soma dos três primeiros números é 6'); assert (sum (1, 2, 3, 4) === 10, 'A soma dos quatro primeiros números é 10');
```

R: Adicione um parâmetro de descanso à definição da função e ajuste ligeiramente o corpo da função:

```
soma de funções (... números){
  var sum = 0;
  para (var i = 0; i < números. comprimento; i++) {
    soma + = números[ eu];
  }
  soma de retorno;
}

assert (sum (1, 2, 3) === 6, 'A soma dos três primeiros números é 6'); assert (sum (1, 2, 3, 4) === 10, 'A soma dos quatro primeiros números é 10');
```

- 2 Depois de executar o seguinte código no navegador, quais são os valores das variáveis `ninja` e `samurai`?

```
function getSamurai (samurai) {
  "use estrito"

  argumentos [0] = "Ishida";

  retorno samurai;
}

function getNinja (ninja) {
  argumentos [0] = "Fuma";
  retornar ninja;
}

var samurai = getSamurai ("Toyotomi"); var ninja = getNinja ("Yoshi");
```

UMA: `samurai` terá o valor `Toyotomi`, e `ninja` terá o valor `Fuma`.

Porque o `getSamurai` função está em modo estrito, o `argumentos` parâmetro não é um alias de parâmetros de função, portanto, alterar o valor do primeiro argumento não mudará o valor do `samurai` parâmetro.

Porque o `getNinja` função está em modo não estrito, quaisquer alterações feitas no `argumentos` parâmetro será refletido nos parâmetros da função.

- 3 Ao executar o código a seguir, qual das afirmações será aprovada?

```
function whoAml1 () {
  "use estrito";
  devolva isso;
}

function whoAml2 () {
  devolva isso;
}

assert (whoAml1 () === window, "Window?"); // falhou
assert (whoAml2 () === window, "Window?"); // passar
```

R: O `whoAml1` a função está no modo estrito; quando é chamado como uma função, o valor do esta parâmetro será `Indefinido` (e não janela). A segunda afirmação será aprovada: Se uma função no modo não estrito for chamada como uma função, esta refere-se ao objeto global (o janela objeto, ao executar o código no navegador).

- 4 Ao executar o código a seguir, qual das afirmações será aprovada?

```
var ninja1 = {
  whoAml: function () {
    devolva isso;
  }
}
```

```

};

var ninja2 = {
  whoAml: ninja1.whoAml
};

var identificar = ninja2.whoAml;

// passar: whoAml chamado como um método de ninja1 assert (ninja1.whoAml () ===
ninja1, "ninja1?");

// falha: whoAml chamado como um método de ninja2 assert (ninja2.whoAml () === ninja1, "ninja1
novamente?");

// falha: identificar chamadas a função como uma função
// porque estamos no modo não estrito, isso se refere à declaração da janela (identifique () === ninja1, "ninja1
novamente?");

// passar: usando chamada para fornecer o contexto da função // isso se refere a ninja2

assert (ninja1.whoAml.call (ninja2) === ninja2, "ninja2 aqui?");

```

5 Ao executar o código a seguir, qual das afirmações será aprovada?

```

function Ninja () {
  this.whoAml = () => isso;
}

var ninja1 = novo Ninja (); var ninja2 = {

  whoAml: ninja1.whoAml
};

// pass: whoAml é uma função de seta herda o contexto da função // do contexto em que foi criado.

// Porque foi criado durante a construção do ninja1 // isso sempre apontará para ninja1

assert (ninja1.whoAml () === ninja1, "ninja1 aqui?");

// falso: sempre se refere a ninja1
assert (ninja2.whoAml () === ninja2, "ninja2 aqui?");

```

6 Qual das afirmações a seguir será aprovada?

```

function Ninja () {
  this.whoAml = function () {
    devolva isso;
  }.bind (este);
}

var ninja1 = novo Ninja (); var ninja2 = {

  whoAml: ninja1.whoAml
};

```

```
// passar: a função atribuída a whoAml é um limite de função
// para ninja1 (o valor deste quando o construtor foi chamado) // isso sempre se referirá a ninja1

assert (ninja1.whoAml () === ninja1, "ninja1 aqui?"); // falha: em whoAml sempre se refere a ninja1
// porque whoAml é uma função limitada.

