

OREILLY

Seventh
Edition

JavaScript O Guia Definitivo

Domine a linguagem de programação
mais usada do mundo



David Flanagan

1. Prefácio. Convenções Usadas neste Livro.

Exemplo de Codec. O'Reilly Online Learning.

Como nos Contatar. Agradecimentos

2. Introdução ao JavaScripta. 1.1 Explorando JavaScriptb. 1.2

Olá Worldc. 1.3 Um tour pelo JavaScriptd. 1.4 Exemplo:

histograma de frequência de caracteres. 1.5 Resumo

3. Estrutura Lexicala. 2.1 O texto de um programa

JavaScriptb. 2.2 Comentáriosc. 2.3 Literalmente.

2.4 Identificadores e Palavras Reservadas

i. 2.4.1 Palavras reservadas. 2.5

Unicódigo

i. 2.5.1 Sequências de escape

Unicodeii. 2.5.2 Normalização Unicode

f. 2.6 Semicolons opcionais

g. 2.7 Resumo

4 tipos, valores e variáveis

um. 3.1 Visão geral e definições

b. 3.2 números

eu. 3.2.1 literais inteiros

ii. 3.2.2 Literais de ponto flutuante

iii. 3.2.3 aritmética em javascript

4. 3.2.4 Ponto flutuante binário e erros
de arredondamento

v. 3.2.5 inteiros de precisão arbitrária com
bigint

vi. 3.2.6 Datas e horários

c. 3.3 Texto

eu. 3.3.1 Literais de cordas

ii. 3.3.2 Sequências de fuga em literais de cordas

iii. 3.3.3 Trabalhando com strings

4. 3.3.4 Literais de modelo

v. 3.3.5 correspondência de padrões

d. 3.4 valores booleanos

e. 3.5 NULL e indefinido

f. 3.6 Símbolos

g. 3.7 O objeto global

h. 3.8 Valores primitivos imutáveis e referências de
objeto mutável

eu. 3.9 Conversões de tipo

eu. 3.9.1 Conversões e igualdade

ii. 3.9.2 Conversões explícitas

iii. 3.9.3 Objeto de conversões primitivas

j. 3.10 Declaração e atribuição variáveis

eu. 3.10.1 declarações com Let and Const

ii. 3.10.2 declarações variáveis com VAR

iii. 3.10.3 Atribuição de destruição

k. 3.11 Resumo

5. Expressões e operadores

um. 4.1 Expressões primárias

b. 4.2 Inicializadores de objeto e matriz

c. 4.3 Expressões de definição de função

d. 4.4 Expressões de acesso à propriedade

eu. 4.4.1 Acesso à propriedade condicional

e. 4.5 Expressões de invocação

eu. 4.5.1 Invocação condicional

f. 4.6 Expressões de criação de objetos

g. 4.7 Visão geral do operador

eu. 4.7.1 Número de operandos

ii. 4.7.2 Operando e tipo de resultado

iii. 4.7.3 Efeitos colaterais do operador

4. 4.7.4 Precedência do operador

v. 4.7.5 Associatividade do operador

vi. 4.7.6 Ordem de avaliação

h. 4.8 Expressões aritméticas

eu. 4.8.1 O operador +

ii. 4.8.2 Operadores aritméticos unários

iii. 4.8.3 Operadores bitwise

eu. 4.9 Expressões relacionais

eu. 4.9.1 Operadores de igualdade e desigualdade

ii. 4.9.2 Operadores de comparação

iii. 4.9.3 O operador no

4. 4.9.4 A instância do operador

j. 4.10 Expressões lógicas

eu. 4.10.1 lógico e (&&)

ii. 4.10.2 Lógico ou (||)

iii. 4.10.3 Lógico não (!)

k. 4.11 Expressões de atribuição

eu. 4.11.1 Atribuição com operação

l. 4.12 Expressões de avaliação

eu. 4.12.1 Eval ()

ii. 4.12.2 Avaliação global()

iii. 4.12.3 Avaliação estrita()

m. 4.13 Operadores Diversos

eu. 4.13.1 O Operador Condicional (?:)

ii. 4.13.2 Definido em Primeiro Lugar (??)

iii. 4.13.3 O tipo de Operador

4. 4.13.4 O operador de exclusão

v. 4.13.5 O operador de espera

vi. 4.13.6 O operador vazio

vii. 4.13.7 O Operador vírgula (,)

n. 4.14 Resumo

6. Declarações

um. 5.1 Declarações de Expressão

b. 5.2 Declarações Compostas e Vazias

c. 5.3 Condicionais

eu. 5.3.1 se

ii. 5.3.2 senão se

iii. 5.3.3 interruptor

d. 5.4 Laços

eu. 5.4.1 enquanto

ii. 5.4.2 fazer/enquanto

iii. 5.4.3 para

4. 5.4.4 para/de

v. 5.4.5 para/em

e. 5.5 Saltos

eu. 5.5.1 Declarações Rotuladas

ii. 5.5.2 pausa

iii. 5.5.3 continuar

4. 5.5.4 retorno

v. 5.5.5 rendimento

vi. 5.5.6 lançamento

vii. 5.5.7 tentar/capturar/finalmente

f. 5.6 Declarações Diversas

eu. 5.6.1 com

ii. 5.6.2 depurador

iii. 5.6.3 “uso estrito”

g. 5.7 Declarações

eu. 5.7.1 const, let e var

ii. 5.7.2 função

iii. 5.7.3 aula

4. 5.7.4 importação e exportação

h. 5.8 Resumo das declarações JavaScript

7. Objetos

um. 6.1 Introdução aos Objetos

b. 6.2 Criando objetos

eu. 6.2.1 Literais de objeto

ii. 6.2.2 Criando objetos com novo

iii. 6.2.3 Protótipos

4. 6.2.4 Object.Create ()

c. 6.3 Propriedades de consulta e definição

eu. 6.3.1 Objetos como matrizes associativas

ii. 6.3.2 Herança

iii. 6.3.3 Erros de acesso à propriedade

d. 6.4 Excluindo propriedades

e. 6.5 Propriedades de teste

f. 6.6 Propriedades de enumeração

eu. 6.6.1 Ordem de enumeração da propriedade

g. 6.7 estendendo objetos

h. 6.8 Objetos serializados

eu. 6.9 Métodos de objeto

eu. 6.9.1 O método ToString ()

ii. 6.9.2 O método tolocalestring ()

iii. 6.9.3 o método ValueOf ()

4. 6.9.4 O método Tojson ()

j. 6.10 Sintaxe literal de objeto estendido

eu. 6.10.1 Propriedades abreviadas

ii. 6.10.2 Nomes de propriedades computadas

iii. 6.10.3 Símbolos como nomes de propriedades

4. 6.10.4 Operador de espalhamento

v. 6.10.5 Métodos de abreviação

vi. 6.10.6 Getters de propriedades e setters

k. 6.11 Resumo

8. Matrizes

um. 7.1 Criando matrizes

eu. 7.1.1 Literais da matriz

ii. 7.1.2 O operador de propagação

iii. 7.1.3 O construtor Array ()

4. 7.1.4 Array.of ()

v. 7.1.5 Array.From ()

b. 7.2 Elementos de matriz de leitura e escrita

c. 7.3 Matrizes esparsas

d. 7.4 Comprimento da matriz

e. 7.5 Adicionando e excluindo elementos de matriz

f. 7.6 Matrizes de iteração

g. 7.7 Matrizes multidimensionais

h. 7.8 Métodos de matriz

eu. 7.8.1 Métodos de iterador de matriz

ii. 7.8.2 Matrizes achatadas com plano () e
plangmap ()

iii. 7.8.3 Adicionando arrays com concat()

4. 7.8.4 Pilhas e filas com push(), pop(), shift() e unshift()

v. 7.8.5 Submatrizes com slice(), splice(), fill() e copyWithin()

vi. 7.8.6 Métodos de busca e classificação de array

vii. 7.8.7 Conversões de array para string

viii. 7.8.8 Funções de matriz estática

eu. 7.9 Objetos semelhantes a array

j. 7.10 Strings como Matrizes

k. 7.11 Resumo

9. Funções

um. 8.1 Definindo Funções

eu. 8.1.1 Declarações de Função

ii. 8.1.2 Expressões de Função

iii. 8.1.3 Funções de seta

4. 8.1.4 Funções aninhadas

b. 8.2 Invocando Funções

eu. 8.2.1 Invocação de Função

ii. 8.2.2 Invocação de Método

iii. 8.2.3 Invocação do Construtor

4. 8.2.4 Invocação Indireta

v. 8.2.5 Invocação de função implícita

c. 8.3 Argumentos e parâmetros de função

eu. 8.3.1 parâmetros e padrões opcionais

ii. 8.3.2 Parâmetros de descanso e listas de argumentos de comprimento variável

iii. 8.3.3 O objeto de argumentos

4. 8.3.4 O operador de propagação para chamadas de função

v. 8.3.5 Argumentos de função de destruição em parâmetros

vi. 8.3.6 Tipos de argumento

d. 8.4 Funções como valores

eu. 8.4.1 Definindo suas próprias propriedades de função

e. 8.5 Funções como namespaces

f. 8.6 fechamentos

g. 8.7 Propriedades, métodos e construtores da função

eu. 8.7.1 A propriedade de comprimento

ii. 8.7.2 A propriedade Nome

iii. 8.7.3 A propriedade do protótipo

4. 8.7.4 Os métodos Call () e Apply ()

v. 8.7.5 O método bind ()

vi. 8.7.6 O método ToString ()

vii. 8.7.7 O construtor function ()

h. 8.8 Programação funcional

eu. 8.8.1 Matrizes de processamento com funções

ii. 8.8.2 Funções de ordem superior

iii. 8.8.3 Aplicação parcial de funções

4. 8.8.4 MEMOIZAÇÃO

eu. 8.9 Resumo

10. Classes

um. 9.1 classes e protótipos

b. 9.2 Classes e Construtores

eu. 9.2.1 Construtores, identidade de classe e instância

ii. 9.2.2 A propriedade do construtor

c. 9.3 Classes com a palavra -chave da classe

eu. 9.3.1 Métodos estáticos

ii. 9.3.2 Getters, Setters e outros formulários de método

iii. 9.3.3 Campos públicos, privados e estáticos

4. 9.3.4 Exemplo: uma classe de números complexos

d. 9.4 Adicionando métodos às classes existentes

e. 9.5 Subclasses

eu. 9.5.1 subclasses e protótipos

ii. 9.5.2 subclasses com extensões e super

iii. 9.5.3 Delegação em vez de herança

4. 9.5.4 Hierarquias de classes e classes abstratas

f. 9.6 Resumo

11. Módulos

um. 10.1 Módulos com Classes, Objetos e Closures

eu. 10.1.1 Automatizando Modularidade Baseada em Closure

b. 10.2 Módulos no Nô

eu. 10.2.1 Exportações de nós

ii. 10.2.2 Importações de nós

iii. 10.2.3 Módulos estilos nô na Web

c. 10.3 Módulos no ES6

eu. 10.3.1 Exportações ES6

ii. 10.3.2 Importações ES6

iii. 10.3.3 Importações e Exportações com Renomeação

4. 10.3.4 Reexportações

v. 10.3.5 Módulos JavaScript na Web

vi. 10.3.6 Importações Dinâmicas com import()

ao vivo 10.3.7 importar.meta.url

d. 10.4 Resumo

12. A Biblioteca Padrão JavaScript

um. 11.1 conjuntos e mapas

eu. 11.1.1 A classe definida

ii. 11.1.2 A classe do mapa

iii. 11.1.3 Frawmap e fraco

b. 11.2 Matrizes digitadas e dados binários

eu. 11.2.1 Tipos de matriz digitados

ii. 11.2.2 Criando matrizes digitadas

iii. 11.2.3 Usando matrizes digitadas

4. 11.2.4 Métodos e propriedades de matriz digitada

v. 11.2.5 DataView e Endianness

c. 11.3 Combinação de padrões com expressões regulares

eu. 11.3.1 Definindo expressões regulares

ii. 11.3.2 Métodos de string para correspondência de padrões

iii. 11.3.3 A classe Regexp

d. 11.4 datas e horários

eu. 11.4.1 Timestamps

ii. 11.4.2 Data aritmética

iii. 11.4.3 Strings de data de formatação e análise

e. 11.5 Error Classes

f. 11.6 Serialização e análise de JSON

eu. 11.6.1 Personalizações JSON

g. 11.7 A API de Internacionalização

eu. 11.7.1 Formatando Números

ii. 11.7.2 Formatando Datas e Horas

iii. 11.7.3 Comparando Strings

h. 11.8 A API do console

eu. 11.8.1 Saída formatada com console

eu. 11.9 APIs de URL

eu. 11.9.1 Funções de URL herdadas

j. 11.10 Temporizadores

k. 11.11 Resumo

13. Iteradores e Geradores

um. 12.1 Como funcionam os iteradores

b. 12.2 Implementando Objetos Iteráveis

eu. 12.2.1 “Fechando” um Iterador: O Método
de Retorno

c. 12.3 Geradores

eu. 12.3.1 Exemplos de Geradores

ii. 12.3.2 rendimento* e geradores recursivos

d. 12.4 Recursos Avançados do Gerador

eu. 12.4.1 O valor de retorno de uma função
geradora

ii. 12.4.2 O valor de uma expressão de rendimento

iii. 12.4.3 Os métodos de retorno () e arremesso ()
de um gerador

4. 12.4.4 Uma nota final sobre geradores

e. 12.5 Resumo

14. JavaScript assíncrono

um. 13.1 Programação assíncrona com retornos de chamada

eu. 13.1.1 Timers

ii. 13.1.2 Eventos

iii. 13.1.3 Eventos de rede

4. 13.1.4 retornos de chamada e eventos no nó

b. 13.2 promessas

eu. 13.2.1 Usando promessas

ii. 13.2.2 Promessas de encadeamento

iii. 13.2.3 Resolvendo promessas

4. 13.2.4 Mais sobre promessas e erros

v. 13.2.5 promessas em paralelo

vi. 13.2.6 Fazendo promessas

vii. 13.2.7 Promessas em sequência

c. 13.3 assíncrono e aguardar

eu. 13.3.1 Aguardar expressões

ii. 13.3.2 Funções assíncronas

iii. 13.3.3 Aguardando várias promessas

4. 13.3.4 Detalhes da implementação

d. 13.4 Iteração Assíncrona

eu. 13.4.1 O loop for/await

ii. 13.4.2 Iteradores Assíncronos

iii. 13.4.3 Geradores Assíncronos

4. 13.4.4 Implementando Iteradores Assíncronos

e. 13.5 Resumo

15. Metaprogramação

a. 14.1 Atributos de Propriedade

b. 14.2 Extensibilidade de Objeto

c. 14.3 O Atributo protótipo

d. 14.4 Símbolos Bem Conhecidos

eu. 14.4.1 Símbolo.iterator e Símbolo.asyncIterator

ii. 14.4.2 Símbolo.getInstance

iii. 14.4.3 Símbolo.toStringTag

4. 14.4.4 Símbolo.espécie

v. 14.4.5 Símbolo.isConcatSpreadable

vi. 14.4.6 Símbolos de correspondência de padrões

ao vivo 14.4.7 Símbolo para Primitivo

viii. 14.4.8 Símbolos não esponjáveis

e. 14.5 Tags de modelo

f. 14.6 A API Reflect

g. 14.7 Objetos de proxy

eu. 14.7.1 Invariantes de procuração

h. 14.8 Resumo

16. JavaScript em navegadores da web

um. 15.1 básicos de programação da web

eu. 15.1.1 JavaScript em HTML<script> Tags

ii. 15.1.2 O modelo de objeto de documento

iii. 15.1.3 O objeto global nos
navegadores da Web

4. 15.1.4 Os scripts compartilham um espaço para nome

v. 15.1.5 Execução de programas JavaScript

vi. 15.1.6 Entrada e saída do programa

vii. 15.1.7 Erros do programa

viii. 15.1.8 O modelo de segurança da web

b. 15.2 Eventos

eu. 15.2.1 Categorias de eventos

ii. 15.2.2 Manipuladores de eventos de registro

iii. 15.2.3 Invocação do manipulador de eventos

4. 15.2.4 Propagação de eventos

v. 15.2.5 Cancelamento de eventos

vi. 15.2.6 Despacha eventos personalizados

c. 15.3 Documentos de script

eu. 15.3.1 Selecionando Elementos do Documento

ii. 15.3.2 Estrutura e Travessia do Documento

iii. 15.3.3 Atributos

4. 15.3.4 Conteúdo do Elemento

v. 15.3.5 Criando, Inserindo e Excluindo Nós

vi. 15.3.6 Exemplo: Gerando um Índice

d. 15.4 Scripts CSS

eu. 15.4.1 Classes CSS

ii. 15.4.2 Estilos embutidos

iii. 15.4.3 Estilos computados

4. 15.4.4 Folhas de estilo de script

v. 15.4.5 Animações e eventos CSS

e. 15.5 Geometria e rolagem do documento

eu. 15.5.1 Coordenadas do documento e
coordenadas da janela de visualização

ii. 15.5.2 Consultando a Geometria de um
Elemento

iii. 15.5.3 Determinando o Elemento em um
Ponto

4. 15.5.4 Rolagem

v. 15.5.5 Tamanho da janela de visualização,
tamanho do conteúdo e posição de rolagem

f. 15.6 Componentes da Web

eu. 15.6.1 Usando componentes da Web

ii. 15.6.2 Modelos HTML

iii. 15.6.3 Elementos Personalizados

4. 15.6.4 Sombra DOM

v. 15.6.5 Exemplo: a<search-box>
Componente Web

g. 15.7 SVG: gráficos vetoriais escaláveis

eu. 15.7.1 SVG em HTML

ii. 15.7.2 Script SVG

iii. 15.7.3 Criando imagens SVG com
JavaScript

h. 15.8 Gráficos em um<canvas>

eu. 15.8.1 Caminhos e Polígonos

ii. 15.8.2 Dimensões e coordenadas da
tela

iii. 15.8.3 Atributos Gráficos

4. 15.8.4 Operações de desenho em tela

v. 15.8.5 Transformações do Sistema de Coordenadas

nós. 15.8.6 Recorte

vii. 15.8.7 Manipulação de pixels

eu. 15.9 APIs de áudio

eu. 15.9.1 O Construtor Audio()

ii. 15.9.2 A API WebAudio

j. 15.10 Localização, navegação e histórico

eu. 15.10.1 Carregando Novos Documentos

ii. 15.10.2 Histórico de navegação

iii. 15.10.3 Gerenciamento de histórico
com eventos hashchange

4. 15.10.4 Gerenciamento de histórico com
pushState()

k. 15.11 Rede

eu. 15.11.1 buscar()

ii. 15.11.2 Eventos enviados pelo servidor

iii. 15.11.3 WebSockets

eu. 15.12 Armazenamento

eu. 15.12.1 armazenamento local e armazenamento de sessão

ii. 15.12.2 Cookies

iii. 15.12.3 IndexadoDB

m. 15.13 Threads de trabalho e mensagens

eu. 15.13.1 Objetos Trabalhadores

ii. 15.13.2 O Objeto Global nos Trabalhadores

iii. 15.13.3 Importando Código para um Worker

4. 15.13.4 Modelo de Execução do Trabalhador

v. 15.13.5 postMessage(), MessagePorts e
MessageChannels

vi. 15.13.6 Mensagens de origem cruzada
com postMessage()

n. 15.14 Exemplo: O Conjunto Mandelbrot

ó. 15.15 Resumo e sugestões para leitura adicional

eu. 15.15.1 HTML e CSS

ii. 15.15.2 Desempenho

iii. 15.15.3 Segurança

4. 15.15.4 WebAssembly

v. 15.15.5 Mais recursos de documentos e
janelas

vi. 15.15.6 Eventos

vii. 15.15.7 Aplicativos Web Progressivos e
Service Workers

viii. 15.15.8 APIs de dispositivos móveis

Ix. 15.15.9 APIs binárias

x. 15.15.10 APIs de mídia

xii. 15.15.11 Criptografia e APIs relacionadas

17. JavaScript do lado do servidor com Node

um. 16.1 Noções básicas de programação de nós

eu. 16.1.1 Saída do console

ii. 16.1.2 Argumentos de linha de comando e
variáveis de ambiente

iii. 16.1.3 Ciclo de Vida do Programa

4. 16.1.4 Módulos de Nó

v. 16.1.5 O Gerenciador de Pacotes do Nó

b. 16.2 O nó é assíncrono por padrão

c. 16.3 Buffers

d. 16.4 Eventos e EventEmitter

e. 16.5 Fluxos

eu. 16.5.1 Tubulações

ii. 16.5.2 Iteração Assíncrona

iii. 16.5.3 Gravando em Streams e Manipulando Contrapressão

4. 16.5.4 Lendo Streams com Eventos

f. 16.6 Detalhes do processo, CPU e sistema operacional

g. 16.7 Trabalhando com Arquivos

eu. 16.7.1 Caminhos, Descritores de Arquivos e FileHandles

ii. 16.7.2 Lendo Arquivos

iii. 16.7.3 Gravando Arquivos

4. 16.7.4 Operações de Arquivo

v. 16.7.5 Metadados de arquivo

vi. 16.7.6 Trabalhando com diretórios

h. 16.8 Clientes e Servidores HTTP

eu. 16.9 Servidores e clientes de rede não HTTP

j. 16.10 Trabalhando com Processos Filhos

eu. 16.10.1 execSync() e execFileSync()

ii. 16.10.2 exec() e execFile()

iii. 16.10.3 geração()

4. 16.10.4 garfo()

k. 16.11 Threads de Trabalho

eu. 16.11.1 Criando Workers e Passando Mensagens

ii. 16.11.2 O Ambiente de Execução do Trabalhador

iii. 16.11.3 Canais de comunicação e MessagePorts

4. 16.11.4 Transferindo MessagePorts e Arrays Digitados

v. 16.11.5 Compartilhando arrays digitados entre threads

eu. 16.12 Resumo

18. Ferramentas e extensões JavaScript

um. 17.1 Linting com ESLint

b. 17.2 Formatação JavaScript com mais bonito

c. 17.3 Teste de Unidade com Jest

d. 17.4 Gerenciamento de pacotes com npm

e. 17.5 Pacote de Código

f. 17.6 Transpilação com Babel

g. 17.7 JSX: Expressões de marcação em JavaScript

h. 17.8 Verificação de tipo com fluxo

eu. 17.8.1 Instalando e executando o fluxo

ii. 17.8.2 Usando anotações de tipo

iii. 17.8.3 Tipos de Classe

4. 17.8.4 Tipos de Objetos

v. 17.8.5 Aliases de tipo

vi. 17.8.6 Tipos de Matriz

vii. 17.8.7 Outros tipos parametrizados

viii. 17.8.8 Tipos somente leitura

ix. 17.8.9 Tipos de Funções

x. 17.8.10 Tipos de União

xii. 17.8.11 Tipos Enumerados e
Uniões Discriminadas

eu. 17.9 Resumo

19. Índice

Louvor ao JavaScript: The Definitive Guide, Sétima Edição

“Este livro é tudo o que você nunca soube que queria saber sobre JavaScript. Leve a qualidade do código JavaScript e a produtividade para o próximo nível. O conhecimento de David sobre a linguagem, seus meandros e betchas, é surpreendente, e brilha neste guia verdadeiramente definitivo para a linguagem JavaScript.”

- Schalk Neethling, engenheiro de frente sênior da MDN Web Docs

“David Flanagan leva os leitores a uma visita guiada a JavaScript que fornecerá a eles uma imagem completa do idioma e seu ecossistema”.

- Sarah Wachs, desenvolvedor de front -end e mulheres que codificam o líder de Berlim

“Qualquer desenvolvedor interessado em ser produtivo em bases de código desenvolvido durante toda a vida de JavaScript (incluindo os recursos mais recentes e emergentes) será bem servido por uma jornada profunda e reflexiva por este livro abrangente e definitivo.”

—Brian Sletten, presidente da Bosatsu Consulting

JavaScript: o definitivo

Guia

SÉTIMA EDIÇÃO

Domine a programação mais usada do mundo

Linguagem

David Flanagan



JavaScript: o guia definitivo, sétima edição

por David Flanagan

Direitos autorais © 2020 David Flanagan. Todos os direitos reservados.

Impresso nos Estados Unidos da América.

Publicado por O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Os livros da O'Reilly podem ser adquiridos para uso educacional, comercial ou promocional de vendas. Edições online também estão disponíveis para a maioria dos títulos (<http://oreilly.com>). Para mais informações, entre em contato com nosso

departamento de vendas corporativas/institucionais:
800-998-9938 ou corporate@oreilly.com.

Editora de Aquisições: Jennifer Pollock

Development Editor: Angela Rufino

Editora de produção: Deborah Baker

Editora: Holly Bauer Forsyth

Revisor: Piper Editorial, LLC

Indexadora: Judith McConville

Designer de Interiores: David Futato

Designer da capa: Karen Montgomery

Ilustrador: Rebecca DeMarest

Junho de 1998: Terceira edição

Novembro de 2001: Quarta edição

Agosto de 2006: Quinta Edição

Maio de 2011: Sexta edição

Maio de 2020: sétima edição

Histórico de revisão para a sétima edição

- 2020-05-13: Primeira versão

Consulte <http://oreilly.com/catalog/errata.csp?isbn=9781491952023> para obter detalhes de lançamento.

O logotipo O'Reilly é uma marca registrada da O'Reilly Media, Inc. JavaScript: The Definitive Guide, Sétima Edição, a imagem da capa e o vestido comercial relacionado são marcas comerciais da O'Reilly Media, Inc.

Embora o editor e os autores tenham usado esforços de boa fé para garantir que as informações e instruções contidas neste trabalho sejam precisas, o editor e os autores renunciam a toda a responsabilidade por erros ou omissões, incluindo a responsabilidade sem limitação por danos resultantes do uso de ou confiança nesse trabalho. O uso das informações e instruções contidas neste trabalho é por sua conta e risco. Se qualquer amostras de código ou outra tecnologia que este trabalho contenha ou descreva estiver sujeito a licenças de código aberto ou aos direitos de propriedade intelectual de outras pessoas, é sua responsabilidade garantir que seu uso esteja em conformidade com essas licenças e/ou direitos.

978-1-491-95202-3

[Lsi]

Dedicação

Aos meus pais, Donna e Matt, com amor e gratidão.

Prefácio

Este livro aborda a linguagem JavaScript e as APIs JavaScript implementadas por navegadores web e por Node. Eu o escrevi para leitores com alguma experiência anterior em programação que desejam aprender JavaScript e também para programadores que já usam JavaScript, mas desejam levar seu entendimento a um novo nível e realmente dominar a linguagem. Meu objetivo com este livro é documentar a linguagem JavaScript de forma abrangente e definitiva e fornecer uma introdução aprofundada às APIs mais importantes do lado do cliente e do lado do servidor disponíveis para programas JavaScript. Como resultado, este é um livro longo e detalhado. Minha esperança, entretanto, é que ele recompense o estudo cuidadoso e que o tempo que você gasta lendo seja facilmente recuperado na forma de maior conhecimento.

produtividade da programação.

As edições anteriores deste livro incluíam uma seção de referência abrangente. Não sinto mais que faça sentido incluir esse material em formato impresso quando é tão rápido e fácil encontrar online material de referência atualizado. Se você precisar procurar algo relacionado ao JavaScript principal ou do lado do cliente, recomendo que você visite o site do MDN. E para APIs Node do lado do servidor, recomendo que você vá diretamente à fonte e consulte a documentação de referência do Node.js.

Convenções usadas neste livro

Eu uso as seguintes convenções tipográficas neste livro:

ítálico

É usado para ênfase e para indicar o primeiro uso de um termo. O itálico também é usado para endereços de email, URLs e nomes de arquivos.

Largura constante

É usado em todo o código JavaScript e listagens CSS e HTML e, geralmente, para qualquer coisa que você digite literalmente ao programar.

Largura constante em itálico

É usado ocasionalmente ao explicar a sintaxe do JavaScript.

Largura constante em negrito

Mostra comandos ou outro texto que deve ser digitado literalmente pelo usuário

OBSERVAÇÃO

Esse elemento significa uma nota geral.

IMPORTANTE

Este elemento indica um aviso ou cautela.

Código de exemplo

Material suplementar (exemplos de código, exercícios, etc.) para este livro está disponível para download em:

https://oreil.ly/javascript_defgd7

Este livro está aqui para ajudá-lo a fazer seu trabalho. Em geral, se o código de exemplo for oferecido com este livro, você poderá usá-lo em seus programas e documentação. Você não precisa entrar em contato conosco para obter permissão, a menos que esteja reproduzindo uma parte significativa do código. Por exemplo, escrever um programa que usa vários pedaços de código deste livro não requer permissão. Vender ou distribuir exemplos de livros de O'Reilly requer permissão. Responder a uma pergunta citando este livro e citando o código de exemplo não requer permissão.

A incorporação de uma quantidade significativa de código de exemplo deste livro na documentação do seu produto requer permissão.

Agradecemos, mas geralmente não exigem, atribuição. Uma atribuição geralmente inclui o título, autor, editor e ISBN. Por exemplo: "JavaScript: The Definitive Guide, Sétima Edição, de David Flanagan (O'Reilly). Copyright 2020 David Flanagan, 978-1-491- 95202-3. "

Se você sentir que o uso de exemplos de código cai fora do uso justo ou da permissão dada acima, sinta-se à vontade para entrar em contato conosco em

permissions@oreilly.com.

O'Reilly Online Learning

OBSERVAÇÃO

Por mais de 40 anos, a O'Reilly Media forneceu treinamento em tecnologia e negócios, conhecimento e insight para ajudar as empresas a ter sucesso.

Nossa rede exclusiva de especialistas e inovadores compartilham seus conhecimentos e conhecimentos por meio de livros, artigos e nossa plataforma de aprendizado on-line. A plataforma de aprendizado on-line de O'Reilly oferece acesso sob demanda a cursos de treinamento ao vivo, caminhos de aprendizado detalhados, codificação interativa

Ambientes e uma vasta coleção de texto e vídeo de O'Reilly e mais de 200 outros editores. Para mais informações, visite

<http://oreilly.com>.