assert (ninja2.whoAml () === ninja2, "ninja2 aqui?");
```

Capítulo 5. Funções para o mestre: fechamentos e escopos

- 1 Fechamentos permitem funções para

R: Acesse variáveis externas que estão no escopo quando a função é definida (opção a)

- 2 Fechamentos vêm com

R: Custos de memória (os fechamentos mantêm vivas as variáveis que estão no escopo quando a função é definida) (opção b)

- 3 No exemplo de código a seguir, marque os identificadores acessados por meio de fechamentos:

```
function Samurai (nome) {
  var arma = "katana";

  this.getWeapon = function () {
    // acessa a variável local: weapon return weapon;

  };

  this.getName = function () {
    // acessa o parâmetro da função: nome nome do retorno;

  }

  esta.mensagem = nome + "empunhando uma" + arma;

  this.getMessage = function () {
    // esta. mensagem não é acessada por meio de um fechamento // é uma propriedade do
    objeto (e não uma variável) return this.message;

  }
}

var samurai = novo Samurai ("Hattori");

samurai.getWeapon ();
samurai.getName ();
samurai.getMessage ();
```

- 4 No código a seguir, quantos contextos de execução são criados e qual é o maior tamanho da pilha de contexto de execução?

```
function perfom (ninja) {
  sneak (ninja);
  infiltrar (ninja);
```



```

    }

    função sneak (ninja) {
        retornar ninja + "esquiva";
    }

    função infiltrar (ninja) {
        retornar ninja + "infiltrando";
    }

    perform ("Kuma");

```

R: O maior tamanho de pilha é 3, nas seguintes situações:

- código global -> executar -> esgueirar
- código global -> executar -> infiltrar

- 5 Qual palavra-chave em JavaScript nos permite definir variáveis que não podem ser reatribuídas a um valor completamente novo?

UMA: const variáveis não podem ser reatribuídas a novos valores.

- 6 Qual é a diferença entre var e deixei?

R: A palavra-chave var é usado para definir apenas variáveis de escopo de função ou global, enquanto deixei nos permite definir variáveis de escopo de bloco, escopo de função e escopo global.

- 7 Onde e por que o código a seguir lançará uma exceção?

```

getNinja ();
getSamurai (); // lança uma exceção

function getNinja () {
    retornar "Yoshi";
}

var getSamurai = () => "Hattori";

```

R: Uma exceção será lançada ao tentar invocar o getSamurai função. O getNinja function é definida com uma declaração de função e será criada antes que qualquer código seja executado; podemos chamá-lo de "antes" de sua declaração ser alcançada em código. O getSamurai função, por outro lado, é uma função de seta criada quando a execução a alcança, portanto, será indefinida quando tentarmos invocá-la.

Capítulo 6. Funções para o futuro: geradores e promessas

- 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 ();

// 2 o primeiro valor gerado
deixe a1 = generator.next (). value;

// 4 o segundo valor gerado
deixe a2 = generator.next (). value;

// 2, porque iniciamos um novo gerador let a3 = EvenGenerator (). Next (). Value; // 6,
voltamos para o primeiro gerador let a4 = generator.next (). Value;
```

- 2 Qual é o conteúdo do ninjas array depois de executar o seguinte código? (Dica: Pense em 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;
```

R: O ninjas array irá conter apenas Yoshi. Isso acontece porque o para de loop itera sobre um gerador até que o gerador diga que está pronto (sem incluir o valor passado junto com done). Isso acontece quando não há mais código no gerador para executar ou quando um Retorna declaração é encontrada.

- 3 Qual é o valor das variáveis a1 e a2, depois de executar o seguinte código?

```

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

deixe gerador = Gen (2);

// 4. O valor do primeiro valor passado por next: 3 é ignorado // porque o gerador ainda não iniciou sua execução e // não há nenhuma
expressão de rendimento em espera.

// Como o gerador é criado com val sendo 2 // o primeiro rendimento ocorre para val * 2, ou seja, 2
* 2 == 4 let a1 = generator.next (3) .value;
```

```
// 5: passando 5 como um argumento para o próximo
// significa que a expressão produzida em espera obterá o valor 5 // (yield val * 2) == 5

// porque esse valor é então atribuído a val, a próxima expressão de rendimento // rendimento val;

// retornará 5
deixe a2 = generator.next (5) .value;
```

4 Qual é a saída do código a seguir?