Como nos contatar

Aborde os comentários e perguntas sobre este livro ao editor:

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (nos Estados Unidos ou no Canadá)

707-829-0515 (internacional ou local)

707-829-0104 (fax)

Temos uma página da Web para este livro, onde listamos erratas, exemplos e qualquer informação adicional. Você pode acessar esta página em

https://oreil.ly/javascript_defgd7.

Envie um email para bookquestions@oreilly.com para comentar ou perguntar técnico

perguntas sobre este livro.

Para novidades e mais informações sobre nossos livros e cursos, consulte nosso site em <http://www.oreilly.com>.

Encontre-nos no Facebook: <http://facebook.com/oreilly>

Siga-nos no Twitter: <http://twitter.com/oreillymedia>

Assista-nos no YouTube: <http://www.youtube.com/oreillymedia>

Agradecimentos

Muitas pessoas ajudaram na criação deste livro. Gostaria de agradecer à minha editora, Angela Rufino, por me manter no caminho certo e pela paciência com meus prazos perdidos. Obrigado também aos meus revisores técnicos: Brian Sletten, Elisabeth Robson, Ethan Flanagan, Maximiliano Firtman, Sarah Wachs e Schalk Neethling. Seus comentários e sugestões tornaram este livro melhor.

A equipe de produção da O'Reilly fez seu excelente trabalho de sempre: Kristen Brown gerenciou o processo de produção, Deborah Baker foi a editora de produção, Rebecca Demarest desenhou os números e Judy McConville criou o índice.

Editores, revisores e colaboradores de edições anteriores deste livro incluíram: Andrew Schulman, Angelo Sirigos, Aristóteles Pagaltzis, Brendan Eich, Christian Heilmann, Dan Shafer, Dave C. Mitchell, Deb Cameron, Douglas Crockford, Dr.

Schiemann, Frank Willison, Geoff Stearns, Herman Venter, Jay Hodges, Jeff Yates, Joseph Kesselman, Ken Cooper, Larry Sullivan, Lynn Rollins, Neil Berkman, Mike Loukides, Nick Thompson, Norris Boyd, Paula Ferguson, Peter-Paul Koch, Philippe Le Hegaret, Raffaele Cecco, Richard Yaker, Sanders Kleinfeld, Scott Furman, Scott Isaacs, Shon Katzenberger, Terry Allen, Todd Ditchendorf, Vidur Aparrao, Waldemar Horwat e Zachary Kessin.

Escrever esta sétima edição me manteve longe da minha família por muitas noites. Meu amor a eles e aos meus agradecimentos por aguentar minhas ausências.

David Flanagan, março de 2020

Capítulo 1. Introdução ao JavaScript

JavaScript é a linguagem de programação da web. A esmagadora maioria dos sites usa JavaScript, e todos os navegadores modernos — em desktops, tablets e telefones — incluem interpretadores de JavaScript, tornando o JavaScript a linguagem de programação mais implantada da história. Na última década, o Node.js permitiu a programação JavaScript fora dos navegadores da web, e o sucesso dramático do Node significa que o JavaScript é agora também a linguagem de programação mais usada entre os desenvolvedores de software. Esteja você começando do zero ou já usando JavaScript profissionalmente, este livro o ajudará a dominar a linguagem.

Se você já está familiarizado com outras linguagens de programação, pode ser útil saber que JavaScript é uma linguagem de programação interpretada, dinâmica e de alto nível, adequada para estilos de programação funcionais e orientados a objetos. As variáveis do JavaScript não são digitadas. Sua sintaxe é vagamente baseada em Java, mas as linguagens não estão relacionadas. JavaScript deriva suas funções de primeira classe de Scheme e sua herança baseada em protótipo da linguagem pouco conhecida Self. Mas você não precisa conhecer nenhuma dessas linguagens, nem estar familiarizado com esses termos, para usar este livro e aprender JavaScript.

O nome “JavaScript” é bastante enganador. Exceto por uma semelhança sintática superficial, JavaScript é completamente diferente do Java

linguagem de programação. Javascript já superou suas raízes em linguagem de script para se tornar uma linguagem geral robusta e eficiente, adequada para engenharia e projetos sérios de software com enormes bases de código.

JavaScript: nomes, versões e modos

O JavaScript foi criado no Netscape nos primeiros dias da Web e, tecnicamente, "JavaScript" é uma marca registrada licenciada da Sun Microsystems (agora Oracle) usada para descrever a implementação do idioma do idioma do Netscape (agora Mozilla). O Netscape enviou o idioma para padronização à ECMA - a Associação Europeia de Fabricante de Computadores - e, devido a questões de marca registrada, a versão padronizada do idioma ficou presa com o nome estranho "EcmaScript". Na prática, todos chamam o idioma JavaScript. Este livro usa o nome "ECMAScript" e a abreviação "es" para se referir ao padrão do idioma e às versões desse padrão.

Na maior parte dos 2010, a versão 5 do padrão ECMAScript foi suportada por todos os navegadores da Web. Este livro trata o ES5 como a linha de base da compatibilidade e não discute mais versões anteriores do idioma. O ES6 foi lançado em 2015 e adicionou novos recursos - incluindo a sintaxe da classe e do módulo - que alterou o JavaScript de um idioma de script em um idioma sério e de uso geral adequado para engenharia de software em larga escala. Desde o ES6, a especificação do ECMAScript mudou para uma cadência de liberação anual e versões do idioma - ES2016, ES2017, ES2018, ES2019 e ES2020 - agora são identificadas no ano de lançamento.

À medida que o JavaScript evoluiu, os designers de idiomas tentaram corrigir falhas nas versões iniciais (pré-ES5). Para manter a compatibilidade com versões anteriores, não é possível remover os recursos herdados, por mais falha. Mas no ES5 e posterior, os programas podem optar pelo modo rigoroso de JavaScript, no qual vários erros de idioma inicial foram corrigidos. O mecanismo de opção é a diretiva "Use Strict", descrita no §5.6.3. Essa seção também resume as diferenças entre JavaScript Legacy e JavaScript rigoroso. No ES6 e posterior, o uso de novos recursos de idioma geralmente invoca implicitamente o modo rigoroso. Por exemplo, se você usar a palavra-chave da classe ES6 ou criar um módulo ES6, todo o código dentro da classe ou módulo será automaticamente rigoroso e os recursos antigos e falhos não estarão disponíveis nesses contextos. Este livro abordará os recursos legados do JavaScript, mas tem o cuidado de salientar que eles não estão disponíveis no modo rigoroso.

Para ser útil, todo idioma deve ter uma plataforma ou biblioteca padrão, para realizar coisas como entrada e saída básicas. A linguagem principal JavaScript define uma API mínima para trabalhar com números, texto, matrizes, conjuntos, mapas e assim por diante, mas não inclui nenhuma funcionalidade de entrada ou saída. Entrada e saída (bem como recursos mais sofisticados, como redes, armazenamento e gráficos) são de responsabilidade do

“ambiente host” no qual o JavaScript está incorporado.

O ambiente host original para JavaScript era um navegador da web e ainda é o ambiente de execução mais comum para código JavaScript. O ambiente do navegador da web permite que o código JavaScript obtenha informações do mouse e do teclado do usuário e faça solicitações HTTP. E permite que o código JavaScript exiba a saída ao usuário com HTML e CSS.

Desde 2010, outro ambiente host está disponível para código JavaScript. Em vez de restringir o JavaScript para funcionar com as APIs fornecidas por um navegador da web, o Node dá acesso ao JavaScript a todo o sistema operacional, permitindo que programas JavaScript leiam e gravem arquivos, enviem e recebam dados pela rede e façam e sirvam solicitações HTTP. Node é uma escolha popular para implementação de servidores web e também uma ferramenta conveniente para escrever scripts utilitários simples como uma alternativa aos scripts shell.

A maior parte deste livro se concentra na própria linguagem JavaScript. O Capítulo 11 documenta a biblioteca padrão JavaScript, o Capítulo 15 apresenta o ambiente host do navegador da web e o Capítulo 16 apresenta o ambiente host Node.

Este livro cobre primeiro os fundamentos de baixo nível e, em seguida, desenvolve abstrações mais avançadas e de nível superior. Os capítulos devem ser lidos mais ou menos em ordem. Mas aprender um novo

Uma linguagem de programação nunca é um processo linear, e descrever uma linguagem também não é linear: cada recurso da linguagem está relacionado a outros recursos, e este livro está cheio de referências cruzadas – às vezes

para trás e às vezes para frente - para material relacionado. Este capítulo introdutório faz uma rápida primeira passagem pela linguagem, introduzindo recursos-chave que facilitarão a compreensão do tratamento aprofundado nos capítulos que se seguem. Se você já é um programador JavaScript praticante, provavelmente poderá pular este capítulo. (Embora você possa gostar de ler o Exemplo 1-1 no final do capítulo antes de prosseguir.)

1.1 Explorando JavaScript

Ao aprender uma nova linguagem de programação, é importante experimentar os exemplos do livro, depois modificá-los e tentar novamente para testar sua compreensão da linguagem. Para fazer isso, você precisa de um interpretador JavaScript.

A maneira mais fácil de experimentar algumas linhas de JavaScript é abrir as ferramentas de desenvolvedor web em seu navegador (com F12, Ctrl-Shift-I ou Command-Option-I) e selecionar a guia Console. Você pode então digitar o código no prompt e ver os resultados enquanto digita. As ferramentas de desenvolvimento do navegador geralmente aparecem como painéis na parte inferior ou direita da janela do navegador, mas geralmente você pode separá-las como janelas separadas (conforme ilustrado na Figura 1-1), o que geralmente é bastante conveniente.

The screenshot shows the Firefox Developer Tools interface with the 'Console' tab selected. The title bar reads 'Developer Tools - Node.js - https://nodejs.org/en/'. Below the tabs, there's a toolbar with icons for Inspector, Console, Debugger, Style Editor, Performance, Memory, Network, and more. The main area is a text-based console window. It starts with a command to create an array of primes, followed by several arithmetic operations and function definitions, including a factorial function named 'fact'. The console outputs the results of each expression or function call.

```
» let primes = [2, 3, 5, 7];
← undefined
» primes[primes.length-1]
← 7
» primes[0] + primes[1]
← 5
» function fact(x) {
  if (x > 1) return x * fact(x-1);
  else return 1;
}
← undefined
» fact(4)
← 24
» fact(5)
← 120
» fact(6)
← 720
»
```

Figura 1-1. O console JavaScript nas ferramentas para desenvolvedores do Firefox

Outra maneira de testar o código JavaScript é baixar e instalar o Node em <https://nodejs.org>. Depois que o Node estiver instalado em seu sistema, você pode simplesmente abrir uma janela do Terminal e digitar node para iniciar um

sessão JavaScript interativa como esta:

```
$ node Bem-vindo ao Node.js v12.13.0. Digite
"help" para obter mais informações.
> .help .break Às vezes você fica preso,
isso te tira de lá .clear Alias para .break

.editor      Entre no modo editor
.saída       Saia da resposta
.ajuda        Imprima esta mensagem de ajuda
.carregar    Carregar JS de um arquivo na sessão REPL
.salvar       Salve todos os comandos avaliados nesta sessão REPL em
um arquivo

Pressione ^C para abortar a expressão atual, ^D para sair da
reposição >; deixe x = 2, y = 3;
indefinido
> x + y 5

> (x === 2) &&
(y === 3) verdadeiro
> (x > 3) || (y
< 3) falso
```

1.2 Olá mundo

Quando você estiver pronto para começar a experimentar trechos de código mais longos, esses ambientes interativos linha por linha poderão não ser mais adequados e você provavelmente preferirá escrever seu código em um editor de texto. A partir daí, você pode copiar e colar no console JavaScript ou em uma sessão do Node. Ou você pode salvar seu código em um arquivo (a extensão de nome de arquivo tradicional para código JavaScript é `.js`) e então executar esse arquivo de código JavaScript com o Node:

```
$ trecho de nó.js
```

Se você usar o Node de maneira não interativa como esta, ele não

imprima automaticamente o valor de todo o código executado, então você terá que fazer isso sozinho. Você pode usar a função `console.log()` para exibir texto e outros valores JavaScript na janela do terminal ou no console de ferramentas do desenvolvedor do navegador. Então, por exemplo, se você criar um arquivo `hello.js` contendo esta linha de código:

```
console.log("Olá mundo!");
```

e execute o arquivo com o nó `hello.js`, você verá a mensagem “Hello World!” impresso.

Se você quiser ver a mesma mensagem impressa no console JavaScript de um navegador da web, crie um novo arquivo chamado `hello.html` e coloque este texto nele:

```
<script src="hello.js"></script>
```

Em seguida, carregue `hello.html` em seu navegador usando um URL file:// como este:

```
arquivo:///Usuários/nomedesusuário/javascript/hello.html
```

Abra a janela de ferramentas do desenvolvedor para ver a saudação no console.

1.3 Um tour pelo JavaScript

Esta seção apresenta uma introdução rápida, por meio de exemplos de código, à linguagem JavaScript. Após este capítulo introdutório, mergulhamos no JavaScript no nível mais baixo: o Capítulo 2 explica coisas como comentários JavaScript, ponto-e-vírgula e o conjunto de caracteres Unicode. O Capítulo 3 começa a ficar mais interessante: ele explica as variáveis JavaScript e os valores

você pode atribuir a essas variáveis.

Aqui está um exemplo de código para ilustrar os destaques desses dois capítulos:

```
// Qualquer coisa após barras duplas é um comentário em
// inglês.
// Leia os comentários com atenção: eles explicam o código
// JavaScript.

// Uma variável é um nome simbólico para um valor. // Variáveis são
declaradas com a palavra-chave let: let x; // Declara uma variável
chamada x.

// Valores podem ser atribuídos a variáveis com
sinal = x = 0; // Agora a variável x tem o valor 0

x                         // => 0: Uma variável é avaliada como
seu valor.

// JavaScript suporta vários tipos de valores x = 1; // Números.

x = 0,01;                  // Os números podem ser inteiros ou
reais. x = "olá
mundo";                   // Sequências de texto entre aspas
marcas. x =
'JavaScript';            // Aspas simples também delimitam
cordas. x =
verdadeiro;                // Um valor booleano.
x = false;                 // O outro valor booleano.
x = nulo;                  // Nulo é um valor especial que
significa "sem
valor". x = indefinido;    // Indefinido é outro especial
valor como nulo.
```

Dois outros tipos muito importantes que os programas JavaScript podem manipular são objetos e arrays. Esses são os assuntos dos Capítulos 6 e 7, mas são tão importantes que você os verá muitas vezes antes de chegar a esses capítulos:

```
// O tipo de dados mais importante do JavaScript é
o objeto. // Um objeto é uma coleção de pares
nome/valor ou uma string para mapa de valor. let
book = { // Os objetos são colocados entre chaves.
    tópico: "JavaScript", // A propriedade "tópico" tem valor
    "JavaScript": "JavaScript",
    edição: 7 // A propriedade "edição" possui
    valor 7 }; // A chave marca o final do objeto.

//Acesse as propriedades de um objeto com . ou []: book.topic
// => "JavaScript"; book["edition"] // => 7: outra forma de acessar valores de propriedades.

livro.autor = "Flanagan"; // Cria novas propriedades por
atribuição.
livro.contents = {} // {} é um objeto vazio sem
propriedades.

// Acessar condicionalmente as propriedades com ?. (ES2020):
book.contents?.ch01?.sect1 // => undefined: book.contents não
possui propriedade ch01.

// JavaScript também suporta arrays (listas indexadas
numericamente) de valores:
deixe primos = [2, 3, 5, 7]; // Um array de 4 valores, delimitado por [
e ].
primos[0] // => 2: o primeiro elemento (índice
0) da matriz.
primos.comprimento // => 4: quantos elementos no
variedade.
primos[primos.comprimento-1] // => 7: o último elemento do
variedade. primos[4] = 9; // Adiciona um novo elemento por
atribuição. primos[4] = 11; // Ou altere um elemento existente por
atribuição. deixe
vazio = []; // [] é um array vazio sem
elementos.
vazio.comprimento // => 0

// Arrays e objetos podem conter outros arrays e objetos:
```

```
deixe pontos = [ // Um array com 2 elementos.  
    {x: 0, y: 0}, // Cada elemento é um objeto.  
    {x: 1, y: 1}];  
  
deixe dados = { //Um objeto com 2 propriedades  
    ensai01: [[1,2], [3,4]], // O valor de cada propriedade é  
    uma matriz.  
    ensai02: [[2,3], [4,5]] //os elementos dos arrays  
    são matrizes. };
```

SINTAXE DE COMENTÁRIO EM EXEMPLOS DE CÓDIGO

Você deve ter notado no código anterior que alguns dos comentários começam com uma seta (=>). Eles mostram o valor produzido pelo código antes do comentário e são minha tentativa de emular um ambiente JavaScript interativo como um console de navegador da web em um livro impresso.

Esses // => comentários também servem como uma afirmação, e eu escrevi uma ferramenta que testa o código e verifica se ele produz o valor especificado no comentário. Espero que isso ajude a reduzir erros no livro.

Existem dois estilos relacionados de comentário/afirmação. Se você vir um comentário no formato // a == 42, significa que após o código antes da execução do comentário, a variável a terá o valor 42. Se você vir um comentário no formato // !, significa que o código na linha antes do comentário lança uma exceção (e o restante do comentário após o ponto de exclamação geralmente explica que tipo de exceção é lançada).

Você verá esses comentários usados ao longo do livro.

A sintaxe ilustrada aqui para listar elementos de matriz entre colchetes ou mapear nomes de propriedades de objetos para valores de propriedades entre chaves é conhecida como expressão inicializadora e é apenas um dos tópicos do Capítulo 4. Uma expressão é uma frase de JavaScript que pode ser avaliado para produzir um valor. Por exemplo, o uso de . e [] para se referir ao valor de uma propriedade de objeto ou elemento de matriz é uma expressão.

Uma das maneiras mais comuns de formar expressões em JavaScript é

usar operadores:

```
// Os operadores agem sobre os valores (os operandos) para produzir um novo valor.  
// Os operadores aritméticos são alguns dos mais simples:  
3 + 2 // => 5: adição 3 - 2 // => 1: subtração 3 * 2 // => 6: multiplicação 3/2 // => 1,5: divisão  
points[1].x - points[0].x // => 1: operandos mais complicados também funcionam "+" + "2" // => "32":: + soma números,  
concatena strings  
  
// JavaScript define alguns operadores aritméticos abreviados let count = 0; //Definir uma variável  
contar++; //Incrementa a variável  
contar--; //Decrementa a variável  
contar += 2; // Adiciona 2: igual a count = count + 2; contar *= 3; // Multiplique por 3: igual a count = conte * 3; count // => 6: nomes de variáveis também são expressões.  
  
// Operadores relacionais e de igualdade testam se dois valores são iguais,  
// desigual, menor que, maior que e assim por diante. Eles avaliam como verdadeiro ou falso. seja x = 2, y = 3; // Estes sinais = são testes de atribuição, não de igualdade  
  
x === y // => falso: igualdade  
== y // => verdadeiro: desigualdade  
x < y // => verdadeiro: menor que  
x <= y // => verdadeiro: menor ou igual  
x > y // => falso: maior que  
x >= y // => falso: maior que ou igual  
&quot;dois" === &quot;três" // => false: as duas strings são diferentes &quot;dois";  
&gt; &quot;três" // => true: &quot;tw&quot; está em ordem alfabética  
maior que &quot;th&quot;  
falso === (x &gt; y) // => verdadeiro: falso é igual a falso  
  
// Operadores lógicos combinam ou invertem valores booleanos
```

```
(x === 2) && (y === 3) // => true: ambas as comparações são verdadeiro. && é e  
(x> 3) || (y <3) // => false: nenhuma comparação é verdadeiro. ||  
é ou! (x === y) // => true :! inverte um booleano  
valor
```

Se as expressões JavaScript são como frases, as instruções JavaScript são como frases completas. As declarações são o tópico do capítulo 5. Aproximadamente, uma expressão é algo que calcula um valor, mas não faz nada: não altera o estado do programa de forma alguma. As declarações, por outro lado, não têm um valor, mas alteram o estado. Você viu declarações variáveis e declarações de atribuição acima. A outra ampla categoria de declaração são as estruturas de controle, como condicionais e loops. Você verá exemplos abaixo, depois de cobrirmos as funções.

Uma função é um bloco nomeado e parametrizado do código JavaScript que você define uma vez e pode invocar repetidamente. As funções não são cobertas formalmente até o capítulo 8, mas, como objetos e matrizes, você os verá muitas vezes antes de chegar a esse capítulo. Aqui estão alguns exemplos simples:

```
// As funções são blocos parametrizados do código JavaScript que podemos invocar.  
função plus1 (x) { // Defina uma função chamada "Plus1"  
com o parâmetro "x";  
    retornar x + 1; // retorna um valor maior que  
O valor aprovado em} // funções são fechadas em  
aparelhos encaracolados  
  
plus1 (y) // => 4: y é 3, então este  
Retornos de invocação 3+1  
  
Let Square = function (x) { // As funções são valores e podem ser  
atribuídas a VARs
```

```

    retornar x * x;           // Calcule o valor da função
};                         // semicolon marca o fim do
atribuição.

quadrado (mais1 (y))      // => 16: Invoque duas funções em
uma expressão

```

No ES6 e mais tarde, há uma sintaxe abreviada para definir funções. Essa sintaxe concisa usa `=>` para separar a lista de argumentos do corpo da função; portanto, as funções definidas dessa maneira são conhecidas como funções de seta. As funções de seta são mais usadas quando você deseja passar uma função sem nome como argumento para outra função. O código anterior se parece com isso quando reescrito para usar funções de seta:

```

const Plus1 = x => x + 1;    // A entrada X mapeia para a saída
x + 1 const square = x
=> x * x;                   // A entrada X mapeia para a saída
x * x
plus1 (y)                   // => 4: A invocação da função é
o mesmo quadrado (mais1 (y)) // => 16

```

Quando usamos funções com objetos, obtemos métodos:

```

// Quando as funções são atribuídas às propriedades de um
objeto, chamamos
// eles "métodos". Todos os objetos JavaScript (incluindo matrizes)
tem métodos: deixe a = [];
                // Crie uma matriz vazia
a.push (1,2,3);          // o método push () adiciona elementos
a uma matriz
A. reverse ();           // Outro método: reverter o
ordem dos elementos

// Também podemos definir nossos próprios métodos. A palavra
-chave "isto" refere -se ao objeto
// no qual o método é definido: neste caso, a matriz de pontos
anteriores.
pontos.dist = function () { // define um método para calcular

```

```

distância entre pontos
    Seja p1 = this [0];           // Primeiro elemento da matriz somos
invocou
    Seja P2 = este [1];           // segundo elemento do "isto"
objeto
    Seja a = p2.x-p1.x;          // diferença em coordenadas x
    Seja b = p2.y-p1.y;          // diferença nas coordenadas y
    Retornar Math.sqrt (A*A + // O Teorema Pitagórico
                        b*b); // math.sqrt () calcula o quadrado
raiz };
Points.dist ()
                // => math.sqrt (2): distância
Entre nossos 2 pontos

```

Agora, como prometido, aqui estão algumas funções cujos corpos demonstram declarações comuns da estrutura de controle de javascript:

```

// declarações JavaScript incluem condicionais e loops usando a
sintaxe
// de C, C ++, Java e outros idiomas. função abs (x)
{// uma função para calcular o valor absoluto.

    if (x>= 0) {                  // a declaração if ...
        retornar x;               // executa este código se o
A comparação é verdadeira.
    }                                // Este é o fim do IF
cláusula.
    outro {                         // a cláusula opcional mais
executa seu código se
        retornar -x;               // A comparação é falsa.
    }                                // aparelho encaracolado opcional quando 1
declaração por cláusula. } // Nota As
instruções de retorno aninhadas dentro se/else.

ABS (-10) === Abs (10)           // => true

Função Sum (Array) {             // Calcule a soma dos elementos
de uma matriz
    deixe soma = 0;               // Comece com uma soma inicial de 0.
    para (Let X of Array) {       // loop sobre a matriz, atribuindo cada
elemento para x.

```

```

        soma += x;;           // Adicione o valor do elemento ao
soma.
    }
    soma de retorno;      // Este é o fim do loop.
} soma (primos)          // retorna a soma.

Prima 2+3+5+7+11

função factorial (n) {   // uma função para calcular
fatoriais
    deixe o produto = 1;  // Comece com um produto de 1
    while (n> 1) {        // Repita declarações em {} enquanto
expr in () é verdadeiro
        produto *= n;    // atalho para produto = produto
* n;
        n--;
    }
    produto de retorno; // retorna o produto
} Fatorial (4)          // => 24: 1*4*3*2

função factorial2 (n) {  // Outra versão usando um
loop diferente
    Deixe eu, produto = 1; // Comece com 1
    para (i = 2; i <= n; i++) // incremento automaticamente i de
2 até n
        produto *= i;       // Faça isso cada vez. {} não
necessário para loops de 1 line
    produto de retorno;  // retorna o fatorial
} factorial2 (5)          // => 120: 1*2*3*4*5

```

O JavaScript suporta um estilo de programação orientado a objetos, mas é significativamente diferente das linguagens de programação orientadas a objetos "clássicos". O capítulo 9 abrange a programação orientada a objetos em JavaScript em detalhes, com muitos exemplos. Aqui está um exemplo muito simples que demonstra como definir uma classe JavaScript para representar pontos geométricos 2D. Objetos que são casos desta classe têm um

Método único, denominado distância (), que calcula a distância do ponto da origem:

```
Classe Point {                                // por convenção, os nomes das classes são
    capitalizado.
        Construtor (x, y) {                // função construtora para
            Inicialize novas instâncias.
                this.x = x;                  // essa palavra -chave é o novo objeto
                sendo inicializado. this.y   // armazenar argumentos de função como
                = y;                        // Propriedades do objeto.
                }
            Funções do construtor.          // Nenhum retorno é necessário em

            distance () {                // método para calcular a distância de
                origem para apontar.
                    retornar math.sqrt (    // retorna a raiz quadrada de x² +
                    Y².
                        this.x * this.x +   // Isso se refere ao ponto
                    objeto em qual
                        this.y * this.y      // O método de distância é
                    invocado.
                    );
                }

// Use a função do construtor Point () com "novo" para
criar objetos de ponto
Seja p = novo ponto (1, 1);    // O ponto geométrico (1,1).

// Agora use um método do objeto Point P P.Distance () // =>
Math.sqrt2
```

Este passeio introdutório da sintaxe e dos recursos fundamentais do JavaScript termina aqui, mas o livro continua com capítulos independentes que cobrem recursos adicionais do idioma:

Mostra como o código JavaScript em um arquivo ou script pode usar funções e classes JavaScript definidas em outros arquivos ou scripts.

Capítulo 11, A Biblioteca Padrão JavaScript

Abrange as funções e classes integradas que estão disponíveis para todos os programas JavaScript. Isso inclui estruturas de dados importantes como mapas e conjuntos, uma classe de expressão regular para correspondência de padrões textuais, funções para serializar estruturas de dados JavaScript e muito mais.

Capítulo 12, Iteradores e Geradores

Explica como o loop for/of funciona e como você pode tornar suas próprias classes iteráveis com for/of. Também cobre funções geradoras e a declaração de rendimento.

Capítulo 13, JavaScript assíncrono

Este capítulo é uma exploração aprofundada da programação assíncrona em JavaScript, cobrindo retornos de chamada e eventos, APIs baseadas em Promise e as palavras-chave async e await. Embora a linguagem JavaScript principal não seja assíncrona, as APIs assíncronas são o padrão nos navegadores da Web e no Node, e este capítulo explica as técnicas para trabalhar com essas APIs.

Capítulo 14, Metaprogramação

Apresenta uma série de recursos avançados de JavaScript que podem ser de interesse para programadores que escrevem bibliotecas de código para outros programadores JavaScript usarem.

Capítulo 15, JavaScript em navegadores da Web

Apresenta o ambiente host do navegador da Web, explica como os navegadores da Web executam código JavaScript e aborda as mais importantes das muitas APIs definidas pelos navegadores da Web. Este é de longe o mais longo

Capítulo no livro.

Capítulo 16, JavaScript do lado do servidor com nó

Introduz o ambiente do host Node, cobrindo o modelo de programação fundamental e as estruturas de dados e APIs mais importantes de entender.

Capítulo 17, Ferramentas e extensões JavaScript

Abrange ferramentas e extensões de idiomas que valem a pena saber porque são amplamente utilizadas e podem torná-lo um programador mais produtivo.

1.4 Exemplo: histogramas de frequência do personagem

Este capítulo termina com um programa JavaScript curto, mas não trivial. O Exemplo 1-1 é um programa de nós que lê o texto da entrada padrão, calcula um histograma de frequência de caracteres a partir desse texto e, em seguida, imprime o histograma. Você pode invocar o programa como este para analisar a frequência do caractere de seu próprio código -fonte:

```
$ node charfreq.js <charfreq.js t:  
##### 11,22% e: ##### 10,15%  
  
R: ##### 6,68%  
s: ##### 6,44% A:  
##### 6,16% n:  
#### 5,81% o:  
### 5,45% I:  
### 4,54% h:  
### 4,07% c: ###  
3,36% l: ### 3,20%  
u: ### 3,08% /:  
### 2,88%
```

Este exemplo usa vários recursos avançados de JavaScript e destina-se a demonstrar como os programas JavaScript do mundo real podem ser. Você ainda não deve esperar entender todo o código, mas tenha certeza de que tudo isso será explicado nos capítulos a seguir.