```
promessa const = nova promessa ((resolver, rejeitar) => {
  rejeitar ("Hattori"); // a promessa foi rejeitada explicitamente});

// o manipulador de erros será invocado
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) => {
  // a promessa foi explicitamente resolvida resolve ("Hattori");

  // uma vez que uma promessa foi estabelecida, ela não pode ser alterada // rejeitá-la após 500
  ms não terá efeito setTimeout (() => rejeitar ("Yoshi", 500);

});

// o manipulador de sucesso será invocado promessa.then (val => alert ("Sucesso:" +
val))
  . catch (e => alert ("Erro:" + e));
```

Capítulo 7. Orientação a objetos com protótipos

1 Qual das seguintes propriedades aponta para um objeto que será pesquisado se o objeto de destino não tem a propriedade procurada? UMA: protótipo (opção c)

2 Qual é o valor da variável a1 depois que o código a seguir é executado?

```
função Ninja () {}
Ninja.prototype.talk = function () {
  retornar "Olá";
};

const ninja = novo Ninja ();
const a1 = ninja.talk (); // "Olá"
```

R: O valor da variável a1 será Olá. Mesmo que o objeto ninja não possui o conversar método, seu protótipo faz.

3 Qual é o valor de a1 depois de executar o seguinte código?

```

função Ninja () {}
Ninja.message = "Olá";

const ninja = novo Ninja ();

const a1 = ninja.mensagem;

```

R: O valor da variável a1 será indefinido. O mensagem propriedade é definida na função do construtor Ninja, e não é acessível através do ninja objeto.

- 4 Explique a diferença entre o getFullName método nestes dois códigos fragmentos:

```

// Primeiro fragmento
função Pessoa (primeiroNome, últimoNome) {
  this.firstName = primeiroNome;
  this.lastName = últimoNome;

  this.getFullName = function () {
    retorna this.firstName + " " + this.lastName;
  }
}

// segundo fragmento
função Pessoa (primeiroNome, últimoNome) {
  this.firstName = primeiroNome;
  this.lastName = últimoNome;
}

Person.prototype.getFullName = function () {
  retorna this.firstName + " " + this.lastName;
}

```

R: No primeiro fragmento, o getFullName método é definido diretamente na instância criada com o Pessoa construtor.

Cada objeto criado com o

Pessoa construtor obtém seu próprio getFullName método. No segundo fragmento, o getFullName método é definido no protótipo do Pessoa

função. Todas as instâncias criadas com o Pessoa função terá acesso a este único método.

- 5 Depois de executar o código a seguir, o que ninja.constructor aponta para?

```

function Person () {} function Ninja () {}

const ninja = novo Ninja ();

```

R: Ao acessar ninja.constructor, a propriedade constructor é encontrada em ninja protótipo de. Porque ninja foi criado com o Ninja função do construtor, a propriedade do construtor aponta para o Ninja função.

- 6 Depois de executar o código a seguir, o que `ninja.constructor` aponta para?

```
function Person () {} function Ninja () {}

Ninja.prototype = nova pessoa (); const ninja = novo Ninja ();
```

R: O construtor `property` é a propriedade do objeto de protótipo que foi criado com a função de construtor. Neste exemplo, substituímos o protótipo integrado do `Ninja` funcionar com um novo `Pessoa` objeto. Portanto, quando um `ninja` objeto é criado com o `Ninja` construtor, seu protótipo é definido para o novo `pessoa` objeto. Finalmente, quando acessamos o construtor `property` no `ninja` objeto, porque o `ninja` objeto não tem seu próprio construtor `property`, seu protótipo, o novo `pessoa` objeto, é consultado. O `pessoa` objeto também não tem um construtor `property`, então seu protótipo, o `Person.prototype` objeto, é consultado. Esse objeto tem um construtor `property`, referenciando o `Pessoa`

função. Este exemplo ilustra perfeitamente por que devemos ter cuidado ao usar o construtor `property`: Mesmo que nosso `ninja` objeto foi criado com o `Ninja` função, o construtor `property`, devido ao soluço de substituir o padrão `Ninja.prototype`, aponta para o `Pessoa` função.

- 7 Explique como o instancia de operador funciona no exemplo a seguir.

```
função Guerreiro () {}

função Samurai () {}
Samurai.prototype = novo Guerreiro ();

var samurai = novo Samurai ();

exemplo de samurai do guerreiro; //Explicar
```

R: O instancia de O operador verifica se o protótipo da função do lado direito está na cadeia de protótipos do objeto do lado esquerdo. O objeto à esquerda é criado com o `Samurai` função, e seu protótipo tem um novo `Guerreiro` objeto, cujo protótipo é o protótipo do `Guerreiro`

função (`Warrior.prototype`). À direita temos o `Guerreiro` função. Então, neste exemplo, o instancia de operador irá retornar verdade, porque o protótipo da função à direita, `Warrior.prototype`, pode ser encontrado na cadeia de protótipos do objeto à esquerda.

- 8 Traduza o seguinte código ES6 para o código ES5.