Exemplo 1-1. Histogramas de frequência de caracteres de computação com JavaScript

```
/**  
 * Este programa de nó lê texto da entrada padrão, calcula a  
 * frequência  
 * de cada letra nesse texto e exibe um histograma do mais  
 * caracteres usados com frequência. Requer o nó 12  
 * ou superior para ser executado.  
 *  
 * Em um ambiente do tipo Unix, você pode invocar o programa como  
 este:  
 *      node charfreq.js <corpus.txt  
 */  
  
// Esta classe estende mapa para que o método get () retorne o  
especificado  
// valor em vez de nulo quando a chave não está na classe MAP Classe  
DefaultMap estende o mapa {  
    construtor (defaultValue) {  
        super();                                // Invoca a Superclass  
        construtor  
            this.defaultValue = defaultValue; // lembre -se do  
            valor padrão  
    }  
  
    Get (key) {  
        if (this.has (key)) {                    // se a chave for  
            já no mapa  
                retornar super.get (chave);        // retorna seu valor  
                da superclass.  
        } outro  
        {  
            retornar this.defaultValue;          // de outra forma retornar  
            o valor padrão  
        }  
    }
```

```

} }

// Esta classe calcula e exibe histogramas de frequêcia de letras class
Histogram {
    construtor() {
        this.letterCounts = new DefaultMap(0);           //Mapa de
        letras para contar
        this.totalLetters = 0;                         // Quantos
        letras em tudo
    }

// Esta função atualiza o histograma com as letras do texto.

adicionar(texto) {
    //Remove os espaços em branco do texto e converte para
    maiúsculas
    texto = texto.replace(/\s/g, ""&quot;").toUpperCase();

    // Agora percorre os caracteres do texto for(let character of
    text) {
        deixe contar = this.letterCounts.get(personagem); ///
        obter contagem antiga
        this.letterCounts.set(caractere, contagem+1);      ///
        Aumente
        this.totalLetters++; }

}

// Converte o histograma em uma string que exibe um gráfico ASCII

toString() {
    // Converte o mapa em um array de arrays [key,value] let entradas
    = [...this.letterCounts];

    // Classifica o array por contagem e depois em ordem
    alfabeticaentries.sort((a,b) => { // Uma função para
definir a ordem de classificação.
        se (a[1] === b[1]) {                      //Se a contagem
são iguais
            retornar a[0] < b[0] ? -1: 1; // organizar
alfabeticamente.
        } outro {                                //Se a contagem
}
}

```

```

diferir
        retornar b[1] - a[1];           //classifica pelo maior
contar.
    } });

// Converte as contagens em porcentagens for(let input of records)
{
    entrada[1] = entrada[1] / this.totalLetters*100; }

// Elimine quaisquer entradas menores que 1% entradas
= entradas.filter(entry => entrada[1] >= 1);

// Agora converta cada entrada em uma linha de texto let linhas =
entradas.map(
    ([1,n]) => `${1}: ${".".repeat(Math.round(n))}`
${n.toFixed(2)}%
);
}

// E retorna as linhas concatenadas, separadas por
caracteres de nova linha.
return linhas.join("\n");
}

// Esta função assíncrona (retorno de promessa) cria um objeto
Histograma,
// lê de forma assíncrona pedaços de texto da entrada padrão e
adiciona esses pedaços a
// o histograma. Quando chega ao final do stream, ele retorna
este histograma
função assíncrona histogramFromStdin() {
process.stdin.setEncoding("utf-8"); //Lê Unicode
strings, não bytes
    deixe histograma = novo histograma(); para
    aguardar (deixe um pedaço de process.stdin) {
        histograma.add(pedaço); }

histograma de retorno; }

// Esta última linha de código é o corpo principal do programa.

```

```
// Cria um objeto Histograma a partir da entrada padrão e depois
imprime o histograma.
histogramaFromStdin().then(histograma =>
{ console.log(histograma.toString()); });
```

1.5 Resumo

Este livro explica o JavaScript de baixo para cima. Isso significa que começamos com detalhes de baixo nível, como comentários, identificadores, variáveis e tipos; em seguida, construa expressões, instruções, objetos e funções; e depois cobrir abstrações de linguagem de alto nível, como classes e módulos. Levo a sério a palavra definitivo no título deste livro, e os próximos capítulos explicam a linguagem em um nível de detalhe que pode parecer desanimador à primeira vista. O verdadeiro domínio do JavaScript requer um compreensão dos detalhes, no entanto, e espero que você reserve tempo para ler este livro de capa a capa. Mas, por favor, não sinta que precisa fazer isso na primeira leitura. Se você se sentir atolado em uma seção, simplesmente pule para a próxima. Você pode voltar e dominar os detalhes assim que tiver um conhecimento prático do idioma como um todo.

Capítulo 2. Estrutura Lexical

A estrutura lexical de uma linguagem de programação é o conjunto de regras elementares que especifica como você escreve programas nessa linguagem. É a sintaxe de nível mais baixo de uma linguagem: especifica a aparência dos nomes das variáveis, os caracteres delimitadores para comentários e como uma instrução de programa é separada da próxima, por exemplo. Este breve capítulo documenta a estrutura lexical do JavaScript. Abrange:

- Sensibilidade a maiúsculas e minúsculas, espaços e quebras de linha
- Comentários
- Literais
- Identificadores e palavras reservadas
- Unicode
- Ponto-e-vírgula opcional

2.1 O texto de um programa JavaScript

JavaScript é uma linguagem que diferencia maiúsculas de minúsculas. Isso significa que palavras-chave de linguagem, variáveis, nomes de funções e outros identificadores devem sempre ser digitados com letras maiúsculas consistentes. A palavra-chave `while`, por exemplo, deve ser digitada “`while`”, não “`While`” ou “`WHILE`”. Da mesma forma, `online`, `Online`, `OnLine` e `ONLINE` são quatro nomes de variáveis distintos.

JavaScript ignora espaços que aparecem entre tokens em programas. Na maior parte, JavaScript também ignora quebras de linha (mas veja §2.6 para uma exceção). Como você pode usar espaços e novas linhas livremente em seus programas, você pode formatar e recuar seus programas de maneira organizada e consistente, tornando o código fácil de ler e entender.

Além do caractere de espaço regular (\u0020), o JavaScript também reconhece tabulações, diversos caracteres de controle ASCII e vários caracteres de espaço Unicode como espaços em branco. JavaScript reconhece novas linhas, retornos de carro e uma sequência de retorno de carro/alimentação de linha como terminadores de linha.

2.2 Comentários

JavaScript oferece suporte a dois estilos de comentários. Qualquer texto entre // e o final de uma linha é tratado como um comentário e ignorado pelo JavaScript. Qualquer texto entre os caracteres /* e */ também é tratado como comentário; esses comentários podem abranger várias linhas, mas não podem estar aninhados. As linhas de código a seguir são comentários legais de JavaScript:

```
// Este é um comentário de linha única.

/* Isto também é um comentário */ // e aqui está outro comentário.

/*
 * Este é um comentário de várias linhas. Os caracteres *
 * extras no início de
 * cada linha não é uma parte obrigatória da sintaxe; eles
 * simplesmente parecem legais!
 */
```

2.3 Literais

Um literal é um valor de dados que aparece diretamente em um programa. A seguir estão todos literais:

```
12           // O número doze
1.2          // O número um ponto dois
&quot;olá mundo&quot; // Uma string de texto
'oi'         // Outra string
verdadeir
o           // Um valor booleano
falso        // O outro valor booleano
nulo         // Ausência de um objeto
```

Detalhes completos sobre literais numéricos e de string aparecem no Capítulo 3.

2.4 Identificadores e Palavras Reservadas

Um identificador é simplesmente um nome. Em JavaScript, os identificadores são usados para nomear constantes, variáveis, propriedades, funções e classes e para fornecer rótulos para determinados loops no código JavaScript. Um identificador JavaScript deve começar com uma letra, um sublinhado (_) ou um cifrão (\$). Os caracteres subsequentes podem ser letras, dígitos, sublinhados ou cifrões. (Dígitos não são permitidos como primeiro caractere para que o JavaScript possa distinguir facilmente identificadores de números.) Estes são todos identificadores legais:

```
eu_meu_nome_variável
v13

manequ
im $str
```

Como qualquer linguagem, o JavaScript reserva certos identificadores para uso da própria linguagem. Estas “palavras reservadas” não podem ser usadas como identificadores regulares. Eles estão listados na próxima seção.

2.4.1 Palavras Reservadas

As palavras a seguir fazem parte da linguagem JavaScript. Muitas delas (como if, while e for) são palavras-chave reservadas que não devem ser usadas como nomes de constantes, variáveis, funções ou classes (embora todas possam ser usadas como nomes de propriedades dentro de um objeto). Outros (como from, of, get e set) são usados em contextos limitados sem ambiguidade sintática e são perfeitamente legais como identificadores. Ainda outras palavras-chave (como let) não podem ser totalmente reservadas para manter a compatibilidade retroativa com programas mais antigos e, portanto, existem regras complexas que governam quando elas podem ser usadas como identificadores e quando não podem. (let pode ser usado como um nome de variável se declarado com var fora de uma classe, por exemplo, mas não se declarado dentro de uma classe ou com const.) O caminho mais simples é evitar o uso de qualquer uma dessas palavras como identificadores, exceto from , set e target, que são seguros para uso e já são de uso comum.

como	const	exportar	pegar	nulo	alvo
anular					
assíncrono	continuar	estende	se de		esse
enquanto					
espero	depurador	falso	importar	retornar	lançar
com					
intervalo	padrão	finalmente	em definir		verdadeir o
caso de					
rendimento	excluir	para	instância de	estático	tentar
pegar	do	de	deixar	super	tipo de
aula	outro	função	novo	trocar	nosso

JavaScript também reserva ou restringe o uso de certas palavras-chave que não são usadas atualmente pela linguagem, mas que podem ser usadas em versões futuras:

enumeração	implementos	interface	pacote	privado	protegido
------------	-------------	-----------	--------	---------	-----------

público

Por razões históricas, argumentos e avaliação não são permitidos como identificadores em certas circunstâncias e são melhor evitados inteiramente.

2.5 Unicode

Os programas JavaScript são escritos usando o conjunto de caracteres Unicode e você pode usar qualquer caractere unicode em strings e comentários. Para portabilidade e facilidade de edição, é comum usar apenas letras e dígitos ASCII em identificadores. Mas esta é apenas uma convenção de programação, e o idioma permite letras, dígitos e ideografias unicode (mas não emojis) nos identificadores. Isso significa que os programadores podem usar

Símbolos e palavras matemáticas de idiomas não ingleses como constantes e variáveis:

```
const π = 3,14;  
Const si = true;
```

2.5.1 Sequências de Escape Unicode

Alguns hardware e software de computador não podem exibir, entrar ou processar corretamente o conjunto completo de caracteres Unicode. Para apoiar programadores e sistemas usando a tecnologia mais antiga, o JavaScript define sequências de fuga que nos permitem escrever caracteres Unicode usando apenas caracteres ASCII. Essas escapadas unicode começam com os caracteres \ u e são seguidos por exatamente quatro dígitos hexadecimais (usando letras maiúsculas ou minúsculas A - F) ou por um a seis dígitos hexadecimais incluídos em chaves encaracolados. Essas fugas de unicode podem aparecer em literais de cordas JavaScript, literais de expressão regular e identificadores (mas

não em palavras-chave de idioma). O escape Unicode para o caractere “é”, por exemplo, é \u00E9; aqui estão três maneiras diferentes de escrever um nome de variável que inclua este caractere:

```
deixe café = 1; // Defina uma variável usando um
caractere Unicode caf\u00e9 // => 1; acesse a
variável usando uma sequência de escape caf\u{E9} // 
=> 1; outra forma da mesma sequência de escape
```

As primeiras versões do JavaScript suportavam apenas a sequência de escape de quatro dígitos. A versão com chaves foi introduzida no ES6 para melhor suportar codepoints Unicode que requerem mais de 16 bits, como emoji:

```
console.log(""\u{1F600}"" // Imprime um emoji de carinha sorridente
t);
```

Os escapes Unicode também podem aparecer nos comentários, mas como os comentários são ignorados, eles são simplesmente tratados como caracteres ASCII nesse contexto e não interpretados como Unicode.

2.5.2 Normalização Unicode

Se você usar caracteres não ASCII em seus programas JavaScript, deverá estar ciente de que o Unicode permite mais de uma maneira de codificar o mesmo caractere. A string “é”, por exemplo, pode ser codificada como um único caractere Unicode \u00E9 ou como um “e” ASCII regular seguido pela marca de combinação de acento agudo \u0301. Essas duas codificações normalmente parecem exatamente iguais quando exibidas por um editor de texto, mas possuem codificações binárias diferentes, o que significa que são consideradas diferentes pelo JavaScript, o que pode levar a programas muito confusos:

```
const café = 1;      // Esta constante é nomeada "Café" e tem o valor 1;
const café = 2;      // Esta constante é diferente: "Café" e tem o valor 2;
café    // > 1: esta constante tem um valor
Café // > 2: Esta constante indistinguível tem um
valor diferente
```

O padrão Unicode define a codificação preferida para todos os caracteres e especifica um procedimento de normalização para converter o texto em uma forma canônica adequada para comparações. O JavaScript assume que o código -fonte que ele está interpretando já foi normalizado e não faz nenhuma normalização por conta própria. Se você planeja usar caracteres Unicode em seus programas JavaScript, verifique se o seu editor ou outra ferramenta executa a normalização do Unicode do seu código -fonte para impedir que você acabe com identificadores diferentes, mas visualmente indistinguíveis.

2.6 Semicolons opcionais

Como muitas linguagens de programação, o JavaScript usa o semicolon (;) para separar declarações (consulte o Capítulo 5) um do outro. Isso é importante para esclarecer o significado do seu código: sem um separador, o final de uma declaração pode parecer o começo do próximo ou vice -versa. No JavaScript, geralmente você pode omitir o ponto e vírgula entre duas declarações se essas instruções forem escritas em linhas separadas. (Você também pode omitir um ponto e vírgula no final de um programa ou se o próximo token do programa é uma cinta encurrallada: {}.) Muitos programadores JavaScript (e o código neste livro) usam semicolons para marcar explicitamente os fins das declarações , mesmo onde eles não são necessários. Outro estilo é omitir os semicolons sempre que possível, usá -los apenas nas poucas situações que os exigem. Qualquer estilo que você

Escolha, existem alguns detalhes que você deve entender sobre os semicolons opcionais no JavaScript.

Considere o seguinte código. Como as duas declarações aparecem em linhas separadas, o primeiro ponto de vírgula pode ser omitido:

```
a = 3;  
b = 4;
```

Escrito da seguinte forma, no entanto, é necessário o primeiro ponto de vírgula:

```
a = 3; b = 4;
```

Observe que o JavaScript não trata todas as quebras de linha como um semicolon: geralmente trata as quebras de linha como semicolons apenas se não puder analisar o código sem adicionar um semicolon implícito. Mais formalmente (e com três exceções descritas um pouco mais tarde), o JavaScript trata uma quebra de linha como um semicolon se o próximo caractere não espacial não puder ser interpretado como uma continuação da declaração atual. Considere o seguinte código:

```
deixe aa =  
3  
console.log  
(a)
```

JavaScript interpreta este código como este:

```
deixe um; a = 3; console.log (a);
```

O JavaScript trata a primeira ruptura da primeira linha como um ponto e vírgula porque não pode analisar o código, deixe AA sem um ponto de vírgula. O segundo A pode ficar sozinho como a declaração a;, mas JavaScript não trata o

Segunda linha quebra como um semicolon porque pode continuar analisando a declaração mais longa a = 3;.

Essas regras de rescisão de declaração levam a alguns casos surpreendentes. Este código parece duas declarações separadas separadas com uma nova linha:

```
Seja y = x + f (a  
+ b) .ToString ()
```

Mas os parênteses na segunda linha de código podem ser interpretados como uma invocação de função de F a partir da primeira linha, e JavaScript interpreta o código como este:

```
Seja y = x + f (a + b) .ToString ();
```

Provavelmente do que não, essa não é a interpretação pretendida pelo autor do Código. Para funcionar como duas declarações separadas, é necessário um semicolon explícito neste caso.

Em geral, se uma declaração começar com (, /, +, ou -, há uma chance de ser interpretada como uma continuação da declaração antes. Declarações que começam com /, +e - são bastante raras na prática , mas declarações começando com (e [não são incomuns, pelo menos em alguns estilos de programação de JavaScript. Alguns programadores gostam de colocar um semicolon defensivo no início de qualquer afirmação, para que continue funcionando corretamente, mesmo que a declaração Antes de ser modificado e um semicolon anteriormente removido:

```
Seja x = 0 // semicolon omitido aqui  
; [x, x+1, x+2] .foreach (console.log) // defensivo; Mantém esta  
declaração separada
```

Existem três exceções à regra geral de que o JavaScript interpreta quebras de linha como ponto e vírgula quando não pode analisar a segunda linha como uma continuação da instrução na primeira linha. A primeira exceção envolve as instruções `return`, `throw`, `yield`, `break` e `continue` (consulte o Capítulo 5). Essas declarações geralmente são independentes, mas às vezes são seguidas por um identificador ou expressão. Se uma quebra de linha aparecer após qualquer uma dessas palavras (antes de qualquer outro token), o JavaScript sempre interpretará essa quebra de linha como ponto e vírgula. Por exemplo, se você escrever:

```
retornar  
verdadeiro;
```

JavaScript pressupõe que você quis dizer:

```
retornar; verdadeiro;
```

No entanto, você provavelmente quis dizer:

```
retornar verdadeiro;
```

Isso significa que você não deve inserir uma quebra de linha entre `return`, `break` ou `continue` e a expressão que segue a palavra-chave. Se você inserir uma quebra de linha, seu código provavelmente falhará de uma forma não óbvia e difícil de depurar.

A segunda exceção envolve os operadores `++` e `--` (§4.8). Esses operadores podem ser operadores de prefixo que aparecem antes de uma expressão ou operadores pós-fixados que aparecem depois de uma expressão. Se você quiser usar qualquer um desses operadores como operadores pós-fixados, eles deverão aparecer no

mesma linha da expressão à qual se aplicam. A terceira exceção envolve funções definidas usando uma sintaxe concisa de “seta”: a própria seta => deve aparecer na mesma linha da lista de parâmetros.

2.7 Resumo

Este capítulo mostrou como os programas JavaScript são escritos no nível mais baixo. O próximo capítulo nos leva um passo além e apresenta os tipos e valores primitivos (números, strings e assim por diante) que servem como unidades básicas de computação para programas JavaScript.

Capítulo 3. Tipos, Valores e Variáveis

Os programas de computador funcionam manipulando valores, como o número 3,14 ou o texto “Hello World”. Os tipos de valores que podem ser representados e manipulados em uma linguagem de programação são conhecidos como tipos e uma das características mais fundamentais de uma linguagem de programação.

linguagem de programação é o conjunto de tipos que ela suporta. Quando um programa precisa reter um valor para uso futuro, ele atribui o valor a (ou “armazena” o valor em) uma variável. As variáveis têm nomes e permitem o uso desses nomes em nossos programas para se referir a valores. A forma como as variáveis funcionam é outra característica fundamental de qualquer linguagem de programação. Este capítulo explica tipos, valores e variáveis em JavaScript. Começa com uma visão geral e algumas definições.

3.1 Visão Geral e Definições

Os tipos JavaScript podem ser divididos em duas categorias: tipos primitivos e tipos de objetos. Os tipos primitivos do JavaScript incluem números, strings de texto (conhecidas como strings) e valores de verdade booleanos (conhecidos como booleanos). Uma parte significativa deste capítulo é dedicada a uma análise detalhada

explicação dos tipos numérico (§3.2) e string (§3.3) em JavaScript. Booleanos são abordados em §3.4.

Os valores especiais de JavaScript nulos e indefinidos são primitivos

valores, mas não são números, cordas ou booleanos. Cada valor é normalmente considerado o único membro de seu próprio tipo especial. §3.5 tem mais sobre nulo e indefinido. O ES6 adiciona um novo tipo de propósito especial, conhecido como símbolo, que permite a definição de extensões de linguagem sem prejudicar a compatibilidade com versões anteriores. Os símbolos são abordados brevemente no §3.6.

Qualquer valor JavaScript que não seja um número, uma string, um booleano, um símbolo, nulo ou indefinido é um objeto. Um objeto (ou seja, um membro do objeto de tipo) é uma coleção de propriedades em que cada propriedade tem um nome e um valor (um valor primitivo ou outro objeto). Um objeto muito especial, o objeto global, é abordado no §3.7, mas a cobertura mais geral e mais detalhada dos objetos está no capítulo 6.

Um objeto JavaScript comum é uma coleção não ordenada de valores nomeados. O idioma também define um tipo especial de objeto, conhecido como uma matriz, que representa uma coleção ordenada de valores numerados. A linguagem JavaScript inclui sintaxe especial para trabalhar com matrizes, e as matrizes têm algum comportamento especial que as distingue de objetos comuns. Matrizes são o assunto do capítulo 7.

Além de objetos e matrizes básicos, o JavaScript define vários outros tipos de objetos úteis. Um objeto definido representa um conjunto de valores. Um objeto de mapa representa um mapeamento de chaves para valores. Vários tipos de “matriz digitada” facilitam operações em matrizes de bytes e outros dados binários. O tipo regexp representa padrões textuais e permite correspondência sofisticada, pesquisa e substituição de operações em seqüências de caracteres. O tipo de data representa datas e horários e suporta a data rudimentar aritmética. Erro e seus subtipos representam erros que podem surgir

Ao executar o código JavaScript. Todos esses tipos são abordados no capítulo 11.

O JavaScript difere de idiomas mais estáticos, pois funções e classes não fazem apenas parte da sintaxe da linguagem: eles são os próprios valores que podem ser manipulados pelos programas JavaScript. Como qualquer valor JavaScript que não seja um valor primitivo, as funções e as classes são um tipo de objeto especializado. Eles são cobertos em detalhes nos capítulos 8 e 9.

O intérprete JavaScript executa a coleta automática de lixo para gerenciamento de memória. Isso significa que um programador JavaScript geralmente não precisa se preocupar com a destruição ou desalocação de objetos ou outros valores. Quando um valor não é mais acessível - quando um programa não tem mais maneira de se referir a ele - o intérprete sabe que nunca pode ser usado novamente e recupera automaticamente a memória que estava ocupando. (Os programadores JavaScript às vezes precisam tomar cuidado para garantir que os valores não permaneçam inadvertidamente alcançáveis - e, portanto, não reclamáveis - mais que o necessário.)

O JavaScript suporta um estilo de programação orientado a objetos. Pouco, isso significa que, em vez de ter funções definidas globalmente para operar em valores de vários tipos, os próprios tipos definem métodos para trabalhar com valores. Para classificar os elementos de uma matriz A, por exemplo, não passamos a uma função de classificação (). Em vez disso, invocamos o método Sort () de A:

```
a.sort ();           // A versão orientada ao objeto (a).
```

A definição de métodos é abordada no Capítulo 9. Tecnicamente, apenas objetos JavaScript possuem métodos. Mas números, strings, valores booleanos e símbolos se comportam como se tivessem métodos. Em JavaScript, nulo e indefinido são os únicos valores nos quais os métodos não podem ser invocados.

Os tipos de objetos do JavaScript são mutáveis e seus tipos primitivos são imutáveis. Um valor de tipo mutável pode mudar: um programa JavaScript pode alterar os valores das propriedades do objeto e dos elementos do array. Números, booleanos, símbolos, nulos e indefinidos são imutáveis — nem faz sentido falar em alterar o valor de um número, por exemplo. Strings podem ser consideradas matrizes de caracteres e você pode esperar que sejam mutáveis. Em JavaScript, entretanto, as strings são imutáveis: você pode acessar o texto em qualquer índice de uma string, mas o JavaScript não oferece nenhuma maneira de alterar o texto de uma string existente. As diferenças entre valores mutáveis e imutáveis são exploradas mais detalhadamente em §3.8.

JavaScript converte livremente valores de um tipo para outro. Se um programa espera uma string, por exemplo, e você fornece um número, ele converterá automaticamente o número em uma string para você. E se você usar um valor não booleano onde um booleano é esperado, o JavaScript será convertido de acordo. As regras para conversão de valor são explicadas em §3.9. As regras liberais de conversão de valor do JavaScript afetam sua definição de igualdade, e o operador `==` de igualdade realiza conversões de tipo conforme descrito em §3.9.1. (Na prática, entretanto, o operador de igualdade `==` está obsoleto em favor do operador de igualdade estrito `===`, que não faz conversões de tipo. Consulte §4.9.1 para obter mais informações sobre ambos os operadores.)

Constantes e variáveis permitem que você use nomes para se referir a valores em seus programas. As constantes são declaradas com const e as variáveis são declaradas com LET (ou com var no código JavaScript mais antigo). As constantes e variáveis de JavaScript não estão sem forma: as declarações não especificam que tipo de valores serão atribuídos. A declaração e a atribuição variáveis são cobertas no §3.10.

Como você pode ver nesta longa introdução, este é um capítulo abrangente que explica muitos detalhes fundamentais sobre como os dados são representados e manipulados em JavaScript. Começaremos mergulhando diretamente nos detalhes dos números e texto de JavaScript.

3.2 números

O número numérico principal do JavaScript, número, é usado para representar números inteiros e aproximar números reais. O JavaScript representa números usando o formato de ponto flutuante de 64 bits definido pelo padrão IEEE 754, o que significa que pode representar números tão grandes quanto $\pm 1,7976931348623157 \times 10$ e tão pequeno quanto $\pm 5 \times 10^{-324}$.

308 –324

O formato do número JavaScript permite representar exatamente todos os números inteiros entre -9.007.199.254.740.992 (-2) e 9.007.199.254.740.992 (2), inclusive. Se você usar valores inteiros maiores que isso, poderá perder precisão nos dígitos à direita. Observe, no entanto, que certas operações no JavaScript (como indexação de matrizes e os operadores bites descritos no capítulo 4) são realizados com números inteiros de 32 bits. Se você precisar representar exatamente números inteiros maiores, consulte §3.2.5.

Quando um número aparece diretamente em um programa JavaScript, ele é chamado de

literal numérico. JavaScript oferece suporte a literais numéricos em vários formatos, conforme descrito nas seções a seguir. Observe que qualquer literal numérico pode ser precedido por um sinal de menos (-) para tornar o número negativo.

3.2.1 Literais Inteiros

Em um programa JavaScript, um número inteiro de base 10 é escrito como uma sequência de dígitos. Por exemplo:

```
0 3  
100000000
```

Além dos literais inteiros de base 10, o JavaScript reconhece valores hexadecimais (base 16). Um literal hexadecimal começa com 0x ou 0X, seguido por uma sequência de dígitos hexadecimais. Um dígito hexadecimal é um dos dígitos de 0 a 9 ou as letras de a (ou A) a f (ou F), que representam valores de 10 a 15. Aqui estão exemplos de

literais inteiros hexadecimais:

```
0xff      // => 255: (15*16 + 15)  
0xBADCAFE // => 195939070
```

No ES6 e posteriores, você também pode expressar inteiros em binário (base 2) ou octal (base 8) usando os prefixos 0b e 0o (ou 0B e 0O) em vez de 0x:

```
0b10101   // => 21:    (1*16 + 0*8 + 1*4 + 0*2 + 1*1)  
0o377     // => 255: (3*64 + 7*8 + 7*1)
```

3.2.2 Literais de Ponto Flutuante

Literais de ponto flutuante podem ter um ponto decimal; eles usam o tradicional

sintaxe para números reais. Um valor real é representado como a parte integrante do número, seguida por uma vírgula decimal e a parte fracionária do número.

Literais de ponto flutuante também podem ser representados usando notação exponencial: um número real seguido pela letra e (ou E), seguido por um sinal opcional de mais ou menos, seguido por um expoente inteiro. Esta notação representa o número real multiplicado por 10 elevado à potência do expoente.

Mais sucintamente, a sintaxe é:

```
[dígitos][.dígitos][(E|e)[(+|-)]dígitos]
```

Por exemplo:

```
3,14 2345,6789  
.3333333333333333 6,02e23  
// 6,02 x 1023 1,4738223E-  
32 // 1,4738223 x 10-32
```

SEPARADORES EM LITERAIS NUMÉRICOS

Você pode usar sublinhados em literais numéricos para dividir literais longos em partes que são mais fáceis de ler:

```
deixe bilhão = 1_000_000_000;      // Sublinhado como separador de milhares.  
deixe bytes = 0x89_AB_CD_EF;      // Como separador de bytes.  
deixe bits = 0b0001_1101_0111;    // Como separador de mordidela.  
deixe fração = 0,123_456_789;     // Funciona na parte fracionária também.
```

No momento em que este artigo foi escrito, no início de 2020, os sublinhados em literais numéricos ainda não estavam formalmente padronizados como parte do JavaScript. Mas eles estão em estágios avançados do processo de padronização e são implementados por todos os principais navegadores e pelo Node.

3.2.3 aritmética em javascript

Os programas JavaScript funcionam com números usando os operadores aritméticos que o idioma fornece. Isso inclui + para adição, - para subtração, * para multiplicação, / para divisão e % para módulo (restante após a divisão). O ES2016 adiciona ** para exponenciação. Detalhes completos sobre esses e outros operadores podem ser encontrados no capítulo 4.

Além desses operadores aritméticos básicos, o JavaScript suporta operações matemáticas mais complexas por meio de um conjunto de funções e constantes definidas como propriedades do objeto de matemática:

```
Math.pow (2,53)           // => 9007199254740992: 2 para o
potência 53
Math.Round (.6)           // => 1,0: rodada para o mais próximo
Inteiro
math.ceil (.6)            // => 1.0: Recurso para um número inteiro
Math.floor (.6)           // => 0,0: Recurt -in para um número inteiro
Math.abs (-5)              // => 5: Valor absoluto
Math.max (x, y, z)         // retorna o maior argumento
Math.min (x, z)             // retorna o menor argumento
Math.Random ()             // Número pseudo-random x onde 0 <=
x <1,0 math.pi             // π: circunferência de um círculo /
diameter Math.E             // e: a base do natural
Logarithm
Math.Sqrt (3)               // => 3 ** 0.5: a raiz quadrada de 3
Math.pow (3, 1/3)            // => 3 ** (1/3): a raiz do cubo de 3
Math.sin (0)                 // Trigonometria: também Math.cos,
Math.atan, etc. Math.log
(10)                       // logaritmo natural de 10
Math.log (100) /math.ln10      // base 10 logaritmo de 100
Math.log (512) /math.ln2        // Base 2 logaritmo de 512
Math.exp (3)                  // Math.e Cubed
```

ES6 define mais funções no objeto de matemática:

```
Math.cbrt (27)      // => 3: raiz de cubo
Math.hypot (3, 4) // => 5: raiz quadrada da soma dos quadrados de
                   todos os argumentos
Math.Log10 (100)    // => 2: Logaritmo Base-10
Math.log2 (1024)    // => 10: logaritmo base-2
Math.log1p (x)      // log natural de (1+x); preciso para muito
                   pequeno x
math.expm1 (x)      // math.exp (x) -1; o inverso de
Math.log1p ()
math.sign (x)        // -1, 0 ou 1 para argumentos <0, == ou> 0
Math.imul (2,3)      // => 6: multiplicação otimizada de 32 bits
Inteiros
Math.clz32 (0xf)    // => 28: Número de zero bits líderes em um
Math.Trun (3.9)
de 32 bits (3.9)   // => 3: converte para um número inteiro truncando
peça fracionária
matemática.FROUND (X) // Round para o número de flutuação de 32 bits mais próximo
Math.sinh (x)        // seno hiperbólico. Também Math.Cosh (),
Math.tanh ()          // Arcsina hiperbólica. Também math.acosh (),
math.asinh (x)        // math.atanh ()
```

A aritmética em JavaScript não levanta erros nos casos de transbordamento, subfluxo ou divisão por zero. Quando o resultado de uma operação numérica é maior que o maior número representável (estouro), o resultado é um valor especial do infinito, o infinito. Da mesma forma, quando o valor absoluto de um valor negativo se torna maior que o valor absoluto do maior número negativo representável, o resultado é infinito negativo, - infinito. Os valores infinitos se comportam como seria de esperar: adicionar, subtrair, multiplicar ou dividir -los por qualquer coisa resulta em um valor infinito (possivelmente com o sinal revertido).

O fluxo ocorre quando o resultado de uma operação numérica está mais próxima de

zero que o menor número representável. Nesse caso, JavaScript retorna 0. Se ocorrer underflow a partir de um número negativo, JavaScript retorna um valor especial conhecido como “zero negativo”. Este valor é quase completamente indistinguível do zero normal e os programadores JavaScript raramente precisam detectá-lo.

A divisão por zero não é um erro em JavaScript: simplesmente retorna infinito ou infinito negativo. Há uma exceção, porém: zero dividido por zero não tem um valor bem definido, e o resultado desta operação é o valor especial que não é um número, NaN. NaN também surge se você tentar dividir infinito por infinito, extrair a raiz quadrada de um número negativo ou usar operadores aritméticos com operandos não numéricos que não podem ser convertidos em números.

JavaScript predefine as constantes globais Infinity e NaN para conter o valor infinito positivo e não um número, e esses valores também estão disponíveis como propriedades do objeto Number:

```
Infinidade          // Um número positivo muito grande para
representa o
número.POSITIVE_INFINITY // Mesmo valor
1/0                // => Infinito
Número.MAX_VALUE * 2 // => Infinito; transbordar

-Infinidade        // Um número negativo muito grande para
representa o
número.NEGATIVE_INFINITY // O mesmo valor
-1/0               // => -Infinito
-Número.MAX_VALUE * 2 // => -Infinito

NaN                // O valor que não é um número
Número.NaN         // O mesmo valor, escrito
outra maneira 0/0 // => NaN
Infinito/Infinito // => NaN
```

```

Número.MIN_VALUE/2           // => 0: estouro negativo
-Número.MIN_VALUE/2          // => -0: zero negativo
-1/Infinito                  // -> -0: também 0 negativo
-0

// As seguintes propriedades Number são definidas no ES6
Number.parseInt() // Igual à função global parseInt()

Número.parseFloat()          // Igual ao parseFloat() global
função Número.isNaN(x)       // x é o valor NaN?
Número.isFinite(x)           // Xa é número e finito?
Número.isInteger(x)          //X é um número inteiro?
Number.isSafeInteger(x) // X é um número inteiro -(2**53) < x <
2**53?
Número.MIN_SAFE_INTEGER // => -(2**53 - 1)
Number.MAX_SAFE_INTEGER // => 2**53 - 1 Number.EPSILON
// => 2**-52: menor diferença entre números

```

O valor não-um-número tem uma característica incomum em JavaScript: ele não se compara a nenhum outro valor, incluindo ele mesmo. Isso significa que você não pode escrever `x === NaN` para determinar se o valor de uma variável `x` é `NaN`. Em vez disso, você deve escrever `x! = x` ou

`Número.isNaN(x)`. Essas expressões serão verdadeiras se, e somente se, `x` tiver o mesmo valor que a constante global `NaN`.

A função global `isNaN()` é semelhante a `Number.isNaN()`. Ele retorna verdadeiro se seu argumento for `NaN` ou se esse argumento for um valor não numérico que não pode ser convertido em um número. A função relacionada `Number.isFinite()` retorna verdadeiro se seu argumento for um número diferente de `NaN`, `Infinity` ou `-Infinity`. A função `isFinite()` global retorna verdadeiro se seu argumento for ou puder ser convertido em um número finito.

O valor zero negativo também é um tanto incomum. Ele compara igual (mesmo usando o teste de igualdade estrito do JavaScript) com zero positivo, o que significa que os dois valores são quase indistinguíveis, exceto quando usados como divisor:

```
seja zero = 0;           //Zero normal
seja negz = -0;          //Zero negativo
zero === negativo        // => verdadeiro: zero e zero negativo são
igual a 1/zero ===      // => falso: Infinity e -Infinity são
1/negz                  // não é igual
```

3.2.4 Erros binários de ponto flutuante e arredondamento

Existem infinitos números reais, mas apenas um número finito deles (18.437.736.874.454.810.627, para ser exato) pode ser representado exatamente pelo formato de ponto flutuante JavaScript. Isso significa que quando você trabalha com números reais em JavaScript, a representação do número geralmente será uma aproximação do número real.

A representação de ponto flutuante IEEE-754 usada pelo JavaScript (e por quase todas as outras linguagens de programação modernas) é uma representação binária, que pode representar exatamente frações como $1/2$, $1/8$ e $1/1024$. Infelizmente, as frações que usamos com mais frequência (especialmente ao realizar cálculos financeiros) são frações decimais: $1/10$, $1/100$ e assim por diante. Ponto flutuante binário

representações não podem representar exatamente números tão simples quanto $0,1$.

Os números JavaScript têm bastante precisão e podem se aproximar muito de $0,1$. Mas o fato de esse número não poder ser representado com exatidão pode causar problemas. Considere este código:

```
Seja x = 0.3 - .2;      // trinta centavos menos 20 centavos
Seja y = .2 - .1;       // vinte centavos menos 10 centavos
x === y                 // => false: os dois valores não são os
mesmo! x === .1         // => false: .3-.2 não é igual a .1
y === .1                // => true: .2-.1 é igual a .1
```

Devido ao erro de arredondamento, a diferença entre as aproximações de 0,3 e 0,2 não é exatamente a mesma que a diferença entre as aproximações de 0,2 e 0,1. É importante entender que esse problema não é específico para o JavaScript: afeta qualquer linguagem de programação que use números binários de ponto flutuante. Além disso, observe que os valores x e y no código mostrado aqui estão muito próximos um do outro e com o valor correto. Os valores calculados são adequados para quase qualquer finalidade; O problema só surge quando tentamos comparar valores para a igualdade.

Se essas aproximações de ponto flutuante forem problemáticas para seus programas, considere o uso de números inteiros em escala. Por exemplo, você pode manipular os valores monetários como centavos inteiros, em vez de dólares fracionários.

3.2.5 números inteiros de precisão arbitrária com bigint

Um dos recursos mais recentes do JavaScript, definido no ES2020, é um novo tipo numérico conhecido como bigint. No início de 2020, foi implementado no Chrome, Firefox, Edge e Node, e há uma implementação em andamento no Safari. Como o nome indica, Bigint é um tipo numérico cujos valores são inteiros. O tipo foi adicionado ao JavaScript principalmente para permitir a representação de números inteiros de 64 bits, necessários para a compatibilidade com muitas outras linguagens de programação

e APIs. Mas os valores BigInt podem ter milhares ou até milhões de dígitos, caso você precise trabalhar com números tão grandes. (Observe, entretanto, que as implementações do BigInt não são adequadas para criptografia porque não tentam impedir ataques de temporização.)

Literais BigInt são escritos como uma sequência de dígitos seguida por uma letra minúscula n. Por padrão, eles estão na base 10, mas você pode usar os prefixos 0b, 0o e 0x para BigInts binários, octais e hexadecimais:

```
1234n           // Um literal BigInt não tão grande
0b111111n      // Um BigInt binário
0o7777n         // Um BigInt octal
0x8000000000000000n // => 2n*63n: um número inteiro de 64 bits
```

Você pode usar BigInt() como uma função para converter números ou strings JavaScript regulares em valores BigInt:

```
BigInt(Número.MAX_SAFE_INTEGER)           // => 9007199254740991n
deixe string = "1" + "0".repeat(100); // 1 seguido de 100 zeros.
BigInt(string)                         // => 10n*100n: um
corte isso
```

A aritmética com valores BigInt funciona como a aritmética com números JavaScript regulares, exceto que a divisão elimina qualquer resto e arredonda para baixo (em direção a zero):

```
1000n + 2000n // => 3000n
3.000n - 2.000n // => 1000n
2.000n * 3.000n // => 6000000n
3000n/997n      // => 3n: o quociente é 3
3000n%997n      // => 9n: e o resto é 9
(2n ** 131071n) - 1n // Um primo de Mersenne com 39457 decimal
dígitos
```

Embora os operadores padrão +, -, *, /, % e ** trabalhem com o BIGINT, é importante entender que você não pode misturar operandos do tipo bigint com operando de números regulares. Isso pode parecer confuso no começo, mas há uma boa razão para isso. Se um tipo numérico fosse mais geral que o outro, seria fácil definir aritmética em operandos mistos para simplesmente retornar um valor do tipo mais geral. Mas nenhum dos tipos é mais geral que o outro: Bigint pode representar valores extraordinariamente grandes, tornando-o mais geral que os números regulares. Mas o Bigint só pode representar números inteiros, tornando o número regular de número JavaScript mais geral. Não há como contornar esse problema, portanto, JavaScript o desvia simplesmente não permitindo operando mistas para os operadores aritméticos.

Os operadores de comparação, por outro lado, trabalham com tipos numéricos mistos (mas consulte §3.9.1 para obter mais sobre a diferença entre == e ===):

```
1 < 2n      // => true
2>; 1n      // => true
0 == 0n      // => true
0 === 0n    // => false: o === verifica o tipo de igualdade como
            bem
```

Os operadores bitwise (descritos no §4.8.3) geralmente funcionam com operandos Bigint. No entanto, nenhuma das funções do objeto de matemática aceita operando Bigint.

3.2.6 Datas e horários

O JavaScript define uma classe de data simples para representar e manipular os números que representam datas e horários. As datas de JavaScript são objetos, mas também têm uma representação numérica como um

Timestamp que especifica o número de milissegundos decorridos desde 1º de janeiro de 1970:

```
deixe timestamp = date.now (); // a hora atual como um
Timestamp (um número).
deixe agora = new Date (); // a hora atual como uma data
objeto. deixe ms =
agora.gettime (); // converter em um milissegundo
Timestamp. vamos iso = agora.toISOString ();
// converte em uma string no formato padrão.
```

A classe de data e seus métodos são abordados em detalhes no §11.4. Mas veremos os objetos da data novamente em §3.9.3 quando examinarmos os detalhes das conversões do tipo JavaScript.

3.3 Texto

O tipo JavaScript para representar o texto é a string. Uma string é uma sequência ordenada imutável de valores de 16 bits, cada um dos quais normalmente representa um caractere unicode. O comprimento de uma string é o número de valores de 16 bits que ele contém. As cadeias de JavaScript (e suas matrizes) usam indexação baseada em zero: o primeiro valor de 16 bits está na posição 0, a segunda na posição 1 e assim por diante. A sequência vazia é a sequência de comprimento 0. O JavaScript não possui um tipo especial que representa um único elemento de uma string. Para representar um único valor de 16 bits, basta usar uma string com um comprimento de 1.

Personagens, pontos de código e strings de javascript

O JavaScript usa a codificação UTF-16 do conjunto de caracteres Unicode, e as sequências de JavaScript são sequências de valores de 16 bits não assinados. Os caracteres Unicode mais usados (aqueles do "plano multilíngue básico") possuem pontos de código que se encaixam em 16 bits e podem ser representados por um elemento de uma string. Caracteres unicode cujos pontos de código não se encaixam em 16 bits são codificados usando as regras de

UTF-16 como uma sequência (conhecida como "par substituta") de dois valores de 16 bits. Isso significa que uma sequência JavaScript de comprimento 2 (dois valores de 16 bits) pode representar apenas um único caractere unicode:

```
deixe euro = "€"; Let Love =
"“”; Euro.length // => 1: este personagem tem um elemento de 16 bits
amor.length // => 2: UTF-16 A codificação de ‘é “\udc99’;
```

A maioria dos métodos de manipulação de cordas definidos pelo JavaScript opera em valores de 16 bits, não caracteres. Eles não tratam pares substitutos, especialmente, não realizam normalização da corda e nem sequer garantem que uma corda seja UTF-16 bem formada.

No ES6, no entanto, as strings são iteráveis e, se você usar o operador for/de loop ou ... com uma string, ele itera os caracteres reais da string, não os valores de 16 bits.

3.3.1 Literais de cordas

Para incluir uma string em um programa JavaScript, basta envolver os caracteres da string dentro de um par correspondente de citações ou backticks simples ou duplos (''ou "ou `'). Caracteres e backticks de dupla cita podem estar contidos em strings delimitados por uma única- Personagens de citação e, da mesma forma, para strings delimitadas por citações duplas e backtics.

```
"" // A sequência vazia: tem
'teste'; com zero caracteres
`3.14`; nome = "myform";
"Você não prefere o livro de 0'Reilly?";
`τ é a razão entre a circunferência de um círculo e
seu raio`;
`Ela disse 'oi', ,
disse ele.
```

Strings delimitadas com backticks são uma característica do ES6 e permitem que as expressões JavaScript sejam incorporadas (ou interpoladas em) a string literal. Esta sintaxe de interpolação de expressão é coberta em §3.3.4.

As versões originais do JavaScript exigiam que os literais fossem escritos

em uma única linha, e é comum ver código JavaScript que cria strings longas concatenando strings de linha única com o operador `+`. A partir do ES5, entretanto, você pode quebrar uma string literal em múltiplas linhas terminando cada linha, exceto a última, com uma barra invertida (`\`). Nem a barra invertida nem o terminador de linha que a segue fazem parte da string literal. Se você precisar incluir um caractere de nova linha em uma string literal entre aspas simples ou duplas, use a sequência de caracteres `\n` (documentada na próxima seção). A sintaxe de backtick ES6 permite que strings sejam quebradas em múltiplas linhas e, neste caso, os terminadores de linha fazem parte da string literal:

```
// Uma string representando 2 linhas escritas em uma
// linha: 'two\nlines';

// Uma string de uma linha escrita em 3
// linhas: "one\
// longa\
// linha";

// Uma string de duas linhas escrita em duas linhas: `o caractere de
// nova linha no final desta linha está incluído literalmente nesta string`
```

Observe que ao usar aspas simples para delimitar suas strings, você deve ter cuidado com as contrações e os possessivos do inglês, como `can't` e `O'Reilly's`. Como o apóstrofo é igual ao caractere de aspas simples, você deve usar o caractere de barra invertida (`\`) para “escapar” de quaisquer apóstrofos que aparecem em strings entre aspas simples (os escapes são explicados na próxima seção).

Na programação JavaScript do lado do cliente, o código JavaScript pode conter sequências de código HTML e o código HTML pode conter sequências de código JavaScript. Assim como o JavaScript, o HTML usa caracteres simples ou duplos

Citações para delimitar suas cordas. Assim, ao combinar JavaScript e HTML, é uma boa idéia usar um estilo de citações para JavaScript e o outro estilo para HTML. No exemplo a seguir, a sequência "Thank You" é citada única em uma expressão de JavaScript, que é então citada em um atributo HTML Event Handler:

```
<button onclick="alert('Thank you')">Clique em mim</button>
```

3.3.2 Sequências de fuga em literais de cordas

O caractere de barragem () tem um propósito especial em strings de javascript. Combinado com o personagem que o segue, ele representa um personagem que não é representável na sequência. Por exemplo, \n é uma sequência de fuga que representa um caractere de nova linha.

Outro exemplo, mencionado anteriormente, é o \''Escape, que representa o caractere de citação única (ou apóstrofo). Essa sequência de fuga é útil quando você precisa incluir um apóstrofo em um literal de string que esteja contido em cotações únicas. Você pode ver por que elas são chamadas de sequências de escape: a barra de barriga permite que você escape da interpretação usual do caractere único. Em vez de usá -lo para marcar o final da string, você a usa como um apóstrofo:

```
'Você está certo, não pode ser uma citação';
```

A Tabela 3-1 lista as sequências de escape JavaScript e os caracteres que eles representam. Três sequências de fuga são genéricas e podem ser usadas para representar qualquer caractere especificando seu código de caracteres Unicode como um número hexadecimal. Por exemplo, a sequência \xa9 representa o símbolo de direitos autorais, que tem o unicode codificado dado pelo

número hexadecimal A9. Da mesma forma, o escape \u representa um caractere Unicode arbitrário especificado por quatro dígitos hexadecimais ou de um a cinco dígitos quando os dígitos estão entre chaves: \u03c0 representa o caractere π, por exemplo, e \u{1f600} representa o “sorridente cara” emoji.

Tabela 3-1. Sequências de escape JavaScript

Sigas-nos	Personagem representado
\0	O caractere NUL (\u0000)
\b	Retrocesso (\u0008)
\t	Guia horizontal (\u0009)
\n	Nova linha (\u000A)
\v	Guia vertical (\u000B)
\f	Feed de formulário (\u000C)
\r	Retorno de transporte (\u000D)
\"	Aspas duplas (\u0022)
\'	Apóstrofo ou aspas simples (\u0027)
\`	Barra invertida (\u005C)
\xnn	O caractere Unicode especificado pelos dois dígitos hexadecimais nn
\unn nn	O caractere Unicode especificado pelos quatro dígitos hexadecimais nnnn
\u{n} }	O caractere Unicode especificado pelo ponto de código n, onde n é de um a seis dígitos hexadecimais entre 0 e 10FFFF (ES6)

Se o caractere \ preceder qualquer caractere diferente dos mostrados na Tabela 3-1, a barra invertida será simplesmente ignorada (embora versões futuras da linguagem possam, é claro, definir novas sequências de escape). Por exemplo, \# é igual a #. Finalmente, como observado anteriormente, o ES5 permite uma barra invertida antes de uma quebra de linha para quebrar uma string literal em várias linhas.

3.3.3 Trabalhando com Strings

Um dos recursos integrados do JavaScript é a capacidade de concatenar strings. Se você usar o operador + com números, ele os somará. Mas se você usar este operador em strings, ele as une anexando o segundo ao primeiro. Por exemplo:

```
deixe msg = "Olá, " + "mundo"; //Produz a string  
"Olá, mundo" let saudação = "Bem-vindo ao meu blog,"  
+ " " + nome;
```

As strings podem ser comparadas com os operadores padrão === igualdade e !== desigualdade: duas strings são iguais se e somente se consistirem exatamente na mesma sequência de valores de 16 bits. Strings também podem ser comparadas com os operadores <, <=, > e >=. A comparação de strings é feita simplesmente comparando os valores de 16 bits. (Para comparação e classificação de strings com reconhecimento de localidade mais robustas, consulte §11.7.3.)

Para determinar o comprimento de uma string – o número de valores de 16 bits que ela contém – use a propriedade length da string:

```
comprimento
```

Além desta propriedade de comprimento, o JavaScript fornece uma API rica para trabalhar com strings:

```
Seja s = "Olá, mundo"; // Comece com algum texto.

// obtenção de partes de uma corda S.Substring (1,4)
// => "ell": o 2º, 3º e 4º caracteres.
s.slice (1,4) // => "ell": a mesma coisa

S.Slice (-3)           // => "rld": últimos 3 caracteres
s.split ("")           // => ["Hello", "World"]:
String delimiter       dividido em

// pesquisando uma string
s.IndexOf ("L") // => 2: Posição da primeira letra L
s.IndexOf ("L", 3) // => 3: posição de primeiro "L" em ou
Depois de 3 s.IndexOf
("Z") // => -1: s não inclui o
Substring "Z";
S.LastIndexOf ("L") // => 10: Posição da última letra L

// funções de pesquisa booleana no ES6 e mais tarde s.startsWith
("inferno") // => true: a string começa com estes

s.endsWith (!) // => false: s não termina com isso
s.includes ("ou") // => true: s inclui substring "ou"

// Criando versões modificadas de uma string s.replace
("lló", "ya") // => "heya",
"mundo" s.toLowerCase () // => "hello",
"mundo" s.ToUpper () // => "Olá, mundo"
s.Normalize () // Unicode NFC Normalização: ES6 S.Normalize
("NFD") // Normalização NFD. Também
"nfkc", "nfkd";

// Inspecionando caracteres individuais (16 bits) de uma string
s.charAt (0) // => "h": o primeiro caractere
s.charAt (s.length-1) // => "d": o último caractere
S.charCodeAt (0) // => 72: número de 16 bits na posição
especificada
S.codePointAt (0) // => 72: ES6, trabalha para pontos de código
```

16 bits

```
// Funções de preenchimento de string no ES2017
" ".padStart (3) // => " x": adicione espaços à esquerda
a um comprimento de 3 "x";
.padEnd (3) // => "x"      : Adicione espaços à direita
a um comprimento de 3 "x";
.padStart (3, "*") // => ** x**: adicione estrelas à esquerda para
um comprimento de 3 "x";
.padEnd (3, "-") // => "x--": adicione traços à direita
para um comprimento de 3

// Funções de corte de espaço. Trim () é ES5; Outros ES2019
"Teste".Trim () // => "Teste": Remova os
espaços no início e no final
"Teste".TrimStart () // => teste: remova os espaços à esquerda.
Também TrimLeft
"teste".TrimEnd () // => teste: remova os espaços em
certo. Também TrimRight

// Métodos de string diversos s.concat ("!") // =>
"Olá, mundo!" : Apenas use + operador em vez disso em vez
disso
".".Repeat (5) // => "<> <> <> <> <>": concatenar n
cópias. ES6
```

Lembre -se de que as cordas são imutáveis em JavaScript.
Métodos como substituir () e touppercase () retornam novas strings: eles não modificam a string na qual são chamados.

As strings também podem ser tratadas como matrizes somente leitura e você pode acessar caracteres individuais (valores de 16 bits) de uma string usando colchetes quadrados em vez do método Charat ():

Seja s = "Olá, mundo"; s [0] // =>
" " ; s [s.length-1] // => "d";

3.3.4 Literais de modelo

No ES6 e posteriores, literais de string podem ser delimitados com crases:

```
vamos s = `olá mundo`;
```

No entanto, isso é mais do que apenas outra sintaxe de literal de string, porque esses literais de modelo podem incluir expressões JavaScript arbitrárias. O valor final de uma string literal em crases é calculado avaliando quaisquer expressões incluídas, convertendo os valores dessas expressões em strings e combinando essas strings computadas com os caracteres literais dentro dos crases:

```
deixe nome = "Conta";
deixe saudação = `Olá ${ nome }.`;      // saudação == "Olá
Conta.";
```

Tudo entre \${ e } correspondente é interpretado como uma expressão JavaScript. Tudo fora das chaves é texto literal de string normal. A expressão entre colchetes é avaliada e depois convertida em uma string e inserida no modelo, substituindo o cífrão, as chaves e tudo o que estiver entre eles.

Um literal de modelo pode incluir qualquer número de expressões. Ele pode usar qualquer um dos caracteres de escape que as strings normais podem usar e pode abranger qualquer número de linhas, sem necessidade de escape especial. O seguinte modelo literal inclui quatro expressões JavaScript, uma sequência de escape Unicode e pelo menos quatro novas linhas (os valores da expressão também podem incluir novas linhas):

```
deixe errorMessage = `\\ \u2718 Falha no teste em
${filename}:${linenumber}: ${exception.message}`
```

```
Rastreamento de  
pilha:  
${exception.stack} `;
```

A barra invertida no final da primeira linha aqui escapa da nova linha inicial para que a string resultante comece com o caractere Unicode **X** (\u2718) em vez de uma nova linha.

LITERÁIS DE MODELO TAGADOS

Um recurso poderoso, mas menos comumente usado, dos literais de modelo é que, se um nome de função (ou “tag”) vier logo antes do crase de abertura, o texto e os valores das expressões dentro do literal de modelo serão passados para a função. O valor deste “modelo literal marcado” é o valor de retorno da função. Isso poderia ser usado, por exemplo, para aplicar escape HTML ou SQL aos valores antes de substituí-los no texto.

ES6 tem uma função de tag integrada: `String.raw()`. Ele retorna o texto entre crases sem qualquer processamento de escapes de barra invertida:

```
`\n`.comprimento          // => 1: a string possui um único  
caractere de nova linha  
String.raw`\n`.length    // => 2: um caractere de barra invertida e o  
letra n
```

Observe que mesmo que a parte da tag de um literal de modelo marcado seja uma função, não há parênteses usados em sua invocação. Neste caso muito específico, os caracteres de crase substituem os parênteses de abertura e fechamento.

A capacidade de definir suas próprias funções de tag de modelo é uma ferramenta poderosa

recurso do JavaScript. Essas funções não precisam retornar strings e podem ser usadas como construtores, como se definissem uma nova sintaxe literal para a linguagem. Veremos um exemplo em §14.5.

3.3.5 Correspondência de padrões

JavaScript define um tipo de dados conhecido como expressão regular (ou RegExp) para descrever e combinar padrões em strings de texto. RegExps não são um dos tipos de dados fundamentais em JavaScript, mas têm uma sintaxe literal como os números e as strings, então às vezes parecem fundamentais. A gramática dos literais de expressões regulares é complexa e a API que eles definem não é trivial. Eles estão documentados em detalhes em §11.3. Entretanto, como os RegExps são poderosos e comumente usados para processamento de texto, esta seção fornece uma breve visão geral.

O texto entre um par de barras constitui uma expressão regular literal. A segunda barra do par também pode ser seguida por uma ou mais letras, que modificam o significado do padrão. Por exemplo:

```
/^HTML/;          //Corresponde as letras HTML no
início de uma string
/[1-9][0-9]*/;    //Corresponde a um dígito diferente de zero, seguido por
qualquer número de dígitos
/\bjavascript\b/i; //Corresponde a "javascript" como uma palavra, caso-
insensível
```

Os objetos RegExp definem vários métodos úteis, e as strings também possuem métodos que aceitam argumentos RegExp. Por exemplo:

```
deixe texto = "teste: 1, 2, 3"; //Exemplo de texto
deixe padrão = /\d+/g;           //Corresponde a todas as instâncias de
um ou mais dígitos
```

```
Pattern.test (texto)          // => true: existe uma correspondência
text.search (padrão)          // => 9: posição do primeiro
Combine o
texto.match (padrão)          // => ["1","2","3"]: matriz
de todas as correspondências
text.replace (padrão, "")      // => "Teste: #, #, #";
text.split (/^\d+/)           // => ["","1","2","3"]:
dividido em não "
```

3.4 valores booleanos

Um valor booleano representa a verdade ou a falsidade, dentro ou fora, sim ou não. Existem apenas dois valores possíveis desse tipo. As palavras reservadas verdadeiras e falsas avaliam para esses dois valores.

Os valores booleanos são geralmente o resultado de comparações que você faz em seus programas JavaScript. Por exemplo:

```
a === 4
```

Esse código testa para verificar se o valor da variável A é igual ao número 4. Se for, o resultado dessa comparação é o valor booleano true. Se A não for igual a 4, o resultado da comparação é falso.

Os valores booleanos são comumente usados nas estruturas de controle de JavaScript. Por exemplo, a instrução if/else no JavaScript executa uma ação se um valor booleano for verdadeiro e outra ação se o valor for falso. Você geralmente combina uma comparação que cria um valor booleano diretamente com uma instrução que a usa. O resultado é assim:

```
if (a === 4) {b = b + 1;
```

```
} outro {  
a = a + 1; }
```

Este código verifica se A é igual a 4. Se sim, adiciona 1 a B; Caso contrário, adiciona 1 a a.

Como discutiremos no §3.9, qualquer valor JavaScript pode ser convertido em um valor booleano. Os seguintes valores se convertem para e, portanto, funcionam como, falso:

```
nulo indefinido 0  
-0 nan  
" " // a  
corda vazia
```

Todos os outros valores, incluindo todos os objetos (e matrizes) se convertem e funcionam como, verdadeiros. Falso, e os seis valores que se convertem a ele, às vezes são chamados de valores falsamente, e todos os outros valores são chamados de verdade. Sempre que o JavaScript espera um valor booleano, um valor falsamente funciona como False e um valor verdadeiro funciona como verdadeiro.

Como exemplo, suponha que a variável O possua um objeto ou o valor nulo. Você pode testar explicitamente para ver se O não é nulo com uma instrução IF como esta:

```
if (o! == NULL) ...
```

O operador não equal! == Compara-se a NULL e avalia como verdadeiro ou falso. Mas você pode omitir a comparação e, em vez disso,

Confie no fato de NULL ser falsamente e objetos são verdadeiros:

```
se (o) ...
```

No primeiro caso, o corpo do IF será executado apenas se O não for nulo. O segundo caso é menos rigoroso: ele executará o corpo do se apenas se o não for falso ou qualquer valor falsamente (como nulo ou indefinido). Qual a declaração se for apropriada para o seu programa realmente depende de quais valores você espera ser atribuído a 0. Se você precisar distinguir nulo de 0 e " ", use uma comparação explícita.