```
class Guerreiro {
  constructor (arma) {
    this.weapon = arma;
  }

  manejar() {
```

```

        return "Wielding" + this.weapon;
    }

    duelo estático (guerreiro1, guerreiro2) {
        return warrior1.wield () + "" + warrior2.wield ();
    }
}

```

R: Podemos traduzir o código da seguinte maneira:

```

função Guerreiro (arma) {
    this.weapon = arma;
}

Warrior.prototype.wield = function () {
    return "Wielding" + this.weapon;
};

Warrior.duel = function (warrior1, warrior2) {
    return warrior1.wield () + "" + warrior2.wield ();
};

```

Capítulo 8. Controlando o acesso aos objetos

- 1 Depois de executar o código a seguir, qual das seguintes expressões lançará uma exceção e por quê?

```

const ninja = {
    obter nome () {
        retornar "Akiyama";
    }
}

```

R: Ligando `ninja.name ()` lança uma exceção porque `ninja` não tem um método de nome (opção a).

Acessando `ninja.name` no `const name = ninja.name`

Funciona como um encanto; o getter é ativado e o nome da variável obtém o valor `Akiyama`.

- 2 No código a seguir, qual mecanismo permite que os getters acessem um objeto privado variável?

```

function Samurai () {
    const _weapon = "katana";
    Object.defineProperty (this, "weapon", {
        obter: () => _weapon});
}

Const samurai = novo Samurai ();
assert (samurai.weapon === "katana", "Um samurai empunhando uma katana");

```

R: Os fechamentos permitem que os getters acessem as variáveis do objeto privado. Neste caso, o obter método cria um fechamento em torno do `_ arma` variável privada definida na função construtora, que mantém o `_ arma` variável viva.

3 Qual das afirmações a seguir será aprovada?

```
const daimyo = {nome: "Matsu", clã: "Takasu"}; const proxy = new Proxy (daimyo, {

  obter: (alvo, chave) => {
    if (chave === "clã") {
      retornar "Tokugawa";
    }
  }
});

assert (daimyo.clan === "Takasu", "Matsu do clã Takasu"); assert (proxy.clan === "Tokugawa", "Matsu do clã Tokugawa"); // passar

proxy.clan = "Tokugawa";

assert (daimyo.clan === "Takasu", "Matsu do clã Takasu"); // falhou
assert (proxy.clan === "Tokugawa", "Matsu do clã Tokugawa?"); // passar
```

R: A primeira afirmação é aprovada porque daimyo tem um clã propriedade com valor Takasu. A segunda afirmação passa, porque acessamos a propriedade clã por meio de um proxy com uma armadilha get que sempre retorna Tokugawa como o valor do clã propriedade.

Quando a expressão `proxy.clan = "Tokugawa"` é avaliado, o valor Tokugawa é armazenado no daimyo de clã propriedade porque o proxy não tem uma armadilha definida, então a ação padrão de definir a propriedade é realizada no destino, daimyo objeto.

A terceira afirmação falha, porque o daimyo a propriedade do clã de tem o valor Tokugawa e não Takasu.

A quarta asserção é aprovada, porque o proxy sempre retorna Tokugawa, independentemente do valor armazenado no objeto de destino clã propriedade.

4 Qual das afirmações a seguir será aprovada?

```
const daimyo = {nome: "Matsu", clã: "Takasu", armySize: 10000}; const proxy = new Proxy (daimyo, {

  definir: (alvo, chave, valor) => {
    if (key === "armySize") {
      número const = Number.parseInt (valor); if (! Number.isNaN (number)) {

        alvo [chave] = número;
      }
    }
    } outro {
      alvo [chave] = valor;
    }
  },
});
```