Os valores booleanos possuem um método `ToString()` que você pode usar para convertê -los em strings "verdadeiros" ou "falsos", mas eles não têm outros métodos úteis. Apesar da API trivial, existem três importantes operadores booleanos.

O operador `&&` executa o booleano e a operação. Ele avalia um valor verdadeiro se e somente se ambos os seus operandos forem verdadeiros; Ele avalia um valor falsário de outra forma. O `||` O operador é o booleano ou operação: ele avalia um valor verdadeiro se um (ou ambos) de seus operandos for verdadeiro e avaliará um valor falsário se ambos os operando forem falsidade. Finalmente, o unário! O operador executa a operação booleana: ele avalia como verdadeiro se o seu operando for falsamente e avaliar como falso se o seu operando for verdade. Por exemplo:

```
if ((x === 0 && y === 0) ||! (z === 0)) {  
    // x e y são zero ou z é diferente de zero}
```

Detalhes completos sobre esses operadores estão em §4.10.

3.5 nulo e indefinido

null é uma palavra-chave de linguagem avaliada como um valor especial que geralmente é usado para indicar a ausência de um valor. Usar o operador `typeof` em `null` retorna a string “objeto”, indicando que `null` pode ser pensado como um valor de objeto especial que indica “nenhum objeto”. Na prática, entretanto, `null` é normalmente considerado o único membro de seu próprio tipo e pode ser usado para indicar “nenhum valor” para números e strings, bem como para objetos. A maioria das linguagens de programação tem um equivalente ao `null` do JavaScript: você pode estar familiarizado com ele como `NULL`, `nil` ou `None`.

JavaScript também possui um segundo valor que indica ausência de valor. O valor indefinido representa um tipo mais profundo de ausência. É o valor das variáveis que não foram inicializadas e o valor que você obtém ao consultar o valor de uma propriedade de objeto ou elemento de array que não existe. O valor indefinido também é o valor de retorno de funções que não retornam explicitamente um valor e o valor de parâmetros de função para os quais nenhum argumento é passado. `undefined` é uma constante global predefinida (não uma palavra-chave de linguagem como `null`, embora esta não seja uma distinção importante na prática) que é inicializada com o valor indefinido. Se você aplicar o operador `typeof` ao valor indefinido, ele retornará “indefinido”, indicando que este valor é o único membro de um tipo especial.

Apesar dessas diferenças, nulo e indefinido indicam um

ausência de valor e geralmente pode ser usada de forma intercambiável. O operador da igualdade == considera iguais. (Use o estrito operador de igualdade === para distinguir -los.) Ambos são valores falsamente: eles se comportam como falsos quando um valor booleano é necessário. Nem nulo nem indefinido têm propriedades ou métodos. De fato, usando. ou [] acessar uma propriedade ou método desses valores causa um TypeError.

Considero indefinido para representar uma ausência de valor e nulo no nível do sistema, inesperado ou semelhante a erros para representar uma ausência de valor em nível de programa, normal ou esperada. Evito usar nulo e indefinido quando puder, mas se eu precisar atribuir um desses valores a uma variável ou propriedade ou passar ou retornar um desses valores para ou para uma função, geralmente uso nulo. Alguns programadores se esforçam para evitar totalmente nulos e usam indefinidos em seu lugar onde quer que possam.

3.6 Símbolos

Os símbolos foram introduzidos no ES6 para servir como nomes de propriedades que não são de corda. Para entender os símbolos, você precisa saber que o JavaScript#39;s

O tipo de objeto fundamental é uma coleção não ordenada de propriedades, onde cada propriedade tem um nome e um valor. Os nomes das propriedades são normalmente (e até ES6, foram exclusivamente) strings. Mas no ES6 e mais tarde, os símbolos também podem servir a esse propósito:

```
Seja strname = "Nome da string"; // uma string para usar como um
nome da propriedade let symname = símbolo
("propname"); // Um símbolo a ser usado
como um nome de propriedade typeof strname //
=> "String": strname é uma string
TIP00F SIMNAME // => "Símbolo": Symname é
```

```

a symbol let o = {};
                    //Cria um novo objeto
o[strname] = 1;
                    // Define uma propriedade com um
nome da string o[symname] = 2;
                    // Define uma propriedade com um
Nome do símbolo
o[strname]
                    // => 1: acessa a string-
propriedade
nomeada o[symname]
                    // => 2: acesse o símbolo-
propriedade nomeada

```

O tipo Símbolo não possui uma sintaxe literal. Para obter um valor de símbolo, você chama a função Symbol(). Esta função nunca retorna o mesmo valor duas vezes, mesmo quando chamada com o mesmo argumento. Isso significa que se você chamar Symbol() para obter um valor de Símbolo, você pode usar esse valor com segurança como um nome de propriedade para adicionar uma nova propriedade a um objeto e não precisa se preocupar com a possibilidade de substituir uma propriedade existente pela mesma propriedade. nome. Da mesma forma, se você usar nomes de propriedades simbólicas e não compartilhar esses símbolos, poderá ter certeza de que outros módulos de código em seu programa não substituirão acidentalmente suas propriedades.

Na prática, os Símbolos servem como um mecanismo de extensão da linguagem. Quando o ES6 introduziu o loop for/of (§5.4.4) e os objetos iteráveis (Capítulo 12), foi necessário definir um método padrão que as classes poderiam implementar para se tornarem iteráveis. Mas a padronização de qualquer nome de string específico para esse método iterador quebraria o código existente, portanto, um nome simbólico foi usado em seu lugar. Como veremos no Capítulo 12, Symbol.iterator é um valor de Símbolo que pode ser usado como um nome de método para tornar um objeto iterável.

A função Symbol() recebe um argumento de string opcional e retorna

um valor único de símbolo. Se você fornecer um argumento de string, essa string será incluída na saída do método `ToString()` do símbolo. Observe, no entanto, que o símbolo de chamada () duas vezes com a mesma string produz dois valores de símbolos completamente diferentes.

```
Seja s = símbolo ("sym_x"); s.ToString () // =>  
"Symbol(sym_x)"
```

`ToString()` é o único método interessante de instâncias de símbolos. No entanto, existem outras duas funções relacionadas ao símbolo que você deve conhecer. Às vezes, ao usar símbolos, você deseja mantê-los privados em seu próprio código para garantir que suas propriedades nunca entrem em conflito com as propriedades usadas por outro código. Outras vezes, no entanto, você pode definir um valor de símbolo e compartilhá-lo amplamente com outro código. Esse seria o caso, por exemplo, se você estivesse definindo algum tipo de extensão em que queria que outro código fosse capaz de participar, como no mecanismo de `símbolo.iterator` descrito anteriormente.

Para servir a este último caso de uso, o JavaScript define um registro de símbolos globais. A função `Symbol.for()` pega um argumento de string e retorna um valor de símbolo associado à string que você passa. Se nenhum símbolo já estiver associado a essa string, um novo será criado e devolvido; Caso contrário, o símbolo já existente é retornado. Isto é, a função `symbol.for()` é completamente diferente da função `símbolo()`: `símbolo()` nunca retorna o mesmo valor duas vezes, mas `símbolo.for()` sempre retorna o mesmo valor quando chamado com a mesma string. A string passada para `symbol.for()` aparece na saída de `toString()` para o símbolo retornado, e também pode ser

Recuperado chamando `Symbol.keyFor()` no símbolo retornado.

```
Seja s = símbolo.for ("compartilhado");
Seja t = símbolo.for ("compartilhado");
s === t // > true
s.toString () // >
"compartilhado"
símbolo (compartilhado) "compartilhado"
(t) // > "compartilhado";
```

3.7 O objeto global

As seções anteriores explicaram os tipos e valores primitivos de JavaScript. Tipos de objetos - objetos, matrizes e funções - são cobertos de capítulos próprios mais tarde neste livro. Mas há um valor de objeto muito importante que devemos cobrir agora. O objeto global é um objeto JavaScript regular que serve a um propósito muito importante: as propriedades desse objeto são os identificadores definidos globalmente que estão disponíveis para um programa JavaScript. Quando o intérprete JavaScript inicia (ou sempre que um navegador da Web carrega uma nova página), ele cria um novo objeto global e fornece um conjunto inicial de propriedades que definem:

- Constantes globais como indefinidas, infinito e nan
- Funções globais como `isNaN()`, `parseInt()` ([§3.9.2.2](#)) e `eval()` ([§4.12](#))
- Funções do construtor como `Date()`, `RegExp()`, `String()`, `Object()` e `Array()` ([§3.9.2.2](#))
- Objetos globais como `Math` e `Json` ([§6.8](#))

As propriedades iniciais do objeto global não são palavras reservadas, mas elas merecem ser tratadas como se fossem. Este capítulo já descreveu algumas dessas propriedades globais. A maioria dos outros será

abordados em outras partes deste livro.

No Node, o objeto global possui uma propriedade chamada `global` cujo valor é o próprio objeto global, portanto você sempre pode se referir ao objeto global pelo nome `global` nos programas Node.

Em navegadores da Web, o objeto `Window` serve como objeto global para todo o código JavaScript contido na janela do navegador que ele representa. Este objeto `Window` global possui uma propriedade de janela auto-referencial que pode ser usada para fazer referência ao objeto global. O objeto `Window` define as principais propriedades globais, mas também define algumas outras propriedades globais que são específicas para navegadores da Web e JavaScript do lado do cliente. Os threads de trabalho da Web (§15.13) têm um objeto global diferente da janela à qual estão associados. O código em um trabalhador pode referir-se ao seu objeto global como `self`.

ES2020 finalmente define `globalThis` como a forma padrão de se referir ao objeto global em qualquer contexto. No início de 2020, esse recurso foi implementado por todos os navegadores modernos e pelo Node.

3.8 Valores primitivos imutáveis e referências de objetos mutáveis

Há uma diferença fundamental em JavaScript entre valores primitivos (indefinidos, nulos, booleanos, números e strings) e objetos (incluindo arrays e funções). Primitivos são imutáveis: não há como alterar (ou “mutar”) um valor primitivo. Isso é óbvio para números e booleanos – nem faz sentido

alterar o valor de um número. Entretanto, não é tão óbvio para strings. Como as strings são como matrizes de caracteres, você pode esperar alterar o caractere em qualquer índice especificado. Na verdade, o JavaScript não permite isso, e todos os métodos de string que parecem retornar uma string modificada estão, na verdade, retornando um novo valor de string. Por exemplo:

```
vamos = "olá"; //Comece com algum texto em minúsculas
s.toUpperCase(); // Retorna "OLÁ", mas não altera s
é // => "hello": a string original não
mudado
```

Os primitivos também são comparados por valor: dois valores são iguais apenas se tiverem o mesmo valor. Isso parece circular para números, booleanos, nulos e indefinidos: não há outra maneira de compará-los. Novamente, porém, isso não é tão óbvio para strings. Se dois valores de string distintos forem comparados, o JavaScript os tratará como iguais se, e somente se, eles tiverem o mesmo comprimento e se o caractere em cada índice for o mesmo.

Os objetos são diferentes dos primitivos. Primeiro, eles são mutáveis – seus valores podem mudar:

```
seja o = { x: 1 }; //Começa com um objeto
o.x = 2; // Mute-o alterando o valor de a
propriedade
oy = 3; // Mute-o novamente adicionando um novo
propriedade

seja a = [1,2,3]; //Arrays também são mutáveis
uma[0] = 0; //Altera o valor de um elemento do array
uma[3] = 4; //Adiciona um novo elemento do array
```

Os objetos não são comparados por valor: dois objetos distintos não são iguais

Mesmo que eles tenham as mesmas propriedades e valores. E duas matrizes distintas não são iguais, mesmo que tenham os mesmos elementos na mesma ordem:

```
Seja o = {x: 1}, p = {x: 1};      // dois objetos com o mesmo
propriedade
s o === p                         // >; false: objetos distintos
nunca são iguais,
vamos a = [], b = [];              // duas matrizes distintas e vazias
a === b                           // >; false: matrizes distintas são
nunca igual
```

Às vezes, os objetos são chamados de tipos de referência para distingui-los dos tipos primitivos de JavaScript. Usando essa terminologia, os valores dos objetos são referências e dizemos que os objetos são comparados por referência: dois valores de objeto são os mesmos se e somente se eles se referirem ao mesmo objeto subjacente.

```
deixe A = [];          // A variável A se refere a uma matriz vazia.
Seja b = a;            // Agora B refere-se à mesma matriz.
b [0] = 1;             // MATA A matriz referida pela variável b.
A [0]                  // >; 1: A mudança também é visível
variável a.
a === b                // >; true: a e b referem-se ao mesmo objeto,
Então eles são iguais.
```

Como você pode ver neste código, atribuindo um objeto (ou matriz) a uma variável simplesmente atribui a referência: ela não cria uma nova cópia do objeto. Se você deseja fazer uma nova cópia de um objeto ou matriz, copie explicitamente as propriedades do objeto ou os elementos da matriz. Este exemplo demonstra o uso de um loop for (§5.4.3):

```
deixe A = ["A", "B", "C"]; // Uma matriz que queremos
cópia, deixe
b = [];                  // Uma matriz distinta nós vamos
```

```

copie para for(let i = 0; i < a.length; i++) { // Para cada índice de
a[]
    b[i] = uma[i];                                // Copia um elemento de um
} em b } deixe c =
Array.from(b);                                    // No ES6, copia arrays
com Array.from()

```

Da mesma forma, se quisermos comparar dois objetos ou arrays distintos, devemos comparar suas propriedades ou elementos. Este código define uma função para comparar dois arrays:

```

função equalArrays(a, b) { if
(a === b) retornar verdadeiro;                      // Idêntico
matrizes são iguais
if (a.length! == b.length) retornar falso; // Arrays de tamanhos
diferentes não são iguais
    for(seja i = 0; i < a.length; i++) {           // Loop através
todos os elementos
        if (a[i] !== b[i]) retornar falso;          // Se houver
diferem, matrizes não são iguais
    } retornar
    verdadeiro; // De outra forma
    eles são iguais }

```

3.9 Conversões de tipo

JavaScript é muito flexível quanto aos tipos de valores que requer. Vimos isso para booleanos: quando o JavaScript espera um valor booleano, você pode fornecer um valor de qualquer tipo e o JavaScript o converterá conforme necessário. Alguns valores (valores “verdadeiros”) são convertidos em verdadeiros e outros (valores “falsos”) são convertidos em falsos. O mesmo se aplica a outros tipos: se o JavaScript quiser uma string, ele converterá qualquer valor que você atribuir a uma string. Se o JavaScript quiser um número, ele tentará converter o valor que você

atribua-o a um número (ou a NaN se não puder realizar uma conversão significativa).

Alguns exemplos:

```
10 + "objetos" // => "10"           Número 10 converte
para uma string "7" "objetos";:  
* "4" // => 28: ambas as strings são convertidas em números
seja n = 1 - "x"; //n == NaN; string "x" não pode ser convertida para
um número n +
"objetos" // => "Objetos NaN": NaN converte para
string "NaN";
```

A Tabela 3-2 resume como os valores são convertidos de um tipo para outro em JavaScript. As entradas em negrito na tabela destacam conversões que você pode achar surpreendentes. Células vazias indicam que nenhuma conversão é necessária e nenhuma é executada.

Tabela 3-2. Conversões de tipo JavaScript

Valor para string	para número	para booleano
indefinido "undefined"	"undefined" NaN	falso
nulo "null"	0	falso
verdadeiro "true"	1	
falso "false"	0	
0 "0"	0	falso
"1,2"	(não vazio, 1.2)	verdadeiro
"um"	(não vazio, não NaN)	verdadeiro
0 "0"		falso

- 0 "0"	falso		
1 (finito, diferente de zero)	"1"	verdadeiro	
Infinidade		"Infinida	verdadeiro
- Infinidade		"-	verdadeiro
Em "No"		falso	
O (qualquer objeto)	Veja §3.9.3	Veja §3.9.3	verdadeiro
[] (matriz vazia) ""	0		verdadeiro
[9] (um elemento numérico)	"9" 9		verdadeiro
['a'] (qualquer outra matriz)	Use o método junção ()	Em	verdadeiro
function () {} (qualquer função)	Veja §3.9.3	Em	verdadeiro

As conversões primitivas para primitivas mostradas na tabela são relativamente diretas. A conversão para booleana já foi discutida no §3.4. A conversão em strings é bem definida para todos os valores primitivos.

A conversão em números é apenas um pouco mais complicada. Strings que podem ser analisadas à medida que os números se convertem a esses números. Os espaços de liderança e à direita são permitidos, mas quaisquer caracteres não espaciais que não façam parte que não fazem parte de uma causa literal numérica, causa a conversão de cordas em número para produzir NAN. Algumas conversões numéricas podem parecer surpreendentes: os verdadeiros convertidos em 1 e false e a corda vazia se convertem para 0.

A conversão de objeto para princípios é um pouco mais complicada e é objeto de §3.9.3.

3.9.1 Conversões e igualdade

JavaScript possui dois operadores que testam se dois valores são iguais. O “operador de igualdade estrita”, ===, não considera seus operandos iguais se não forem do mesmo tipo, e este é quase sempre o operador correto a ser usado durante a codificação. Mas como o JavaScript é tão flexível com conversões de tipo, ele também define o operador == com uma definição flexível de igualdade. Todas as comparações a seguir são verdadeiras, por exemplo:

```
null == indefinido // =&gt; verdadeiro: esses dois valores são  
tratados como iguais.  
"0" == 0           // =&gt; true: String é convertida em um número  
antes de comparar.  
0 == falso         // =&gt; true: Booleano converte em número  
antes de comparar.  
&quot;0&quot; == falso // =&gt; true: ambos os operandos são convertidos para 0  
antes de comparar!
```

§4.9.1 explica exatamente quais conversões são realizadas pelo operador == para determinar se dois valores devem ser considerados iguais.

Tenha em mente que a conversibilidade de um valor para outro não implica igualdade desses dois valores. Se indefinido for usado onde um valor booleano é esperado, por exemplo, ele será convertido em falso. Mas isso não significa que indefinido == falso. Operadores e instruções JavaScript esperam valores de vários tipos e realizam conversões para esses tipos. A instrução if converte indefinido em falso, mas o operador == nunca tenta converter seus operandos em booleanos.

3.9.2 Conversões Explícitas

Embora o JavaScript execute muitas conversões de tipo automaticamente,

às vezes você pode precisar realizar uma conversão explícita ou pode preferir tornar as conversões explícitas para manter seu código mais claro.

A maneira mais simples de realizar uma conversão de tipo explícita é usar as funções Boolean(), Number() e String():

```
Número("3") // => 3  
String(false) // =>  
// &quot;false&quot;;  
Boolean([]) // => verdadeiro  
Ou use false.toString()
```

Qualquer valor diferente de nulo ou indefinido possui um método `toString()`, e o resultado desse método geralmente é o mesmo retornado pela função `String()`.

Além disso, observe que as funções `Boolean()`, `Number()` e `String()` também podem ser invocadas – com `new` – como construtor. Se você usá-los dessa maneira, obterá um objeto “wrapper” que se comporta exatamente como um booleano primitivo, um número ou um valor de string. Esses objetos wrapper são um resquício histórico dos primeiros dias do JavaScript e nunca há um bom motivo para usá-los.

Certos operadores JavaScript realizam conversões implícitas de tipo e às vezes são usados explicitamente para fins de conversão de tipo. Se um operando do operador `+` for uma string, ele converte o outro em uma string. O operador unário `+` converte seu operando em um número. E o unário `!` operador converte seu operando em um booleano e o nega. Esses fatos levam aos seguintes idiomas de conversão de tipo que você pode ver em alguns códigos:

```
x + // => String(x)  
// &quot;&quot;
```

```
+x      // => Número(x)
x-0    // => Número(x)
!x     // => Boolean(x): Nota dupla !
```

Formatar e analisar números são tarefas comuns em programas de computador, e JavaScript possui funções e métodos especializados que fornecem controle mais preciso sobre conversões de número para string e de string para número.

O método `toString()` definido pela classe `Number` aceita um argumento opcional que especifica uma base, ou base, para a conversão. Caso não especifique o argumento, a conversão é feita na base 10. Porém, você também pode converter números em outras bases (entre 2 e 36). Por exemplo:

```
seja n = 17; deixe binário =
"0b" + n.toString(2);      // binário == "0b10001";
deixe octal = "0o" + n.toString(8); // octal == "0o21";
deixe hex = "0x" +           // hexadecimal == "0x11";
n.toString(16);
```

Ao trabalhar com dados financeiros ou científicos, você pode querer converter números em strings de forma que lhe dê controle sobre o número de casas decimais ou o número de dígitos significativos na saída, ou você pode querer controlar se a notação exponencial é usada. A classe `Number` define três métodos para esses tipos de conversões de número em string. `toFixed()` converte um número em uma string com um número especificado de dígitos após o ponto decimal. Nunca usa notação exponencial. `toExponential()` converte um número em uma string usando notação exponencial, com um dígito antes do ponto decimal e um número especificado de dígitos após o ponto decimal (o que significa que o número de dígitos significativos é um maior que o valor

você especifica). `toprecision ()` converte um número em uma string com o número de dígitos significativos que você especificar. Ele usa notação exponencial se o número de dígitos significativos não for grande o suficiente para exibir toda a parte inteira do número. Observe que todos os três métodos contornam os dígitos ou a almoofada à direita com zeros, conforme apropriado. Considere os seguintes exemplos:

```
Seja n = 123456.789; n.toFixed(0) // =>  
"123457"; n.toFixed(2) // =>  
"123456.79"; n.toFixed(5) // =>  
"123456.78900"; n.toExponential(1) //  
=> "1.2e+5"; n.toExponential(3)  
// => "1.235e+5"; n.toPrecision(4)  
// => "1.235e+5"; n.toPrecision(7)  
// => "123456.8"; n.toPrecision  
(10) // => "123456.7890";
```

Além dos métodos de formação de números mostrados aqui, a classe `Intl.NumberFormat` define um método de formatação de número mais geral e internacionalizada. Veja §11.7.1 para obter detalhes.

Se você passar uma string para a função de conversão `number ()`, ela tenta analisar essa string como um número inteiro ou literal de ponto flutuante. Essa função funciona apenas para números inteiros da Base-10 e não permite caracteres à direita que não fazem parte do literal. O `parseint ()` e

funções `parseFloat ()` (essas são funções globais, não métodos de qualquer classe) são mais flexíveis. `Parseint ()` Paresi apenas números inteiros, enquanto `parseFloat ()` analisa números inteiros e números de ponto flutuante. Se uma string começar com `"0x"` ou `"0X"`, o `parseint ()` a interpreta como um número hexadecimal. `Parseint ()` e `parseFloat ()` pula o espaço em branco, analisam o maior número possível de personagens numéricos, e

ignore tudo o que se segue. Se o primeiro caractere não espacial não fizer parte de um literal numérico válido, eles retornarão NaN:

```
parseInt("3 ratos cegos") // => 3
presseFloat("3,14 metros") // => 3.14
parseInt("-12.34") // => -12
parseInt("0xFF") // => 255
parseInt("0xff") // => 255
parseInt("-0xFF") // => -255
pressioneFloat(".1") // => 0.1
parseInt("0.1") // => 0
analisarInt(".1") // => NaN: inteiros não podem iniciar
com "."
pressioneFloat("$72,47") // => NaN: os números não podem começar
com "$";
```

parseInt() aceita um segundo argumento opcional especificando a raiz (base) do número a ser analisado. Os valores legais estão entre 2 e 36. Por exemplo:

```
parseInt("11", 2) // => 3: (1*2 + 1)
parseInt("ff", 16) // => 255: (15*16 + 15)
parseInt("zz", 36) // => 1295: (35*36 + 35)
parseInt("077", 8) // => 63: (7*8 + 7)
parseInt("077", 10) // => 77: (7*10 + 7)
```

3.9.3 Objeção às conversões primitivas

As seções anteriores explicaram como você pode converter explicitamente valores de um tipo para outro tipo e explicaram as conversões implícitas de valores do JavaScript de um tipo primitivo para outro tipo primitivo. Esta seção cobre as regras complicadas que o JavaScript usa para converter objetos em valores primitivos. É longo e obscuro, e se esta é sua primeira leitura deste capítulo, você deve se sentir

livre para pular a frente para o §3.10.

Uma razão para a complexidade das conversões de objeto para primitivas do JavaScript é que alguns tipos de objetos têm mais de uma representação primitiva. Os objetos de data, por exemplo, podem ser representados como strings ou como timestamps numéricos. A especificação JavaScript define três algoritmos fundamentais para converter objetos em valores primitivos:

preferência string

Esse algoritmo retorna um valor primitivo, preferindo um valor de string, se uma conversão para string for possível.

número preferido

Esse algoritmo retorna um valor primitivo, preferindo um número, se essa conversão for possível.

sem preferência

Esse algoritmo não expressa preferência sobre que tipo de valor primitivo é desejado, e as classes podem definir suas próprias conversões. Dos tipos de JavaScript embutidos, todos, exceto a data, implementam esse algoritmo como número preferido. A classe de data implementa esse algoritmo como preferência.

A implementação desses algoritmos de conversão de objeto para princípios é explicada no final desta seção. Primeiro, no entanto, explicamos como os algoritmos são usados no JavaScript.

Conversões objeto para boolean

As conversões de objeto para boolean são triviais: todos os objetos se convertem para true. Observe que esta conversão não requer o uso do objeto para-

Algoritmos primitivos descritos, e que se aplica literalmente a todos os objetos, incluindo matrizes vazias e até o objeto Wrapper New Boolean (falso).

Conversões de objeto para corda

Quando um objeto precisa ser convertido em uma string, o JavaScript primeiro o converte em um primitivo usando o algoritmo preferido e depois converte o valor primitivo resultante em uma string, se necessário, seguindo as regras na Tabela 3-2.

Esse tipo de conversão acontece, por exemplo, se você passar um objeto para uma função interna que espera um argumento de string, se você ligar

`String ()` como uma função de conversão e quando você interpola objetos em literais de modelo (§3.3.4).

Conversões de objeto para número

Quando um objeto precisa ser convertido em um número, o JavaScript primeiro o converte em um valor primitivo usando o algoritmo do número preferido e depois converte o valor primitivo resultante em um número, se necessário, seguindo as regras na Tabela 3-2.

Funções e métodos de JavaScript integrados que esperam argumentos numéricos convertem argumentos de objetos em números dessa maneira e a maioria (veja as exceções a seguir) os operadores JavaScript que esperam que operando numéricos convertem objetos em números dessa maneira também.

Conversões especiais de operadoras de casos

Os operadores são abordados em detalhes no capítulo 4. Aqui, explicamos o

Operadores de casos especiais que não usam as conversões básicas de objeto a cordas e objeto a número descritos anteriormente.

O operador + no JavaScript executa adição numérica e concatenação de string. Se um de seus operandos for um objeto, o JavaScript os converte em valores primitivos usando o algoritmo de não preferência. Depois de ter dois valores primitivos, verifica seus tipos. Se qualquer um dos argumentos for uma string, ela converte a outra em uma string e concatena as seqüências. Caso contrário, ele converte os dois argumentos em números e os adiciona.

O == e! = Os operadores realizam testes de igualdade e desigualdade de uma maneira solta que permite conversões de tipo. Se um operando é um objeto e o outro é um valor primitivo, esses operadores convertem o objeto em primitivo usando o algoritmo de não preferência e comparam os dois valores primitivos.

Finalmente, os operadores relacionais <, <=,> e >= comparam a ordem de seus operandos e podem ser usados para comparar números e strings. Se um operando for um objeto, ele será convertido em um valor primitivo usando o algoritmo de número preferido. Observe, no entanto, que, diferentemente da conversão de objeto-número, os valores primitivos retornados pela conversão do número preferido não são então convertidos em números.

Observe que a representação numérica dos objetos de data é significativamente comparável a <e>; mas a representação da string não é. Para objetos de data, o algoritmo de não preferência se converte em uma string; portanto, o fato de o JavaScript usar o algoritmo de número preferido para esses operadores significa que podemos usá-los para comparar a ordem de dois objetos de data.

Os métodos `ToString ()` e `ValueOf ()`

Todos os objetos herdam dois métodos de conversão usados por conversões de objeto para primitivas e, antes que possamos explicar os algoritmos de conversão de string, número preferido e sem preferência, precisamos explicar esses dois métodos.

O primeiro método é o `ToString ()`, e seu trabalho é retornar uma representação de string do objeto. O método `ToString ()` padrão não retorna um valor muito interessante (embora o achemos útil no §14.4.3):

```
{ {x: 1, y: 2}}. ToString () // => "object [objeto objeto]"
```

Muitas classes definem versões mais específicas do método `ToString ()`. O método `toString ()` da classe de matriz, por exemplo, converte cada elemento da matriz em uma string e une as seqüências resultantes em conjunto com vírgulas no meio. O método `toString ()` da classe de função converte funções definidas pelo usuário em strings do código-fonte JavaScript. A classe de data define um método `toString ()` que retorna uma string de data e hora legível por humanos (e JavaScript). A classe `Regexp` define um método `toString ()` que converte objetos `regexp` em uma string que se parece com um literal `regexp`:

```
[1,2,3] .ToString () // => "1,2,3"  
(função (x) {f (x);}). ToString () // => "function (x) {"  
f (x); }  
&quot;/\d+/g.toString () // => "&quot;/\\ d+/g&quot;  
Seja d = nova data (2020,0,1); D.ToString () // => "Qua  
Jan 01 2020 00:00:00 GMT-0800 (Hora padrão do Pacífico)"
```

A outra função de conversão de objetos é chamada de valueof (). O trabalho deste método é menos bem definido: ele deve converter um objeto em um valor primitivo que represente o objeto, se existir um valor primitivo. Objetos são valores compostos e a maioria dos objetos não pode ser realmente representada por um único valor primitivo; portanto, o método ValueOf () padrão simplesmente retorna o próprio objeto, em vez de retornar um

primitivo. Classes de invólucro, como string, número e booleano, definem os métodos de valueof () que simplesmente retornam o valor primitivo envolto. Matrizes, funções e expressões regulares simplesmente herdam o método padrão. Chamar ValueOf () Para instâncias desses tipos simplesmente retorna o próprio objeto. A classe de data define um método ValueOf () que retorna a data em sua representação interna: o número de milissegundos desde 1º de janeiro de 1970:

```
Seja D = New Date (2010, 0, 1);           // 1 de janeiro de 2010, (Pacífico
tempo)
d.valueOf ()                                // => 1262332800000
```

Algoritmos de conversão de objeto a princípios

Com os métodos ToString () e ValueOf (), agora podemos explicar aproximadamente como os três algoritmos de objeto para princípios funcionam (os detalhes completos são adiados até o §14.4.7):

- O algoritmo preferido primeiro tenta o método ToString (). Se o método for definido e retornar um valor primitivo, o JavaScript usará esse valor primitivo (mesmo que não seja uma string!). Se o ToString () não existir ou se ele retornar um objeto, o JavaScript tenta o método ValueOf (). Se esse método existir e retornar um valor primitivo, o JavaScript usará esse valor. Caso contrário, a conversão falha com um

Typingror.

- O algoritmo do número preferido funciona como o algoritmo preferido, exceto que ele tenta valueof () primeiro e tostring () segundo.
- O algoritmo sem preferência depende da classe do objeto que está sendo convertido. Se o objeto for um objeto de data, o JavaScript usará o algoritmo preferido. Para qualquer outro objeto, o JavaScript usa o algoritmo de número preferido.

As regras descritas aqui são verdadeiras para todos os tipos de javascript embutidos e são as regras padrão para todas as classes que você se define.

§14.4.7 Explica como você pode definir seus próprios algoritmos de conversão de objeto para princípios para as classes que você define.

Antes de deixarmos esse tópico, vale a pena notar que os detalhes da conversão do número preferido explicam por que as matrizes vazias se convertem para o número 0 e as matrizes de elementos únicos também podem se converter em números:

```
Número([])      // => 0: Isso é inesperado!
Número ([99])   // => 99: Sério?
```

A conversão de objeto em número primeiro converte o objeto em um primitivo usando o algoritmo de número preferido e depois converte o valor primitivo resultante em um número. O algoritmo do número preferido tenta valueof () primeiro e depois recai no tostring (). Mas a classe Array herda o método ValueOf () padrão, que não retorna um valor primitivo. Então, quando tentamos converter uma matriz em um número, acabamos invocando o método ToString () da matriz. Matrizes vazias convertem para a string vazia. E a string vazia se converte para o número 0. Uma matriz com um único elemento se converte para a mesma string

que esse elemento faz. Se uma matriz contiver um único número, esse número será convertido em uma string e depois novamente em um número.

3.10 Declaração e Atribuição de Variável

Uma das técnicas mais fundamentais de programação de computadores é o uso de nomes – ou identificadores – para representar valores.

Vincular um nome a um valor nos dá uma maneira de nos referirmos a esse valor e usá-lo nos programas que escrevemos. Quando fazemos isso, normalmente dizemos que estamos atribuindo um valor a uma variável. O termo “variável” implica que novos valores podem ser atribuídos: que o valor associado à variável pode variar à medida que nosso programa é executado. Se atribuirmos permanentemente um valor a um nome, chamaremos esse nome de constante em vez de variável.

Antes de poder usar uma variável ou constante em um programa JavaScript, você deve declará-la. No ES6 e posteriores, isso é feito com as palavras-chave `let` e `const`, que explicaremos a seguir. Antes do ES6, as variáveis eram declaradas com `var`, o que é mais idiossincrático e será explicado mais adiante nesta seção.

3.10.1 Declarações com `let` e `const`

No JavaScript moderno (ES6 e posterior), as variáveis são declaradas com a palavra-chave `let`, assim:

```
deixe eu;  
deixe somar;
```

Você também pode declarar múltiplas variáveis em uma única instrução `let`:

```
Deixe eu, soma;
```

É uma boa prática de programação atribuir um valor inicial às suas variáveis quando você as declarar, quando isso for possível:

```
Deixe Message = "Hello"; Seja i = 0, j = 0, k = 0; Seja x = 2, y = x*x; // Inicializadores podem usar variáveis declaradas anteriormente
```

Se você não especificar um valor inicial para uma variável com a instrução Let, a variável será declarada, mas seu valor será indefinido até que seu código atribua um valor a ele.

Para declarar uma constante em vez de uma variável, use const em vez de let. Const funciona como Let, exceto que você deve inicializar a constante quando a declarar:

```
const h0 = 74;           // Hubble Constant (km/s/mpc)
const C = 299792.458;   // velocidade de luz no vácuo (km/s)
Const i = 1.496e8;      // unidade astronômica: distância do
Sol (km)
```

Como o nome indica, as constantes não podem mudar seus valores, e qualquer tentativa de fazê-lo faz com que um TypeError seja jogado.

É uma convenção comum (mas não universal) declarar constantes usando nomes com todas as letras maiúsculas como H0 ou HTTP_NOT_FOUND como uma maneira de distingui-las das variáveis.

Quando usar const

Existem duas escolas de pensamento sobre o uso da palavra-chave const. Uma abordagem é usar const apenas para valores que são fundamentalmente imutáveis, como as constantes físicas mostradas, ou números de versão de programas, ou sequências de bytes usadas para identificar tipos de arquivos, por exemplo. Outra abordagem reconhece que muitas das chamadas variáveis em nosso programa na verdade nunca mudam enquanto nosso programa é executado. Nesta abordagem, declaramos tudo com const e, então, se descobrirmos que realmente queremos permitir que o valor varie, trocamos a declaração para let. Isso pode ajudar a evitar bugs, descartando alterações acidentais em variáveis que não pretendíamos.

Numa abordagem, usamos const apenas para valores que não devem mudar. No outro, usamos const para qualquer valor que não mude. Prefiro a abordagem anterior em meu próprio código.

No Capítulo 5, aprenderemos sobre as instruções de loop for, for/in e for/of em JavaScript. Cada um desses loops inclui uma variável de loop que recebe um novo valor atribuído a ele em cada iteração do loop. JavaScript nos permite declarar a variável de loop como parte da própria sintaxe do loop, e esta é outra maneira comum de usar let:

```
for(deixe i = 0, len = data.length; i < len; i++)
  console.log(data[i]);
for (deixe o dado dos dados) console.log (dado); for
(deixar propriedade no objeto) console.log (propriedade);
```

Pode parecer surpreendente, mas você também pode usar const para declarar as “variáveis” do loop para loops for/in e for/of, desde que o corpo do loop não reatribua um novo valor. Neste caso, a declaração const está apenas dizendo que o valor é constante durante uma iteração do loop:

```
for (dado const de dados) console.log (dado);
```

```
para (propriedade const em objeto) console.log (propriedade);
```

Escopo variável e constante

O escopo de uma variável é a região do seu código fonte de programa no qual é definido. Variáveis e constantes declaradas com LET e const são escopo de bloco. Isso significa que eles são definidos apenas dentro do bloco de código em que a instrução LET ou const aparece. A classe JavaScript e as definições de função são blocos, assim como os corpos de declarações IF/else, enquanto loops, loops e assim por diante. AGORADO falando, se uma variável ou constante for declarada dentro de um conjunto de aparelhos encaracolados, então os aparelhos encaracolados delimitam a região de código em que a variável ou constante é definida (embora é claro que não é legal referir uma variável ou constante de linhas de linhas do código que executa antes da instrução LET ou const que declara a variável). Variáveis e constantes declaradas como parte de um para, para/in ou para/de loop têm o corpo do loop como seu escopo, mesmo que tecnicamente apareçam fora do aparelho encaracolado.

Quando uma declaração aparece no nível superior, fora de qualquer bloco de código, dizemos que é uma variável ou constante global e possui escopo global. No nó e nos módulos JavaScript do lado do cliente (consulte o Capítulo 10), o escopo de uma variável global é o arquivo em que é definido. No javascript tradicional do lado do cliente, no entanto, o escopo de uma variável global é o documento HTML em que é definido. Isto é: se um<script> declares a global variable or constant, that variable or constant is defined in all of the <script> elements in that document (or at least all of the scripts that execute after the let or const statement executes).

DECLARAÇÕES REPETIDAS

É um erro de sintaxe usar o mesmo nome com mais de uma declaração let ou const no mesmo escopo. É legal (embora seja melhor evitar uma prática) declarar uma nova variável com o mesmo nome em um escopo aninhado:

```
const x = 1;           // Declara x como uma constante global
se (x === 1) {
  seja x = 2;         // Dentro de um bloco x pode se referir a um
  valor diferente
  console.log(x); // Imprime 2 }

console.log(x);       // Prints 1: estamos de volta ao global
escopo agora
seja x = 3;          // ERRO! Erro de sintaxe ao tentar re-
declarar x
```

DECLARAÇÕES E TIPOS

Se você está acostumado com linguagens de tipo estaticamente como C ou Java, você pode pensar que o objetivo principal das declarações de variáveis é especificar o tipo de valores que podem ser atribuídos a uma variável. Mas, como você viu, não existe nenhum tipo associado à variável do JavaScript

declarações.²Uma variável JavaScript pode conter um valor de qualquer tipo. Por exemplo, é perfeitamente legal (mas geralmente um estilo de programação pobre) em JavaScript atribuir um número a uma variável e depois atribuir uma string a essa variável:

```
seja i = 10; eu =
"dez";
```

3.10.2 Declarações de variáveis com var

Nas versões do JavaScript anteriores ao ES6, a única maneira de declarar uma variável é com a palavra-chave var, e não há como declarar constantes. A sintaxe de var é igual à sintaxe de let:

```
era x; var dados = [], contagem = dados.comprimento;
for(var i = 0; i < contagem; i++) console.log(dados[i]);
```

Embora var e let tenham a mesma sintaxe, existem diferenças importantes na forma como funcionam:

- Variáveis declaradas com var não possuem escopo de bloco. Em vez disso, eles têm como escopo o corpo da função que os contém, não importa quão profundamente aninhados estejam dentro dessa função.
- Se você usar var fora do corpo de uma função, ele declara uma variável global. Mas as variáveis globais declaradas com var diferem das globais declaradas com let de uma forma importante. Globais declarados com var são implementados como propriedades do objeto global (§3.7). O objeto global pode ser referenciado como `globalThis`. Então, se você escrever `var x = 2;` fora de uma função, é como se você escrevesse `globalThis.x = 2;`. Observe, entretanto, que a analogia não é perfeita: as propriedades criadas com declarações var globais não podem ser excluídas com o operador `delete` (§4.13.4). Variáveis e constantes globais declaradas com let e const não são propriedades do objeto global.
- Ao contrário das variáveis declaradas com let, é legal declarar a mesma variável múltiplas vezes com var. E como as variáveis var têm escopo de função em vez de escopo de bloco, é comum fazer esse tipo de redeclaração. A variável i é freqüentemente usada para valores inteiros e especialmente como variável de índice de loops for. Em uma função com múltiplos for

Loops, é típico para cada um começar (var i = 0; porque o Var não esconde essas variáveis para o corpo do loop, cada um desses loops é (inofensivamente) re-desconfiar e inicializar o mesmo variável.

- Uma das características mais incomuns das declarações VAR é conhecida como iça. Quando uma variável é declarada com VAR, a declaração é levantada (ou "holed") para o topo da função anexante. A inicialização da variável permanece onde você a escreveu, mas a definição da variável se move para o topo da função. Portanto, as variáveis declaradas com o VAR podem ser usadas, sem erro, em qualquer lugar da função de anexo. Se o código de inicialização ainda não tiver executado, o valor da variável pode ser indefinido, mas você não receberá um erro se usar a variável antes de ser inicializada. (Esta pode ser uma fonte de bugs e é uma das importantes equívocas que permitem corrigir: se você declarar uma variável com LET, mas tentar usá-la antes que a declaração let funcione, você receberá um erro real em vez de apenas ver um valor indefinido.)

Usando variáveis não declaradas

No modo rigoroso (§5.6.3), se você tentar usar uma variável não declarada, receberá um erro de referência ao executar seu código. Fora do modo rigoroso, no entanto, se você atribuir um valor a um nome que não foi declarado com LET, const ou Var, acabará criando uma nova variável global. Será um global, não importa agora profundamente aninhado nas funções e bloqueie seu código, o que quase não é o que você deseja, é propenso a insetos e é uma das melhores razões para usar o modo rigoroso!

As variáveis globais criadas dessa maneira acidental são como variáveis globais declaradas com VAR: elas definem propriedades do objeto global. Mas, diferentemente das propriedades definidas pelas declarações adequadas do VAR, essas propriedades podem ser excluídas com o operador de exclusão (§4.13.4).

3.10.3 Atribuição de destruição

O ES6 implementa um tipo de declaração composta e sintaxe de atribuição conhecida como atribuição de destruição. Em uma atribuição de destruição, o valor no lado direito do sinal igual é uma matriz ou objeto (um valor "estruturado"), e o lado da esquerda especifica um ou mais nomes de variáveis usando uma sintaxe que imita a matriz e a sintaxe literal do objeto. Quando ocorre uma atribuição de destruição, um ou mais valores são extraídos ("destruturados") do valor à direita e armazenados nas variáveis nomeadas à esquerda. A atribuição de destruição talvez seja mais comumente usada para inicializar variáveis como parte de uma declaração de declaração const, let ou VAR, mas também pode ser feita em expressões regulares de atribuição (com variáveis que já foram declaradas). E, como veremos no §8.3.5, a destruição também pode ser usada ao definir os parâmetros para uma função.

Aqui estão tarefas simples de destruição usando matrizes de valores:

```
Seja [x, y] = [1,2];    // o mesmo que let x = 1, y = 2
[x, y] = [x+1, y+1];  // o mesmo que x = x + 1, y = y + 1
[x, y] = [y, x];       // Troque o valor das duas variáveis
[X, y]                 // => [3,2]: o incrementado e trocado
valores
```

Observe como a atribuição de destruição facilita o trabalho com funções que retornam matrizes de valores:

```
// converte [x, y] coordenadas para as coordenadas polares [r, teta]
função topolar (x, y) {
    return [math.sqrt (x*x+y*y), math.atan2 (y, x)]; }

// Converter coordenadas polares para cartesianas
```

```
função toCartesian(r, theta) { return [r*Math.cos(theta),  
r*Math.sin(theta)];  
}  
  
deixe [r, theta] = toPolar (1,0, 1,0); // r == Math.sqrt(2);  
theta == Math.PI/4 let [x,y] =  
toCartesian(r,theta); // [x, y] == [1,0, 1,0]
```

Vimos que variáveis e constantes podem ser declaradas como parte de vários loops for do JavaScript. É possível usar a desestruturação de variáveis neste contexto também. Aqui está um código que percorre os pares nome/valor de todas as propriedades de um objeto e usa atribuição de desestruturação para converter esses pares de matrizes de dois elementos em variáveis individuais:

```
seja o = { x: 1, y: 2 }; // O objeto sobre o qual faremos  
o loop for(const [nome, valor] of Object.entries(o)) {  
  
    console.log(nome, valor); // Imprime "x" e "y"  
    //
```

O número de variáveis à esquerda de uma atribuição de desestruturação não precisa corresponder ao número de elementos do array à direita. Variáveis extras à esquerda são definidas como indefinidas e valores extras à direita são ignorados. A lista de variáveis à esquerda pode incluir vírgulas extras para pular determinados valores à direita:

```
deixe [x,y] = [1]; // x == 1; y == indefinido  
[x,y] = [1,2,3]; // x == 1; y == 2  
[,x,,y] = [1,2,3,4]; // x == 2; y == 4
```

Se você deseja coletar todos os valores não utilizados ou restantes em uma única variável ao desestruturar um array, use três pontos (...) antes do nome da última variável no lado esquerdo:

```
seja [x, ...y] = [1,2,3,4];      // e == [2,3,4]
```

Veremos três pontos usados desta forma novamente na Seção 8.3.2, onde eles são usados para indicar que todos os argumentos restantes da função devem ser coletados em um único array.

A atribuição de desestruturação pode ser usada com matrizes aninhadas. Nesse caso, o lado esquerdo da atribuição deve se parecer com um literal de array aninhado:

```
seja [a, [b, c]] = [1, [2,2,5], 3]; //a == 1; b == 2; c == 2,5
```

Um recurso poderoso da desestruturação de array é que na verdade ela não requer um array! Você pode usar qualquer objeto iterável (Capítulo 12) no lado direito da tarefa; qualquer objeto que possa ser usado com um loop for/of (§5.4.4) também pode ser desestruturado:

```
deixe [primeiro, ...descansar] = "Olá"; // primeiro ==  
"H"; resto ==  
["e",","l","l","o"]
```

A atribuição de desestruturação também pode ser executada quando o lado direito é um valor de objeto. Nesse caso, o lado esquerdo da atribuição se parece com um objeto literal: uma lista separada por vírgulas de nomes de variáveis entre chaves:

```
deixe transparente = {r: 0,0, g: 0,0, b: 0,0, a: 1,0}; // Uma cor RGBA  
seja {r, g, b} = transparente;      //r == 0,0; g == 0,0; b == 0,0
```

O próximo exemplo copia funções globais do objeto Math em variáveis, o que pode simplificar o código que faz muita trigonometria:

```
// O mesmo que const sin=Math.sin, cos=Math.cos, tan=Math.tan const  
{sin, cos, tan} = Math;
```

Observe no código aqui que o objeto Math possui muitas propriedades além das três que são desestruturadas em variáveis individuais. Aqueles que não são nomeados são simplesmente ignorados. Se o lado esquerdo deste

Se a atribuição incluísse uma variável cujo nome não fosse uma propriedade de Math, essa variável seria simplesmente atribuída como indefinida.

Em cada um desses exemplos de desestruturação de objetos, escolhemos nomes de variáveis que correspondem aos nomes de propriedades do objeto que estamos desestruturando. Isso mantém a sintaxe simples e fácil de entender, mas não é obrigatório. Cada um dos identificadores no lado esquerdo de uma atribuição de desestruturação de objeto também pode ser um par de identificadores separados por dois pontos, onde o primeiro é o nome da propriedade cujo valor deve ser atribuído e o segundo é o nome da variável a ser atribuída. para:

```
// O mesmo que const cosseno = Math.cos, tangent = Math.tan;  
const { cos: cosseno, tan: tangente } = Matemática;
```

Acho que a sintaxe de desestruturação de objetos se torna muito complicada para ser útil quando os nomes das variáveis e dos nomes das propriedades não são os mesmos, e tendo a evitar a abreviação neste caso. Se você optar por usá-lo, lembre-se de que os nomes das propriedades estão sempre à esquerda dos dois pontos, em ambos os literais de objeto e à esquerda de uma atribuição de desestruturação de objeto.

A desestruturação da atribuição torna-se ainda mais complicada quando usada com objetos aninhados, ou matrizes de objetos, ou objetos de matrizes, mas é legal:

```
sejam pontos = [{x: 1, y: 2}, {x: 3, y: 4}];           //Uma matriz de
objetos de dois pontos sejam [{x: x1, y: y1}, {x: x2,
y: y2}] = pontos; // desestruturado em 4 variáveis.

(x1 === 1 && y1 === 2 && x2 === 3 && y2 === 4) // =>
verdadeiro
```

Ou, em vez de desestruturar um array de objetos, poderíamos desestruturar um objeto de arrays:

```
deixe pontos = { p1: [1,2], p2: [3,4] };           //Um objeto
com 2 endereços de array, deixe { p1: [x1, y1],
p2: [x2, y2] } = pontos;                         //
desestruturado em 4 vars (x1 === 1 && y1 === 2 && x2 ===
3 && y2 === 4) // => verdadeiro
```

Sintaxes de desestruturação complexas como essa podem ser difíceis de escrever e de ler, e talvez seja melhor apenas escrever suas atribuições explicitamente com código tradicional como let x1 = points.p1[0];.

ENTENDENDO A DESESTRUTURAÇÃO COMPLEXA

Se você estiver trabalhando com código que usa atribuições de desestruturação complexas, existe uma regularidade útil que pode ajudá-lo a entender os casos complexos. Pense primeiro em uma atribuição regular (de valor único). Depois que a atribuição for concluída, você pode pegar o nome da variável do lado esquerdo da atribuição e usá-lo como uma expressão em seu código, onde será avaliado para qualquer valor que você atribuiu. A mesma coisa se aplica à desestruturação da atribuição. O lado esquerdo de uma atribuição de desestruturação se parece com um literal de array ou um literal de objeto (§6.2.1 e §6.10). Depois que a atribuição for concluída, o lado esquerdo funcionará como um literal de array ou literal de objeto válido em outro lugar do seu código. Você pode verificar se escreveu uma atribuição de desestruturação corretamente tentando usar o lado esquerdo do lado direito de outra expressão de atribuição:

```
// Comece com uma estrutura de dados e uma desestruturação
complexa let points = [{x: 1, y: 2}, {x: 3, y: 4}];
deixe [{x: x1, y: y1}, {x: x2, y: y2}] = pontos;

// Verifique sua sintaxe de desestruturação invertendo a atribuição let
points2 = [{x: x1, y: y1}, {x: x2, y: y2}]; // pontos2 == pontos
```

3.11 Resumo

Alguns pontos -chave a serem lembrados sobre este capítulo:

- Como escrever e manipular números e seqüências de texto em JavaScript.
- Como trabalhar com outros tipos primitivos de JavaScript: booleanos, símbolos, nulos e indefinidos.
- As diferenças entre tipos primitivos imutáveis e tipos de referência mutável.
- Como o JavaScript converte valores implicitamente de um tipo para outro e como você pode fazê -lo explicitamente em seus programas.
- Como declarar e inicializar constantes e variáveis (inclusive com a atribuição de destruição) e o escopo lexical das variáveis e constantes que você declara.

Este é o formato para números do tipo duplo em linguagens de programação

¹ Java, C ++ e mais modernas.

² Existem extensões de JavaScript, como TypeScript e Flow (§17.8), que permitem que os tipos sejam especificados como parte de declarações variáveis com sintaxe como let x: número = 0;.

Capítulo 4. Expressões e operadores

Este capítulo documenta expressões JavaScript e os operadores com os quais muitas dessas expressões são construídas. Uma expressão é uma frase de JavaScript que pode ser avaliada para produzir um valor. Um constante incorporado literalmente em seu programa é um tipo muito simples de expressão. Um nome de variável também é uma expressão simples que avalia qualquer valor que tenha sido atribuído a essa variável. Expressões complexas são construídas a partir de expressões mais simples. Uma expressão de acesso à matriz, por exemplo, consiste em uma expressão que avalia a uma matriz seguida por um suporte quadrado aberto, uma expressão que avalia a um número inteiro e um suporte quadrado próximo. Essa nova expressão mais complexa avalia o valor armazenado no índice especificado da matriz especificada. Da mesma forma, uma expressão de invocação de função consiste em uma expressão que avalia a um objeto de função e expressões zero ou mais adicionais que são usadas como argumentos da função.

A maneira mais comum de construir uma expressão complexa a partir de expressões mais simples é com um operador. Um operador combina os valores de seus operandos (geralmente dois deles) de alguma forma e avalia um novo valor. O operador de multiplicação * é um exemplo simples. A expressão $x * y$ avalia o produto dos valores das expressões x e y . Por simplicidade, às vezes dizemos que um operador retorna um valor em vez de "avaliar" um valor.

Este capítulo documenta todos os operadores do JavaScript e também explica expressões (como indexação de matrizes e invocação de funções) que não usam operadores. Se você já conhece outra linguagem de programação que usa sintaxe no estilo C, descobrirá que a sintaxe da maioria das expressões e operadores do JavaScript já é familiar para você.

4.1 Expressões primárias

As expressões mais simples, conhecidas como expressões primárias, são aquelas que permanecem sozinhas - elas não incluem expressões mais simples. As expressões primárias no JavaScript são valores constantes ou literais, certas palavras-chave do idioma e referências variáveis.

Os literais são valores constantes incorporados diretamente em seu programa. Eles se parecem com estes:

```
1.23          // um número literal  
&quot;olá&quot; // uma string literal  
/padrão/      // uma expressão regular literal
```

A sintaxe JavaScript para literais numéricos foi coberta no §3.2. Os literais de string foram documentados no §3.3. A sintaxe literal de expressão regular foi introduzida no §3.3.5 e será documentada em detalhes no §11.3.

Algumas das palavras reservadas de JavaScript são expressões primárias:

```
verdadeir     // avalia para o valor verdadeiro booleano  
o           // avalia o valor falso booleano  
falso        // avalia o valor nulo  
nulo         // avalia o valor nulo  
esse         // avalia o objeto &quot;atual&quot;;
```

Aprendemos sobre verdadeiro, falso e nulo em §3.4 e §3.5. Diferente

As outras palavras -chave, isso não é uma constante - avalia os valores diferentes em diferentes locais do programa. A palavra-chave esta é usada na programação orientada a objetos. Dentro do corpo de um método, isso avalia o objeto no qual o método foi invocado. Veja §4.5, capítulo 8 (especialmente §8.2.2) e capítulo 9 para saber mais sobre isso.

Finalmente, o terceiro tipo de expressão primária é uma referência a uma variável, constante ou propriedade do objeto global:

```
eu           // Avalia o valor da variável i.  
soma         // Avalia o valor da soma variável.  
indefinido // o valor da propriedade "indefinida" do  
               // objeto global
```

Quando qualquer identificador aparece por si só em um programa, o JavaScript pressupõe que seja uma variável ou constante ou propriedade do objeto global e procure seu valor. Se não houver variável com esse nome, uma tentativa de avaliar uma variável inexistente lança um `referenceError`.

4.2 Inicializadores de objeto e matriz

Inicializadores de objeto e matriz são expressões cujo valor é um objeto ou matriz recém -criado. Essas expressões inicializadoras às vezes são chamadas de literais de objetos e literais de matriz. Ao contrário dos literais verdadeiros, no entanto, eles não são expressões primárias, porque incluem várias subexpressões que especificam valores de propriedade e elemento. Os inicializadores da matriz têm uma sintaxe um pouco mais simples, e começaremos com eles.

Um inicializador de matriz é uma lista de expressões separada por vírgula contida em colchetes. O valor de um inicializador de matriz é um recém

matriz criada. Os elementos desta nova matriz são inicializados para os valores das expressões separadas por vírgula:

```
[]           // Uma matriz vazia: sem expressões dentro de colchetes  
significa que não há elementos  
[1+2,3+4] // uma matriz de 2 elementos. O primeiro elemento é 3, segundo  
é 7
```

As expressões de elementos em um inicializador de matriz podem ser iniciantes de matriz, o que significa que essas expressões podem criar matrizes aninhadas:

```
Let Matrix = [[1,2,3], [4,5,6], [7,8,9]];
```

As expressões de elemento em um inicializador de matriz são avaliadas sempre que o inicializador da matriz é avaliado. Isso significa que o valor de uma expressão de inicializador da matriz pode ser diferente cada vez que é avaliado.

Elementos indefinidos podem ser incluídos em uma matriz literal, simplesmente omitindo um valor entre vírgulas. Por exemplo, a matriz a seguir contém cinco elementos, incluindo três elementos indefinidos:

```
Seja SparsArray = [1 , , , 5];
```

Uma única vírgula à direita é permitida após a última expressão em um inicializador de matriz e não cria um elemento indefinido. No entanto, qualquer expressão de acesso à matriz para um índice após a última expressão será necessariamente avaliada como indefinida.

As expressões de inicializador de objetos são como expressões de inicializador de matriz, mas os colchetes são substituídos por colchetes encaracolados, e cada subexpressão é prefixada com um nome de propriedade e um colón:

```
seja p = {x: 2,3, y: -1,2};           //Um objeto com 2 propriedades
seja q = {};                          // Um objeto vazio sem
propriedades qx = 2,3; qy =           // Agora q tem o mesmo
-1,2;                                // propriedades como p
```

No ES6, os literais de objeto têm uma sintaxe muito mais rica em recursos (você pode encontrar detalhes na Seção 6.10). Literais de objeto podem ser aninhados. Por exemplo:

```
-----  
deixe retângulo =  
{superiorEsquerda: {x: 2, y: 2},  
inferiorDireita: {x: 4, y: 5}  
};
```

Veremos inicializadores de objetos e arrays novamente nos Capítulos 6 e 7.

4.3 Expressões de Definição de Função

Uma expressão de definição de função define uma função JavaScript, e o valor dessa expressão é a função recém-definida. De certo modo, uma expressão de definição de função é uma “literal de função” da mesma forma que um inicializador de objeto é um “literal de objeto”. Uma expressão de definição de função normalmente consiste na palavra-chave `function` seguida por uma lista separada por vírgulas de zero ou mais identificadores (os nomes dos parâmetros) entre parênteses e um bloco de código JavaScript (o corpo da função) entre chaves. Por exemplo:

```
// Esta função retorna o quadrado do valor passado para ela.  
  
deixe quadrado = função (x) { return x * x; };
```

Uma expressão de definição de função também pode incluir um nome para o

função. As funções também podem ser definidas usando uma declaração de função em vez de uma expressão de função. E no ES6 e posterior, as expressões de função podem usar uma nova sintaxe compacta "Função de seta". Detalhes completos sobre a definição da função estão no capítulo 8.

4.4 Expressões de acesso à propriedade

Uma expressão de acesso à propriedade avalia o valor de uma propriedade de objeto ou um elemento de matriz.