```
//passar
assert (daimyo.armySize === 10000, "Matsu tem 10.000 homens armados"); //passar

assert (proxy.armySize === 10000, "Matsu tem 10.000 homens armados");

proxy.armySize = "grande";
assert (daimyo.armySize === "grande", "Matsu tem um grande exército"); // falhou

daimyo.armySize = "grande";
assert (daimyo.armySize === "grande", "Matsu tem um grande exército"); // passar
```

R: A primeira afirmação é aprovada; o valor de daimyo de armySize propriedade é 10.000.

A segunda afirmação também é aprovada; o proxy não tem um get trap definido, então o valor do alvo, daimyo de armySize propriedade, é retornado.

Quando a expressão proxy.armySize = "grande"; é avaliada, a armadilha definida do proxy é ativada. O configurador verifica se o valor passado é um número e, somente se for, o valor é atribuído à propriedade do destino. Nesse caso, o valor passado não é um número, portanto, nenhuma alteração é feita no armySize propriedade. Por esse motivo, a terceira asserção, que pressupõe a mudança, falha.

A expressão daimyo.armySize = "grande"; escreve diretamente para o armySize propriedade, ignorando o proxy. Portanto, a afirmação final é aprovada.

Capítulo 9. Lidando com cobranças

- 1 Depois de executar o código a seguir, qual é o conteúdo do samurai variedade?

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

R: O valor do samurai é [" Oda ", " Tomoe ", indefinido, " Hattori "]. O

array começa com Oda e Tomoe em índices 0 e 1 Em seguida, adicionamos um novo samurai, Hattori, no índice 3, que "expande" a matriz e deixa o índice 2 com unde multado.

- 2 Depois de executar o código a seguir, qual é o conteúdo do ninjas variedade?

```
const ninjas = [];

ninjas.push ("Yoshi");
ninjas.unshift ("Hattori");

ninjas.length = 3;

ninjas.pop ();
```

R: O valor de ninjas é [" Hattori ", " Yoshi "]. Começamos com uma matriz vazia, Empurre adiciona Yoshi até o fim, e não mudar adiciona Hattori para o início.

Definindo explicitamente comprimento para 3 expande a matriz com Indefinido no índice 2 Chamando pop remove isso Indefinido da matriz, deixando apenas [" Hattori "," Yoshi "].

- 3 Depois de executar o código a seguir, qual é o conteúdo do samurai variedade?

```
const samurai = [];

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

R: O valor de samurai é [" Tomoe "," Hattori "," Takeda "]. A matriz começa vazio; Empurre adiciona Oda até o fim, e não mudar adiciona Tomoe para o início; emenda remove o item no índice 1 (Oda) e adiciona Hattori e Takeda em vez de.

- 4 Depois de executar o código a seguir, o que está armazenado nas variáveis primeiro segundo, e terceiro?

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

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

A: primeiro: [18, 19, 20]; segundo: [18, 20]; terceiro: 57

- 5 Depois de executar o código a seguir, o que está armazenado nas variáveis primeiro e segundo?

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

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

A: primeiro: verdadeiro; segundo: falso

- 6 Qual das afirmações a seguir será aprovada?

```
const samuraiClanMap = new Map ();

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

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

```

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

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

assert (samuraiClanMap.has (samurai3), "O terceiro samurai tem um mapeamento");

```

R: A primeira afirmação falha, porque um mapeamento para samurai2 foi criado duas vezes. A segunda afirmação é aprovada, porque um mapeamento para samurai1 foi adicionado. E a terceira afirmação falha, porque um mapeamento para samurai3 nunca foi criado.

7 Qual das afirmações a seguir será aprovada?

```

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

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

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

```

R: A primeira afirmação falha porque Akiyama é adicionado apenas uma vez ao conjunto. A segunda afirmação também falha, porque tentar adicionar Akiyama oncemore não mudará o conjunto (nem seu comprimento). As duas últimas afirmações serão aprovadas.

Capítulo 10. *Wrangling expressões regulares*

- 1 Em JavaScript, as expressões regulares podem ser criadas com qual das seguintes opções? R: Literais de expressão regular (opção a) e usando o integrado RegExp construtor (opção B). A resposta c está incorreta; não embutido Expressão regular construtor existe.

- 2 Qual das opções a seguir é um literal de expressão regular?

R: Em JavaScript, um literal de expressão regular é colocado entre duas barras: /teste/ (opção a).

- 3 Escolha os sinalizadores de expressão regular corretos.

R: Com literais de expressão regular, os sinalizadores de expressão são colocados após a barra de fechamento: / teste / g (opção a). Com RegExp construtores, eles são passados como um segundo argumento: novo RegExp ("teste", "gi"); (opção c). A expressão regular / def / corresponde a qual das

- 4 seguintes strings? R: A expressão regular / def / jogos apenas def: *d* seguido pela *e*, seguido pela *f*

(opção b).

- 5 A expressão regular / [^ abc]/ corresponde a qual das seguintes?