JavaScript define duas sintaxes para acesso à propriedade:

expressão. Expressão do identificador [expressão]

O primeiro estilo de acesso à propriedade é uma expressão seguida por um período e um identificador. A expressão especifica o objeto e o identificador especifica o nome da propriedade desejada. O segundo estilo de acesso à propriedade segue a primeira expressão (o objeto ou a matriz) com outra expressão entre colchetes. Esta segunda expressão especifica o nome da propriedade desejada ou o índice do elemento de matriz desejado. Aqui estão alguns exemplos concretos:

```
Seja o = {x: 1, y: {z: 3}}; // Um exemplo de objeto Seja a =
[0, 4, [5, 6]]; // uma matriz de exemplo que contém o objeto

o.x                                // => 1: Propriedade X da expressão
0 OYZ                               // => 3: Propriedade z da expressão
0y ou "x"                           // => 1: Propriedade x do objeto o
A [1]                                // => 4: elemento no índice 1 de
Expressão AA [2]
["1"]                                 // => 6: elemento no índice 1 de
expressão a [2]
```

```
a [0] .x // => 1: Propriedade X de expressão  
A [0]
```

Com qualquer tipo de expressão de acesso à propriedade, a expressão antes do. ou [é primeiro avaliado. Se o valor for nulo ou indefinido, a expressão lança um `TypeError`, pois esses são os dois valores de JavaScript que não podem ter propriedades. Se a expressão do objeto for seguida por um ponto e um identificador, o valor da propriedade nomeada por esse identificador será procurada e se tornará o valor geral da expressão. Se a expressão do objeto for seguida por outra expressão entre colchetes, essa segunda expressão será avaliada e convertida em uma corda. O valor geral da expressão é então o valor da propriedade nomeada por essa string. Em ambos os casos, se a propriedade nomeada não existir, o valor da expressão de acesso à propriedade será indefinido.

A sintaxe `.Identifier` é a mais simples das duas opções de acesso à propriedade, mas observe que só pode ser usado quando a propriedade que você deseja acessar tem um nome que é um identificador legal e quando você souber o nome quando escreve o programa. Se o nome da propriedade incluir espaços ou caracteres de pontuação ou quando for um número (para matrizes), você deverá usar a notação de suporte quadrado. Os suportes quadrados também são usados quando o nome da propriedade não é estático, mas é o resultado de um cálculo (consulte §6.3.1 por exemplo).

Os objetos e suas propriedades são abordados em detalhes no capítulo 6, e as matrizes e seus elementos são abordados no capítulo 7.

4.4.1 Acesso à propriedade condicional

O ES2020 adiciona dois novos tipos de expressões de acesso à propriedade:

```
expressão?. expressão do  
identificador?.[expressão]
```

Em JavaScript, os valores nulos e indefinidos são os únicos dois valores que não possuem propriedades. Em uma expressão regular de acesso à propriedade usando . ou [], você obterá um TypeError se a expressão à esquerda for avaliada como nula ou indefinida. Você pode usar? e sintaxe ?.[] para proteção contra erros desse tipo.

Considere a expressão a?.b. Se a for nulo ou indefinido, a expressão será avaliada como indefinida sem qualquer tentativa de acessar a propriedade b. Se a for algum outro valor, então a?.b será avaliado como qualquer valor que ab avaliaria (e se a não tiver uma propriedade chamada b, o valor será novamente indefinido).

Esta forma de expressão de acesso a propriedade é às vezes chamada de “encadeamento opcional” porque também funciona para expressões de acesso a propriedade “encadeadas” mais longas como esta:

```
deixe a = { b: nulo }; ab?.cd // => indefinido
```

a é um objeto, então ab é uma expressão válida de acesso à propriedade. Mas o valor de ab é nulo, então abc geraria um TypeError. Usando ?. em vez de . evitamos o TypeError e ab?.c é avaliado como indefinido. Isso significa que (ab?.c).d lançará um TypeError, porque essa expressão tenta acessar uma propriedade de valor indefinido. Mas – e esta é uma parte muito importante do “encadeamento opcional” – ab?.cd (sem os parênteses) simplesmente avalia como

indefinido e não apresenta um erro. Isso ocorre porque o acesso à propriedade com?. é "curto-circuito": se a subexpressão à esquerda? Avalia como nulo ou indefinido, e toda a expressão avalia imediatamente como indefinida sem nenhuma tentativa adicional de acesso à propriedade.

Obviamente, se AB é um objeto, e se esse objeto não tiver propriedade denominada C, então AB?

```
Seja a = {b: {}}; ab?.c?.d // => indefinido
```

O acesso à propriedade condicional também é possível usando?. [] Em vez de []. Na expressão a?.[B].[c], se o valor de a é nulo ou Indefinido, então toda a expressão avalia imediatamente a indefinida e as subexpressões B e C nunca são avaliadas. Se qualquer uma dessas expressões tiver efeitos colaterais, o efeito colateral não ocorrerá se A não estiver definido:

```
deixe um;           // opa, esquecemos de inicializar isso
variável!
deixe index =
0; tentar {
  a [index ++]; // lança TypeError} catch (e)
{index // => 1: Ocorre o incremento antes que
o TypeError seja jogado
} a??.[Index
++];           // => indefinido: porque a é indefinido
índice          // => 1: não incrementado porque? [] Curto
CIRCUITOS A
[INDEX ++]      /// TypeError: Não é possível indexar indefinidos.
```

Acesso condicional à propriedade com `?.` e `?[]` é um dos mais novos recursos do JavaScript. Desde o início de 2020, esta nova sintaxe é suportada nas versões atuais ou beta da maioria dos principais navegadores.

4.5 Expressões de Invocação

Uma expressão de invocação é a sintaxe do JavaScript para chamar (ou executar) uma função ou método. Começa com uma expressão de função que identifica a função a ser chamada. A expressão da função é seguida por um parêntese aberto, uma lista separada por vírgulas de zero ou mais expressões de argumento e um parêntese fechado. Alguns exemplos:

```
f(0)          // f é a expressão da função; 0 é o
expressão do argumento. Math.max(x,y,z) //
Math.max é a função; x, y e z são os argumentos.

a.sort()      // a.sort é a função; não há
argumentos.
```

Quando uma expressão de invocação é avaliada, a expressão da função é avaliada primeiro e, em seguida, as expressões dos argumentos são avaliadas para produzir uma lista de valores dos argumentos. Se o valor da expressão da função não for uma função, um `TypeError` será lançado. Em seguida, os valores dos argumentos são atribuídos, em ordem, aos nomes dos parâmetros especificados quando a função foi definida e, em seguida, o corpo da função é executado. Se a função usar uma instrução `return` para retornar um valor, esse valor se tornará o valor da expressão de invocação. Caso contrário, o valor da expressão de invocação será indefinido. Detalhes completos sobre a invocação de funções, incluindo uma explicação do que acontece quando o número de expressões de argumentos não corresponde ao número de parâmetros na definição da função, estão no Capítulo 8.

Toda expressão de invocação inclui um par de parênteses e uma expressão antes dos parênteses abertos. Se essa expressão for uma expressão de acesso à propriedade, a invocação é conhecida como um método

invocação. Nas invocações do método, o objeto ou a matriz que é o assunto do acesso à propriedade se torna o valor dessa palavra -chave enquanto o corpo da função está sendo executado. Isso permite um paradigma de programação orientado a objetos no qual as funções (que chamamos de "métodos" quando usadas dessa maneira) operam no objeto do qual elas fazem parte. Veja o capítulo 9 para obter detalhes.

4.5.1 Invocação condicional

No ES2020, você também pode invocar uma função usando?. () Em vez de (). Normalmente, quando você chama uma função, se a expressão à esquerda dos parênteses for nula ou indefinida ou qualquer outra função, um TypeError será lançado. Com a nova sintaxe de invocação, se a expressão à esquerda do? Avalia como nulo ou indefinido, então toda a expressão de invocação avalia a indefinida e nenhuma exceção é lançada.

Os objetos da matriz possuem um método Sort () que, opcionalmente, pode ser aprovado em um argumento de função que define a ordem de classificação desejada para os elementos da matriz. Antes do ES2020, se você quisesse escrever um método como o Sorn () que leva um argumento de função opcional, normalmente usaria uma instrução IF para verificar se o argumento da função foi definido antes de invocá-lo no corpo do IF:

```
Função quadrada (x, log) { // O segundo argumento é um
```

```
função opcional
    if (log) {                                // se a função opcional for
passou
        log (x);                            // Invocar
    } retornar x * x;                      // retorna o quadrado do
argumento}
```

Com essa sintaxe de invocação condicional do ES2020, no entanto, você pode simplesmente escrever a invocação de funções usando? () . Sabendo que a invocação só acontecerá se houver um valor a ser chamado:

```
Função quadrada (x, log) { // O segundo
argumento é um log de função opcional?. (x);
// Ligue para a função se houver um
    retornar x * x;                      // retorna o quadrado do
argumento}
```

Observe, no entanto, isso?. () Só verifica se o lado da esquerda é nulo ou indefinido. Não verifica que o valor é realmente uma função. Portanto, a função Square () neste exemplo ainda lançaria uma exceção se você passasse dois números, por exemplo.

Como expressões de acesso à propriedade condicional (§4.4.1), a invocação da função com?. () É de curto-circuito: se o valor à esquerda de? é nulo ou indefinido, então nenhuma das expressões de argumento nos parênteses é avaliada:

```
Seja f = nulo, x = 0; tentar {
    f (x ++); // joga TypeError porque f é nulo} catch (e) {
```

```
x          // > 1: x é incrementado antes da exceção  
é lançado }  
  
f?.(x++)   // > indefinido: f é nulo, mas sem exceção  
jogado  
x          // > 1: o incremento é ignorado devido a curto  
circuito
```

Expressões de invocação condicional com `?()` funcionam tão bem para métodos quanto para funções. Mas como a invocação de métodos também envolve acesso a propriedades, vale a pena reservar um momento para ter certeza de que você entendeu as diferenças entre as seguintes expressões:

```
o.m()      // Acesso regular à propriedade, invocação regular  
o?.m()     // Acesso condicional à propriedade, invocação regular  
o?.m()     // Acesso regular à propriedade, invocação condicional
```

Na primeira expressão, o deve ser um objeto com propriedade m e o valor dessa propriedade deve ser uma função. Na segunda expressão, se o for nulo ou indefinido, a expressão será avaliada como

indefinido. Mas se o tiver qualquer outro valor, então deve ter uma propriedade m cujo valor é uma função. E na terceira expressão, o não deve ser nulo ou indefinido. Se não tiver uma propriedade m, ou se o valor dessa propriedade for nulo, toda a expressão será avaliada como indefinida.

A invocação condicional com `?()` é um dos recursos mais recentes do JavaScript. A partir dos primeiros meses de 2020, esta nova sintaxe é suportada nas versões atuais ou beta da maioria dos principais navegadores.

4.6 Expressões de Criação de Objetos

Uma expressão de criação de objeto cria um novo objeto e invoca uma função (chamada construtor) para inicializar as propriedades desse objeto. As expressões de criação de objetos são como expressões de invocação, exceto pelo fato de serem prefixadas com a palavra-chave new:

```
novo Objeto()  
novo Ponto(2,3)
```

Se nenhum argumento for passado para a função construtora em uma expressão de criação de objeto, o par vazio de parênteses poderá ser omitido:

```
novo objeto  
nova data
```

O valor de uma expressão de criação de objeto é o objeto recém-criado. Os construtores são explicados com mais detalhes no Capítulo 9.

4.7 Visão Geral do Operador

Os operadores são usados para expressões aritméticas, expressões de comparação, expressões lógicas, expressões de atribuição de JavaScript e muito mais. A Tabela 4-1 resume os operadores e serve como referência conveniente.

Observe que a maioria dos operadores são representados por caracteres de pontuação como + e =. Alguns, entretanto, são representados por palavras-chave como delete e instanceof. Os operadores de palavras-chave são operadores regulares, assim como aqueles expressos com pontuação; eles simplesmente têm uma sintaxe menos sucinta.

A Tabela 4-1 está organizada por precedência de operador. As operadoras listadas

Primeiro, tem maior precedência do que os listados por último. Os operadores separados por uma linha horizontal têm diferentes níveis de precedência. A coluna rotulada como fornece ao operador associatividade, que pode ser L (da esquerda para a direita) ou R (direita para a esquerda), e a coluna N especifica o número de operandos. Os tipos rotulados pela coluna lista os tipos esperados de operandos e (após o símbolo →) o tipo de resultado para o operador. As subseções que seguem a tabela explicam os conceitos de precedência, associatividade e tipo de operando. Os próprios operadores são documentados individualmente após essa discussão.

Tabela 4-1. Operadores JavaScript

Operador	Operação	UM	Tipos
<code>++</code>	Pré ou pós-incremento	R	1 lval → num
<code>--</code>	Pré ou pós-decisão	R	1 lval → num
<code>-</code>	Negar número	R 1	num → num
<code>+</code>	Converter em númeroR	1	any → num
<code>~</code>	Inverter bits	R 1	int → int
<code>!</code>	Inverter o valor booleanoR	1	bool → bool
<code>excluir</code>	Remova uma propriedade R	1	lval → bool
<code>typeof</code>	Determine o tipo de operando R	1	any → str
<code>vazio</code>	Retornar valor indefinido R	1	any → undef
<code>**</code>	Exponencial	R 2	num,num → num
<code>*, /, %</code>	Multiplique, dividir, restante	L	num,num → num
<code>+, -</code>	Adicionar, subtrair	L 2	num,num → num

+	Cordas concatenadas	L	2	str, str → str
<<	Mudança para a esquerda		L 2	int, int → int
>>	Mudar à direita com a extensão do sinal	L	2	int, int → int
>>>	Mudar à direita com extensão zero	L	2	int, int → int
<, <=,>, >=	Compare em ordem numérica	L	2	num,num → bool
<, <=,>, >=	Compare em ordem alfabética	L	2	str, str → bool
Instância de	Teste da classe de objeto		L 2	Obj, func → bool
em	Teste se a propriedade existe	L	2	qualquer, obj → bool
==	Teste para igualdade não rigorosa	L	2	qualquer, qualque → bool
!=	Teste para desigualdade não esticada	L	2	qualquer, qualque → bool
==≡	Teste para igualdade estrita	L	2	qualquer, qualque → bool
!==	Teste para desigualdade estrita	L	2	qualquer, qualque → bool
&	Calcule bitwise e	L	2	int, int → int
^	Calcule o XOR Bitwise	L	2	int, int → int
 	Calcule bitwise ou	L	2	int, int → int
&&	Calcule lógico e	L	2	qualquer, qualque → qualque
 	Calcule lógico ou	L	2	qualquer, qualque → qualque
??	Escolha 1º operando definido	L	2	qualquer, qualque → qualque
?:	Escolha o 2º ou 3º operando	R	3	bool, qualque, qualque → um y
=	Atribuir a uma variável ou propriedade		R 2	LVAL, qualke → qualke
**=, *=, /=, %=,	Operar e atribuir	R	2	LVAL, qualke → qualke

`+=, -=, &=,`
`^=, |=,`

`<<=, >>=, >>>=`

`,`

Descartar o 1º operando,
retornar o 2º

`eu2`

qualquer, qualquer → qualquer

4.7.1 Quantidade de Operandos

Os operadores podem ser categorizados com base no número de operandos que esperam (sua aridade). A maioria dos operadores JavaScript, como o operador de multiplicação `*`, são operadores binários que combinam duas expressões em uma expressão única e mais complexa. Ou seja, eles esperam dois operandos. JavaScript também oferece suporte a vários operadores unários, que convertem uma única expressão em uma expressão única e mais complexa. O operador `-` na expressão `-x` é um operador unário que realiza a operação de negação no operando `x`. Finalmente, JavaScript suporta um operador ternário, o operador condicional `?:`, que combina três expressões em uma única expressão.

4.7.2 Operando e Tipo de Resultado

Alguns operadores trabalham com valores de qualquer tipo, mas a maioria espera que seus operandos sejam de um tipo específico, e a maioria dos operadores retorna (ou avalia) um valor de um tipo específico. A coluna Tipos na Tabela 4-1 especifica os tipos de operandos (antes da seta) e o tipo de resultado (após a seta) para os operadores.

Os operadores JavaScript geralmente convertem o tipo (ver §3.9) de seus operandos conforme necessário. O operador de multiplicação `*` espera numérico

operandos, mas a expressão "3*5" é válida porque o JavaScript pode converter os operandos em números. O valor desta expressão é o número 15, não a string “15”, claro. Lembre-se também de que todo valor JavaScript é “verdadeiro” ou “falso”, portanto, os operadores que esperam operandos booleanos funcionarão com um operando de qualquer tipo.

Alguns operadores se comportam de maneira diferente dependendo do tipo de operandos utilizados com eles. Mais notavelmente, o operador + adiciona operandos numéricos, mas concatena operandos de string. Da mesma forma, os operadores de comparação como < realizam a comparação em ordem numérica ou alfabética dependendo do tipo dos operandos. As descrições de operadores individuais explicam suas dependências de tipo e especificam quais conversões de tipo eles realizam.

Observe que os operadores de atribuição e alguns dos outros operadores listados na Tabela 4-1 esperam um operando do tipo lval. lvalue é um termo histórico que significa “uma expressão que pode aparecer legalmente no lado esquerdo de uma expressão de atribuição”. Em JavaScript, variáveis, propriedades de objetos e elementos de arrays são lvalores.

4.7.3 Efeitos colaterais do operador

Avaliar uma expressão simples como $2 * 3$ nunca afeta o estado do seu programa, e qualquer cálculo futuro executado pelo seu programa não será afetado por essa avaliação. Algumas expressões, porém, apresentam efeitos colaterais e sua avaliação pode afetar o resultado de avaliações futuras. Os operadores de atribuição são o exemplo mais óbvio: se você atribuir um valor a uma variável ou propriedade, isso altera o valor de qualquer expressão que use essa variável ou propriedade. O ++ e -

Os operadores de incremento e decremento são semelhantes, pois realizam uma atribuição implícita. O operador delete também tem efeitos colaterais: excluir uma propriedade é como (mas não o mesmo que) atribuir indefinido à propriedade.

Nenhum outro operador JavaScript tem efeitos colaterais, mas a invocação de função e as expressões de criação de objeto terão efeitos colaterais se algum dos operadores usados na função ou no corpo do construtor tiver efeitos colaterais.

4.7.4 Precedência do Operador

Os operadores listados na Tabela 4-1 são organizados em ordem de alta precedência para baixa precedência, com linhas horizontais separando grupos de operadores no mesmo nível de precedência. A precedência do operador controla a ordem em que as operações são executadas. Operadores com precedência mais alta (mais próximos do topo da tabela) são executados antes daqueles com precedência mais baixa (mais perto da parte inferior).

Considere a seguinte expressão:

```
w = x + y*z;
```

O operador de multiplicação * tem precedência maior que o operador de adição +, portanto a multiplicação é realizada antes da adição. Além disso, o operador de atribuição = tem a precedência mais baixa, portanto a atribuição é executada após todas as operações do lado direito serem concluídas.

A precedência do operador pode ser substituída pelo uso explícito de parênteses. Para forçar a adição no exemplo anterior a ser

realizado primeiro, escreva:

```
w = (x + y)*z;
```

Observe que o acesso à propriedade e as expressões de invocação têm precedência mais alta do que qualquer um dos operadores listados na Tabela 4-1. Considere esta expressão:

```
// my é um objeto com uma propriedade chamada funções cujo  
valor é um  
// array de funções. Invocamos a função número x, passando-lhe  
o argumento  
// y, e então perguntamos o tipo do valor retornado. tipo de  
minhas.funções[x](y)
```

Embora `typeof` seja um dos operadores de maior prioridade, a operação `typeof` é executada no resultado do acesso à propriedade, índice de array e invocação de função, todos os quais têm prioridade mais alta que os operadores.

Na prática, se você não tiver certeza sobre a precedência de seus operadores, a coisa mais simples a fazer é usar parênteses para tornar explícita a ordem de avaliação. As regras que é importante conhecer são estas: a multiplicação e a divisão são realizadas antes da adição e da subtração, e a atribuição tem precedência muito baixa e quase sempre é executada por último.

Quando novos operadores são adicionados ao JavaScript, eles nem sempre se enquadram naturalmente neste esquema de precedência. O `??` operador (§4.13.2) é mostrado na tabela como de precedência inferior a `||` e `&&`, mas, na verdade, sua precedência em relação a esses operadores não está definida, e o ES2020 exige que você use parênteses explicitamente se você misturar `??` com `||`.

ou $\&$ $\&$. Da mesma forma, o novo operador de exponenciação $**$ não tem uma precedência bem definida em relação ao operador de negação unário, e você deve usar parênteses ao combinar a negação com exponenciação.

4.7.5 Associatividade do Operador

Na Tabela 4-1, a coluna denominada A especifica a associatividade do operador. Um valor de L especifica a associatividade da esquerda para a direita e um valor de R especifica a associatividade da direita para a esquerda. A associatividade de um operador especifica a ordem em que as operações de mesma precedência são executadas. Associatividade da esquerda para a direita significa que as operações são realizadas da esquerda para a direita. Por exemplo, o operador de subtração tem associatividade da esquerda para a direita, então:

```
w = x - y - z;
```

é o mesmo que:

```
w = ((x - y) - z);
```

Por outro lado, as seguintes expressões:

```
y = a ** b ** c; x = ~-y;
```

```
w = x = y = z; q = a?b:c?d:e?f:g;
```

são equivalentes a:

```
y = (a ** (b ** c)); x = ~(-y);
```

```
w = (x = (y = z)); q =  
a? B: (c? D: (e? f: g));
```

Porque os operadores condicionais de exponenciação, unário, atribuição e ternário têm associatividade da direita para a esquerda.

4.7.6 Ordem de avaliação

Precedência e associativa do operador especificam a ordem em que as operações são realizadas em uma expressão complexa, mas não especificam a ordem em que as subexpressões são avaliadas. O JavaScript sempre avalia expressões em ordem estritamente da esquerda para a direita. Na expressão `w = x + y * z`, por exemplo, a subexpressão `w` é avaliada primeiro, seguida por `x`, `y` e `z`. Em seguida, os valores de `Y` e `Z` são multiplicados, adicionados ao valor de `x` e atribuídos à variável ou propriedade especificada pela expressão `w`. Adicionar parênteses às expressões pode alterar a ordem relativa da multiplicação, adição e atribuição, mas não a ordem de avaliação da esquerda para a direita.

A ordem de avaliação faz a diferença se alguma das expressões avaliadas tiver efeitos colaterais que afetam o valor de outra expressão. Se a expressão `X` incrementa uma variável usada pela expressão `z`, o fato de `X` ser avaliado antes de `Z` é importante.

4.8 Expressões aritméticas

Esta seção abrange os operadores que realizam manipulações aritméticas ou outras manipulações numéricas em seus operandos. Os operadores de exponenciação, multiplicação, divisão e subtração são diretos

e são cobertos primeiro. O operador de adição recebe uma subseção própria porque também pode executar concatenação de string e possui algumas regras de conversão de tipo incomum. Os operadores unários e os operadores bitwewward também são cobertos por suas próprias subseções.

A maioria desses operadores aritméticos (exceto como observado da seguinte forma) pode ser usada com operando BIGINT (consulte §3.2.5) ou com números regulares, desde que você não misture os dois tipos.

Os operadores aritméticos básicos são `**` (exponenciação), `*` (multiplicação), `/` (divisão), `%` (módulo: restante após a divisão), `+` (adição) e `-` (subtração). Como observado, discutiremos o operador `+` em uma seção própria. Os outros cinco operadores básicos simplesmente avaliam seus operandos, convertem os valores em números, se necessário, e depois calcule a energia, produto, quociente, restante ou diferença. Operando não numéricos que não podem se converter em números convertem para o valor da NAN. Se um operando for (ou converter para) nan, o resultado da operação é (quase sempre) a NAN.

O operador `**` tem maior precedência que `*`, `/` e `%` (que por sua vez têm maior precedência que `+` e `-`). Ao contrário dos outros operadores, `**` funciona da direita para a esquerda, então `2 ** 2 ** 3` é o mesmo que `2 ** 8`, não `4 ** 3`. Há uma ambiguidade natural em expressões como `-3 ** 2`. Dependendo da precedência relativa de unário menos e exponenciação, essa expressão pode significar `(-3) ** 2` ou `-(3 ** 2)`. Diferentes linguagens lidam com isso de maneira diferente e, em vez de escolher lados, o JavaScript simplesmente faz com que seja um erro de sintaxe omitir parênteses neste caso, forçando você a escrever uma expressão inequívoca. `**` é a mais nova aritmética de JavaScript

Operador: foi adicionado ao idioma com ES2016. A função Math.pow() está disponível desde as primeiras versões do JavaScript, no entanto, e executa exatamente a mesma operação que o operador **.

O operador divide seu primeiro operando pelo segundo. Se você estiver acostumado a programação de linguagens que distinguem entre números inteiros e de ponto flutuante, pode esperar obter um resultado inteiro ao dividir um número inteiro por outro. Em JavaScript, no entanto, todos os números são pontos flutuantes, portanto, todas as operações de divisão têm resultados de ponto flutuante: $5/2$ avalia para 2,5, não 2. A divisão por zero produz infinito positivo ou negativo, enquanto $0/0$ avalia para a NAN: nem Desses casos, levanta um erro.

O operador % calcula o primeiro módulo de operando o segundo operando. Em outras palavras, ele retorna o restante após a divisão de número inteiro do primeiro operando pelo segundo operando. O sinal do resultado é o mesmo que o sinal do primeiro operando. Por exemplo, $5 \% 2$ avalia para 1 e $-5 \% 2$ avalia para -1.

Embora o operador do Modulo seja normalmente usado com operandos inteiros, ele também funciona para valores de ponto flutuante. Por exemplo, $6,5 \% 2,1$ avalia para 0,2.

4.8.1 O operador +

O operador binário + adiciona operandos numéricos ou concatena operandos de string:

```
1 + 2 // => 3
"Olá" + " " // => "Olá "
"1" + "2" // => "12"
```

Quando os valores de ambos os operando são números ou são ambas as seqüências, é óbvio o que o operador + faz. Em qualquer outro caso, no entanto, a conversão do tipo é necessária e a operação a ser realizada depende da conversão realizada. As regras de conversão para + dão prioridade à concatenação da string: se um dos operandos for uma string ou um objeto que se converte em uma string, o outro operando é convertido em uma string e a concatenação é realizada. A adição é realizada apenas se nenhum operando for parecido com cordas.

Tecnicamente, o operador + se comporta assim:

- Se um dos seus valores de operando for um objeto, ele o converte em um primitivo usando o algoritmo objeto para princípio descrito em §3.9.3. Os objetos de data são convertidos pelo método `ToString()`, e todos os outros objetos são convertidos via `valueof()`, se esse método retornar um valor primitivo. No entanto, a maioria dos objetos não possui um método `ValueOf()` útil, portanto, eles também são convertidos via `ToString()`.
- Após a conversão de objeto para princípios, se um operando for uma string, o outro será convertido em uma string e a concatenação será realizada.
- Caso contrário, ambos os operando são convertidos em números (ou na NAN) e a adição é realizada.

Aqui estão alguns exemplos:

```
1 + 2      // => 3: adição
"1" + "2"  // => "12": concatenação
"1" + 2    // => "12": concatenação após número a-
```

```
sequência
1 + {}           // => "1[object Object]": concatenação após
objeto para string
verdadeiro + verdadeiro // => 2: adição após booleano para número
2 + nulo         // => 2: adição após nulo é convertida em 0
2 + indefinido // => NaN: adição após indefinido converte para
NaN
```

Por fim, é importante observar que quando o operador + é utilizado com strings e números, ele pode não ser associativo. Ou seja, o resultado pode depender da ordem em que as operações são realizadas.

Por exemplo:

```
1 + 2 + "ratos cegos" // => "3 ratos cegos";
1 + (2 + "ratos"      // => "12 ratos cegos";
      "cegos")
```

A primeira linha não possui parênteses e o operador + possui associatividade da esquerda para a direita, portanto os dois números são somados primeiro e sua soma é concatenada com a string. Na segunda linha, os parênteses alteram esta ordem de operações: o número 2 é concatenado com a string para produzir uma nova string. Então o número 1 é concatenado com a nova string para produzir o resultado final.

4.8.2 Operadores Aritméticos Unários

Os operadores unários modificam o valor de um único operando para produzir um novo valor. Em JavaScript, todos os operadores unários têm alta precedência e são associativos à direita. Todos os operadores aritméticos unários descritos nesta seção (+, -, ++ e --) convertem seu operando único em um número, se necessário. Observe que os caracteres de pontuação + e - são usados como operadores unários e binários.

Os operadores aritméticos unários são os seguintes:

Unário mais (+)

O Operador Unary Plus converte seu operando em um número (ou em NAN) e retornos que convertiam o valor. Quando usado com um operando que já é um número, ele não faz nada. Esse operador não pode ser usado com valores BIGINT, pois eles não podem ser convertidos em números regulares.

Unário menos (-)

Quando - é usado como um operador unário, ele converte seu operando em um número, se necessário e depois altera o sinal do resultado.

Incremento (++)

O operador ++ incrementos (isto é, adiciona 1 a) seu único operando, que deve ser um LValue (uma variável, um elemento de uma matriz ou uma propriedade de um objeto). O operador converte seu operando em um número, adiciona 1 a esse número e atribui o valor incrementado de volta à variável, elemento ou propriedade.

O valor de retorno do operador ++ depende de sua posição em relação ao operando. Quando usado antes do operando, onde é conhecido como operador de pré-incremento, incrementa o operando e avalia o valor incrementado desse operando. Quando usado após o operando, onde é conhecido como operador pós-incremento, ele incrementa seu operando, mas avalia o valor não conferido desse operando. Considere a diferença entre essas duas linhas de código:

```
Seja i = 1, j = ++ i;      // eu e j são 2
```

```
Seja n = 1, m = n ++;    // n é 2, m é 1
```

Observe que a expressão `x ++` nem sempre é a mesma que `x = x+1`. O operador `++` nunca executa concatenação de string: sempre converte seu operando em um número e o incrementa. Se `x` for a string "1", `++ x` é o número 2, mas `x+1` é a string "11".

Observe também que, devido à inserção automática de semicolon do JavaScript, você não pode inserir uma quebra de linha entre o operador pós-incremento e o operando que o precede. Se você o fizer, o JavaScript tratará o operando como uma declaração completa por si só e insira um semicolon antes dela.

Esse operador, tanto nas formas de pré e pós-incremento, é mais comumente usado para incrementar um contador que controla um loop for (§5.4.3).

Decremento (-)

O operador espera um operando LValue. Ele converte o valor do operando em um número, subtrai 1 e atribui o valor decrementado de volta ao operando. Como o operador `++`, o valor de retorno de `-` depende de sua posição em relação ao operando. Quando usado antes do operando, ele diminui e retorna o valor decrementado. Quando usado após o operando, ele diminui o operando, mas retorna o valor indesejado. Quando usado após o seu operando, nenhuma quebra de linha é permitida entre o operando e o operador.

4.8.3 Operadores bitwise

Os operadores bitwise realizam manipulação de baixo nível dos bits na representação binária dos números. Embora não executem operações aritméticas tradicionais, eles são categorizados como operadores aritméticos aqui porque operam em operandos numéricos e retornam um valor numérico. Quatro desses operadores executam álgebra booleana nos pedaços individuais dos operandos, comportando -se como se cada bit em cada operando

eram um valor booleano (1 = verdadeiro, 0 = falso). Os outros três operadores bit a bit são usados para deslocar bits para a esquerda e para a direita. Esses operadores não são comumente usados em programação JavaScript e, se você não estiver familiarizado com a representação binária de inteiros, incluindo a representação em complemento de dois de inteiros negativos, provavelmente poderá pular esta seção.

Os operadores bit a bit esperam operandos inteiros e se comportam como se esses valores fossem representados como inteiros de 32 bits em vez de valores de ponto flutuante de 64 bits. Esses operadores convertem seus operandos em números, se necessário, e então forçam os valores numéricos a números inteiros de 32 bits, eliminando qualquer parte fracionária e quaisquer bits além do 32º. Os operadores de deslocamento exigem um operando do lado direito entre 0 e 31. Depois de converter esse operando em um número inteiro não assinado de 32 bits, eles eliminam quaisquer bits além do 5º, o que produz um número no intervalo apropriado. Surpreendentemente, NaN, Infinity e -Infinity são convertidos em 0 quando usados como operandos desses operadores bit a bit.

Todos esses operadores bit a bit, exceto `&gt*&gt`, podem ser usados com operandos numéricos regulares ou com operandos BigInt (ver §3.2.5).

E bit a bit (&)

O operador `&` executa uma operação booleana AND em cada bit de seus argumentos inteiros. Um bit é definido no resultado somente se o bit correspondente estiver definido em ambos os operandos. Por exemplo, 0x1234 e 0x00FF são avaliados como 0x0034.

OU bit a bit ()

O `|` operador executa uma operação booleana OR em cada bit de seu

argumentos inteiros. Um bit está definido no resultado se o bit correspondente for definido em um ou ambos os operandos. Por exemplo, $0x1234 \mid 0x00FF$ avalia para $0x12ff$.

Bitwise xor (^)

O operador \wedge executa um exclusivo ou operação booleana em cada bit de seus argumentos inteiros. Exclusivo ou significa que o operando um é verdadeiro ou o operando dois é verdadeiro, mas não ambos. Um bit está definido no resultado desta operação se um bit correspondente for definido em um (mas não ambos) dos dois operandos. Por exemplo, $0xff00 \wedge 0xf0f0$ avalia para $0x0ff0$.

Bitwise não (~)

O operador \sim é um operador unário que aparece antes do seu único operando inteiro. Opera revertendo todos os bits no operando. Devido à maneira como os números inteiros assinados são representados em JavaScript, a aplicação do operador \sim a um valor é equivalente a alterar seu sinal e subtrair 1. Por exemplo, $\sim 0x0f$ avalia para $0xffffffff0$, ou -16.

*Mudança para a esquerda
(<<)*

O operador $<<$ move todos os bits em seu primeiro operando para a esquerda pelo número de lugares especificados no segundo operando, que deve ser um número inteiro entre 0 e 31. Por exemplo, na operação $A << 1$, o primeiro bit (os que mordem) de A se tornam o segundo bit (o bit dois), o segundo bit de A se torna o terceiro, etc. Um zero é usado para o novo primeiro bit e o valor do 32º bit é perdido. A mudança de um valor deixada em uma posição é equivalente a multiplicar por 2, a mudança de duas posições é equivalente à multiplicação por 4 e assim por diante. Por exemplo, $7 << 2$ avalia para 28.

Mudar bem com o sinal (>>)

O operador `>>` move todos os bits em seu primeiro operando para a direita pelo número de casas especificado no segundo operando (um número inteiro entre 0 e 31). Os bits deslocados para a direita são perdidos. Os bits preenchidos à esquerda dependem do bit de sinal do operando original, para preservar o sinal do resultado. Se o primeiro operando for positivo, o resultado terá zeros colocados nos bits altos; se o primeiro operando for negativo, o resultado terá uns colocados nos bits altos. Deslocar um valor positivo uma casa para a direita equivale a dividir por 2 (descartando o resto), deslocar duas casas para a direita equivale a uma divisão inteira por 4 e assim por diante. `7 >> 1` é avaliado como 3, por exemplo, mas observe que `-7 >> 1` é avaliado como -4.

Shift para a direita com preenchimento zero (>>>)

O operador `>>>` é igual ao operador `>>`, exceto que os bits deslocados à esquerda são sempre zero, independentemente do sinal do primeiro operando. Isso é útil quando você deseja tratar valores assinados de 32 bits como se fossem números inteiros não assinados. `-1 >>> 4` é avaliado como -1, mas `-1 >>> 4` é avaliado como `0x0FFFFFFF`, por exemplo. Este é o único operador bit a bit do JavaScript que não pode ser usado com valores BigInt. BigInt não representa números negativos definindo o bit alto da mesma forma que os números inteiros de 32 bits, e este operador só faz sentido para aquele complemento de dois específico representação.

4.9 Expressões Relacionais

Esta seção descreve os operadores relacionais do JavaScript. Esses operadores testam um relacionamento (como “igual”, “menor que” ou “propriedade de”) entre dois valores e retornam verdadeiro ou falso, dependendo se esse relacionamento existe. Expressões relacionais sempre são avaliadas como um valor booleano, e esse valor é frequentemente usado para controlar o fluxo de execução do programa em if, while e for.

Declarações (consulte o Capítulo 5). As subseções a seguir documentam os operadores de igualdade e desigualdade, os operadores de comparação e os outros dois operadores relacionais do JavaScript, em e na instância.

4.9.1 Operadores de igualdade e desigualdade

Os operadores `==` e `====` verificam se dois valores são iguais, usando duas definições diferentes de mesmice. Ambos os operadores aceitam operandos de qualquer tipo, e ambos retornam se seus operandos forem iguais e falsos se forem diferentes. O operador `====` é conhecido como operador estrito de igualdade (ou às vezes o operador de identidade) e verifica se seus dois operandos são “idênticos” usando uma definição estrita de mesmice. O operador `==` é conhecido como operador de igualdade; Ele verifica se seus dois operandos são “iguais” usando uma definição mais relaxada de mesmice que permite conversões de tipo.

O! = E! == Operadores testam exatamente o oposto dos operadores `==` e `====`. O! = O operador de desigualdade retorna `false` se dois valores forem iguais entre si de acordo com `==` e retorna `true` de outra forma. O! == O operador retorna `false` se dois valores forem estritamente iguais e retornam verdadeiro. Como você verá em §4.10, o! O operador calcula a operação booleana e não. Isso facilita a lembrança disso! = E! == significa “não igual a” e “não estritamente igual a”;

O `=`, `==` e `====` operadores

JavaScript suporta `=`, `==` e `====` operadores. Certifique -se de entender as diferenças entre essas atribuições, igualdade e operadores estritas de igualdade e tenha cuidado para usar o correto ao codificar! Embora seja tentador ler os três operadores como “iguais”, pode ajudar a reduzir a confusão se você

leia “obtém” ou “é atribuído” para `=`, “é igual a” para `==` e “é estritamente igual a” para `===`.

O operador `==` é um recurso legado do JavaScript e é amplamente considerado uma fonte de bugs. Você quase sempre deve usar `===` em vez de `==` e `!=` em vez de `!=`.

Conforme mencionado na Seção 3.8, os objetos JavaScript são comparados por referência, não por valor. Um objeto é igual a si mesmo, mas não a qualquer outro objeto. Se dois objetos distintos tiverem o mesmo número de propriedades, com os mesmos nomes e valores, eles ainda não serão iguais. Da mesma forma, duas matrizes que possuem os mesmos elementos na mesma ordem não são iguais entre si.

IGUALDADE ESTRITA

O operador de igualdade estrita `===` avalia seus operandos e, em seguida, compara os dois valores da seguinte forma, sem realizar nenhuma conversão de tipo:

- Se os dois valores tiverem tipos diferentes, eles não serão iguais.
- Se ambos os valores forem nulos ou ambos os valores forem indefinidos, eles serão iguais.
- Se ambos os valores forem o valor booleano verdadeiro ou ambos forem o valor booleano falso, eles serão iguais.
- Se um ou ambos os valores forem `Nan`, eles não serão iguais. (Isso é surpreendente, mas o valor `Nan` nunca é igual a qualquer outro valor, incluindo ele mesmo! Para verificar se um valor `x` é `Nan`, use `x !== x`, ou a função global `isNan()`.)
- Se ambos os valores forem números e tiverem o mesmo valor, eles serão iguais. Se um valor for `0` e o outro for `-0`, eles também serão iguais.
- Se ambos os valores forem strings e contiverem exatamente os mesmos valores de 16 bits (veja a barra lateral em §3.3) nas mesmas posições, eles serão iguais. Se as strings diferirem em comprimento ou conteúdo, elas não serão iguais.

igual. Duas strings podem ter o mesmo significado e a mesma aparência visual, mas ainda assim serem codificadas usando sequências diferentes de valores de 16 bits. JavaScript não realiza normalização Unicode e um par de strings como esse não é considerado igual aos operadores === ou ==.

- Se ambos os valores se referirem ao mesmo objeto, matriz ou função, eles serão iguais. Se se referirem a objetos diferentes, não serão iguais, mesmo que ambos os objetos tenham propriedades idênticas.

IGUALDADE COM CONVERSÃO DE TIPO

O operador de igualdade == é como o operador de igualdade estrita, mas é menos estrito. Se os valores dos dois operandos não forem do mesmo tipo, ele tenta algumas conversões de tipo e tenta a comparação novamente:

- Se os dois valores tiverem o mesmo tipo, teste-os quanto à igualdade estrita conforme descrito anteriormente. Se forem estritamente iguais, são iguais. Se não forem estritamente iguais, não são iguais.
- Se os dois valores não tiverem o mesmo tipo, o operador == ainda poderá considerá-los iguais. Ele usa as seguintes regras e conversões de tipo para verificar a igualdade:
 - Se um valor for nulo e o outro for indefinido, eles serão iguais.
 - Se um valor for um número e o outro for uma string, converta a string em um número e tente a comparação novamente, usando o valor convertido.
 - Se algum dos valores for verdadeiro, converta-o para 1 e tente a comparação novamente. Se algum dos valores for falso, converta-o para 0 e tente a comparação novamente.
 - Se um valor for um objeto e o outro for um número ou string, converta o objeto em um primitivo usando o comando

algoritmo descrito em §3.9.3 e tente a comparação novamente. Um objeto é convertido em um valor primitivo pelo método `toString()` ou pelo método `valueOf()`. As classes integradas do JavaScript principal tentam a conversão de `valueOf()` antes

conversão `toString()`, exceto para a classe `Date`, que realiza a conversão `toString()`.

- Quaisquer outras combinações de valores não são iguais.

Como exemplo de teste de igualdade, considere a comparação:

```
'1' == // => verdadeiro  
verdadeiro
```

Esta expressão é avaliada como verdadeira, indicando que esses valores de aparência muito diferente são de fato iguais. O valor booleano `true` é primeiro convertido para o número `1` e a comparação é feita novamente. Em seguida, a string “`1`” é convertida no número `1`. Como ambos os valores agora são iguais, a comparação retorna `verdadeiro`.

4.9.2 Operadores de Comparação

Os operadores de comparação testam a ordem relativa (numérica ou alfabética) de seus dois operandos:

Menos que (<)

O operador `<` é avaliado como verdadeiro se seu primeiro operando for menor que seu segundo operando; caso contrário, será avaliado como falso.

Maior que (>)

O operador `>` é avaliado como verdadeiro se seu primeiro operando for maior que seu segundo operando; caso contrário, será avaliado como falso.

Menor ou igual (<=)

O operador <= avalia como verdadeiro se seu primeiro operando for menor ou igual ao seu segundo operando; Caso contrário, ele avalia como falso.

Maior ou igual (>=)

O operador >= avalia como true se seu primeiro operando for maior ou igual ao seu segundo operando; Caso contrário, ele avalia como falso.

Os operandos desses operadores de comparação podem ser de qualquer tipo. A comparação pode ser realizada apenas em números e strings, no entanto, os operando que não são números ou strings são convertidos.

Comparação e conversão ocorrem da seguinte forma:

- Se um operando avaliar para um objeto, esse objeto será convertido em um valor primitivo, conforme descrito no final do §3.9.3; Se o método ValueOf () retornar um valor primitivo, esse valor será usado. Caso contrário, o valor de retorno do seu método ToString () é usado.
- Se, após qualquer conversão de objeto para primitiva necessária, ambos os operandos são strings, as duas cordas serão comparadas, usando ordem alfabética, onde “ordem alfabética” é definida pela ordem numérica dos valores de unicode de 16 bits que compõem as seqüências.
 -
- Se, após a conversão de objeto para primitivo, pelo menos um operando não for uma string, ambos os operando são convertidos em números e comparados numericamente. 0 e -0 são considerados iguais. O infinito é maior do que qualquer número que não seja ele mesmo e - o infinito é menor que qualquer número que não seja. Se um operando for (ou converter para) nan, então a comparação

O operador sempre retorna falso. Embora os operadores aritméticos não permitam que os valores do BIGINT sejam misturados com números regulares, os operadores de comparação permitem comparações entre números e bigints.

Lembre-se de que as cadeias de javascript são seqüências de valores inteiros de 16 bits e que a comparação de string é apenas uma comparação numérica dos valores nas duas seqüências. A ordem de codificação numérica definida pelo Unicode não pode corresponder à ordem de agrupamento tradicional usada em qualquer idioma ou local específico. Observe, em particular, que a comparação de string é sensível ao minúsculas e todas as letras ASCII de capital são "menos que" todas as letras ASCII minúsculas. Esta regra pode causar resultados confusos se você não esperar. Por exemplo, de acordo com o < operador, a string "Zoo" vem antes da string "Aardvark".

Para um algoritmo mais robusto de comparação de string-comparação, experimente o método String.LocaleCompare (), que também leva em consideração definições específicas de localidade de ordem alfabética. Para comparações insensíveis ao caso, você pode converter as seqüências para todas as minúsculas ou toda a mancha usando String.TolowerCase () ou

String.Touppercase (). E, para uma ferramenta de comparação de string mais geral e melhor localizada, use a classe Intl.Collator descrita no §11.7.3.

Tanto o operador + quanto os operadores de comparação se comportam de maneira diferente

para operando numérico e string. + Favorias de strings: ele executa concatenação se um operando for uma string. Os operadores de comparação favorecem os números e executam apenas a comparação de strings se ambos os operandos forem strings:

```

1 + 2          // => 3: adição.
"1" + "2"      // => "12": concatenação.
"1" + 2        // => "12": 2 é convertido em "2".
11 < 3         // => falso: comparação numérica.
"11" < "3"      // => true: comparação de strings.
"11" < 3        // => false: comparação numérica, "11" é convertido
para 11.
&quot;um&quot; &lt;3 // => false: comparação numérica, "um" é convertido
para NaN.

```

Por fim, observe que os operadores `<=` (menor ou igual) e `>=` (maior que ou igual) não dependem dos operadores de igualdade ou de igualdade estrita para determinar se dois valores são “iguais”. Em vez disso, o operador menor ou igual é simplesmente definido como “não maior que” e o operador maior ou igual é definido como “não menor que”. A única exceção ocorre quando um dos operandos é (ou é convertido para) NaN; nesse caso, todos os quatro operadores de comparação retornam falso.

4.9.3 O operador in

O operador `in` espera um operando do lado esquerdo que seja uma string, símbolo ou valor que possa ser convertido em uma string. Espera um operando do lado direito que é um objeto. Ele é avaliado como verdadeiro se o valor do lado esquerdo for o nome de uma propriedade do objeto do lado direito. Por exemplo:

```

deixe ponto = {x: 1, y: 1};    //Definir um objeto
&quot;x&quot; in ponto        // => true: objeto possui propriedade
chamado "x"
&quot;z&quot; in ponto        // => falso: objeto não possui "z";
propriedade.
&quot;toString&quot; in ponto // => verdadeiro: o objeto herda
Método toString

deixe dados = [7,8,9];        // Um array com elementos
(índices) 0, 1 e 2
&quot;0&quot; in dados         // => true: array possui um elemento
"0"

```

```
1 em dados          // => true: os números são convertidos
para as cordas
3 em dados          // => false: no element 3
```

4.9.4 A instância do operador

A instância do operador espera um operando do lado esquerdo que é um objeto e um operando do lado direito que identifica uma classe de objetos. O operador avalia como true se o objeto do lado esquerdo for uma instância da classe do lado direito e avaliará como false caso contrário. O capítulo 9 explica que, no JavaScript, as classes de objetos são definidas pela função do construtor que os inicializa. Assim, o operando do lado direito do Instância de deve ser uma função. Aqui estão exemplos:

```
Seja d = new Date ();    // Crie um novo objeto com a data ()
Construtor D
Instância de data      // => true: D foi criado com Date ()
D instância do objeto  // => true: Todos os objetos são instâncias de
Objeto D Instância
de número              // => false: D não é um objeto numérico
Seja a = [1, 2, 3];    // Crie uma matriz com matriz literal
sintaxe uma
instância de matriz    // => true: a é uma matriz
uma instância de objeto // => true: todas as matrizes são objetos
uma instância de regexp // => false: as matrizes não são regulares
expressões
```

Observe que todos os objetos são instâncias de objeto. A instância considera as "superclasses" ao decidir se um objeto é uma instância de uma classe. Se o operando do lado esquerdo do Instância de não for um objeto, a instância do retornará falsa. Se o lado do destro não for uma classe de objetos, ele lança um TypeError.

Para entender como a instância do operador funciona, você deve entender a "cadeia de protótipos". Esta é a herança de JavaScript

mecanismo, e é descrito em §6.3.2. Para avaliar a expressão `o instanceof f`, o JavaScript avalia `f.prototype` e então procura esse valor na cadeia de protótipos de `o`. Se encontrar, então `o` é uma instância de `f` (ou de uma subclasse de `f`) e o operador retorna verdadeiro. Se `f.prototype` não for um dos valores na cadeia de protótipos de `o`, então `o` não é uma instância de `f` e `instanceof` retorna falso.

4.10 Expressões Lógicas

Os operadores lógicos `&&` e `||` executam álgebra booleana e são frequentemente usados em conjunto com os operadores relacionais para combinar duas expressões relacionais em uma expressão mais complexa. Esses operadores são descritos nas subseções a seguir. Para compreendê-los completamente, você pode revisar o conceito de valores “verdadeiros” e “falsos” introduzido na Seção 3.4.

4.10.1 E lógico (`&&`)

O operador `&&` pode ser entendido em três níveis diferentes. No nível mais simples, quando usado com operandos booleanos, `&&` executa a operação booleana AND nos dois valores: ele retorna verdadeiro se e somente se tanto seu primeiro operando quanto seu segundo operando forem verdadeiros. Se um ou ambos os operandos forem falsos, ele retornará falso.

`&&` é frequentemente usado como uma conjunção para unir duas expressões relacionais:

```
x === 0 && y === 0 // verdadeiro se, e somente se, x e y forem  
ambos 0
```

Expressões relacionais sempre são avaliadas como verdadeiras ou falsas, portanto, quando usadas dessa forma, o próprio operador `&&` retorna verdadeiro ou falso. Os operadores relacionais têm precedência mais alta que `&&`; (`e ||`), portanto, expressões como essas podem ser escritas com segurança sem parênteses.

Mas `&&` não exige que seus operandos sejam valores booleanos. Lembre-se de que todos os valores JavaScript são “verdadeiros” ou “falsos”. (Veja §3.4 para detalhes. Os valores falsos são falsos, nulos, indefinidos, 0, -0, NaN e `" "`. Todos os outros valores, incluindo todos os objetos, são verdadeiros.) O segundo nível no qual `&&` pode ser entendido é como um operador booleano AND para valores verdadeiros e falsos. Se ambos os operandos forem verdadeiros, o operador retornará um valor verdadeiro. Caso contrário, um ou ambos os operandos deverão ser falsos e o operador retornará um valor falso. Em JavaScript, qualquer expressão ou instrução que espera um valor booleano funcionará com um valor verdadeiro ou falso, portanto o fato de `&&`; nem sempre retornar verdadeiro ou falso não causa problemas práticos.

Observe que esta descrição diz que o operador retorna “um valor verdadeiro” ou “um valor falso”, mas não especifica qual é esse valor. Para isso, precisamos descrever `&&`; no terceiro e último nível. Este operador começa avaliando seu primeiro operando, a expressão à sua esquerda. Se o valor à esquerda for falso, o valor de toda a expressão também deverá ser falso, então `&&`; simplesmente retorna o valor à esquerda e nem mesmo avalia a expressão à direita.

Por outro lado, se o valor à esquerda for verdadeiro, então o valor global da expressão depende do valor do lado direito. Se o valor à direita for verdadeiro, então o valor geral deve ser verdadeiro,

e se o valor à direita for falso, então o valor geral deverá ser falso. Portanto, quando o valor à esquerda é verdadeiro, o operador `&&` avalia e retorna o valor à direita:

```
let o = {x:  
1}; let p =  
null; o && o //=> 1: o é verdadeiro, então retorna o valor de o  
p && px //=> null: p é falso, então retorne e não  
avaliar px
```

É importante entender que `&&` pode ou não avaliar seu operando do lado direito. Neste exemplo de código, a variável `p` é definida como nula e a expressão `px`, se avaliada, causaria um `TypeError`. Mas o código usa `&&` de forma idiomática para que `px` seja avaliado apenas se `p` for verdadeiro – não nulo ou indefinido.

O comportamento de `&&` às vezes é chamado de curto-circuito, e às vezes você pode ver código que explora propositalmente esse comportamento para executar código condicionalmente. Por exemplo, as duas linhas de código JavaScript a seguir têm efeitos equivalentes:

```
if (a === b) parar();      // Invoca stop() somente se a === b  
(a === b) &&           //Isso faz a mesma coisa  
parar();
```

Em geral, você deve ter cuidado ao escrever uma expressão com efeitos colaterais (atribuições, incrementos, decrementos ou invocações de função) no lado direito de `&&`. A ocorrência desses efeitos colaterais depende do valor do lado esquerdo.

Apesar da maneira um tanto complexa como esse operador realmente funciona, ele é mais comumente usado como um operador simples de álgebra booleana que

trabalha com valores verdadeiros e falsos.

4.10.2 OU lógico (||)

O `||` operador executa a operação booleana OR em seus dois operandos. Se um ou ambos os operandos forem verdadeiros, ele retornará um valor verdadeiro. Se ambos os operandos forem falsos, ele retornará um valor falso.

Embora o `||` é mais frequentemente usado simplesmente como um operador booleano OR; ele, como o operador `&&`, tem um comportamento mais complexo. Começa avaliando seu primeiro operando, a expressão à sua esquerda. Se o valor deste primeiro operando for verdadeiro, ele entra em curto-circuito e retorna esse valor verdadeiro sem nunca avaliar a expressão à direita. Se, por outro lado, o valor do primeiro operando for falso, então `||` avalia seu segundo operando e retorna o valor dessa expressão.

Tal como acontece com o operador `&&`, você deve evitar operandos do lado direito que incluem efeitos colaterais, a menos que queira usar propositalmente o fato de que a expressão do lado direito pode não ser avaliada.

Um uso idiomático deste operador é selecionar o primeiro valor verdadeiro em um conjunto de alternativas:

```
// Se maxWidth for verdadeiro, use isso. Caso contrário,  
procure um valor em  
// o objeto de preferências. Se isso não for verdade, use  
uma constante codificada.  
deixe max = maxWidth || preferencias.maxWidth || 500;
```

Observe que se 0 for um valor válido para `maxWidth`, esse código não funcionará corretamente, pois 0 é um valor falso. Veja o ?? operador (§4.13.2)

para uma alternativa.

Antes do ES6, esse idioma é frequentemente usado em funções para fornecer valores padrão para parâmetros:

```
// Copie as propriedades de o a P, e retorne P cópia  
da função (o, p) {p = p || {}; // Se nenhum objeto  
foi aprovado para P, use um objeto recém -criado.  
  
// O corpo da função vai aqui}
```

No ES6 e posterior, no entanto, esse truque não é mais necessário porque o valor do parâmetro padrão pode ser simplesmente gravado na própria definição da função: cópia da função (o, p = {}) {...}.

4.10.3 Lógico não (!)

O ! O operador é um operador unário; É colocado antes de um único operando. Seu objetivo é inverter o valor booleano de seu operando. Por exemplo, se x é verdade ,! X avalia como falso. Se x é falsamente, então! X é verdadeiro.

Ao contrário do && e || operadores, o ! O operador converte seu operando em um valor booleano (usando as regras descritas no capítulo 3) antes de inverter o valor convertido. Isso significa isso! Sempre retorna verdadeiro ou falso e que você pode converter qualquer valor x em seu valor booleano equivalente aplicando este operador duas vezes: !! x (consulte §3.9.2).

Como um operador unário ,! tem alta precedência e se liga firmemente. Se você deseja inverter o valor de uma expressão como P && q, precisará usar parênteses :! (P && q). Vale a pena notar duas leis de booleanos

álgebra aqui que podemos expressar usando a sintaxe JavaScript:

```
// Leis de DeMorgan !(p  
&& q) === (!p || !q) // => true: para todos os valores de p e  
q !(p || q) === (!p  
&& !q) // => true: para todos os valores de p e  
q
```

4.11 Expressões de Atribuição

JavaScript usa o operador = para atribuir um valor a uma variável ou propriedade. Por exemplo:

```
i = 0;      // Defina a variável i como 0.  
o.x = 1;    // Defina a propriedade x do objeto o como 1.
```

O operador = espera que seu operando do lado esquerdo seja um lvalue: uma variável ou propriedade de objeto (ou elemento de array). Ele espera que seu operando do lado direito seja um valor arbitrário de qualquer tipo. O valor de uma expressão de atribuição é o valor do operando do lado direito. Como efeito colateral, o operador = atribui o valor à direita à variável ou propriedade à esquerda para que futuras referências à variável ou propriedade avaliem o valor.

Embora as expressões de atribuição sejam geralmente bastante simples, às vezes você poderá ver o valor de uma expressão de atribuição usada como parte de uma expressão maior. Por exemplo, você pode atribuir e testar um valor na mesma expressão com um código como este:

```
(uma = b) === 0
```

Se você fizer isso, certifique-se de ter clareza sobre a diferença entre =

e === operadores! Observe que = tem precedência muito baixa e geralmente são necessários parênteses quando o valor de uma atribuição deve ser usado em uma expressão maior.

O operador de atribuição possui associatividade da direita para a esquerda, o que significa que quando vários operadores de atribuição aparecem em uma expressão, eles são avaliados da direita para a esquerda. Assim, você pode escrever um código como este para atribuir um único valor a múltiplas variáveis:

```
eu = j = k = 0;           // Inicializa 3 variáveis com 0
```

4.11.1 Atribuição com Operação

Além do operador de atribuição normal =, o JavaScript suporta vários outros operadores de atribuição que fornecem atalhos combinando a atribuição com alguma outra operação. Por exemplo, o operador += realiza adição e atribuição. A seguinte expressão:

```
total += imposto sobre vendas;
```

é equivalente a este:

```
total = total + imposto sobre vendas;
```

Como seria de esperar, o operador += funciona para números ou strings. Para operandos numéricos, realiza adição e atribuição; para operandos string, ele realiza concatenação e atribuição.

Operadores semelhantes incluem -=, *=, &#= e assim por diante. A Tabela 4-2 lista todos eles.

Tabela 4-2. Operadores de atribuição

Operador	Exemplo	Equivalente
<code>+= uma + = b</code>		<code>uma = uma + b</code>
<code>-= uma - = b</code>		<code>uma = uma - b</code>
<code>*= uma * = b</code>		<code>uma = uma * b</code>
<code>/= uma /= b</code>		<code>uma = uma/b</code>
<code>%= uma % = b</code>		<code>uma = uma % b</code>
<code>**=</code>	<code>uma **= b</code>	<code>uma = uma ** b</code>
<code><<=</code>	<code>uma <<= b</code>	<code>uma = uma << b</code>
<code>>>=</code>	<code>uma >>= b</code>	<code>uma = uma >> b</code>
<code>>>>=</code>	<code>uma >>>= b</code>	<code>uma = uma >>> b</code>
<code>&= uma &= b</code>		<code>uma = uma & b</code>
<code> = uma = b</code>		<code>uma = uma b</code>
<code>^= uma ^ = b</code>		<code>uma = uma ^ b</code>

Na maioria dos casos, a expressão:

```
uma sobre = b
```

onde op é um operador, é equivalente à expressão:

```
uma = uma para b
```

Na primeira linha, a expressão a é avaliada uma vez. Na segunda, é avaliado duas vezes. Os dois casos serão diferentes apenas se a incluir o lado

efeitos como uma chamada de função ou um operador de incremento. As duas atribuições a seguir, por exemplo, não são iguais:

```
dados[i++] *= 2; dados[i++]  
= dados[i++] * 2;
```

4.12 Expressões de Avaliação

Como muitas linguagens interpretadas, o JavaScript tem a capacidade de interpretar strings de código-fonte JavaScript, avaliando-as para produzir um valor. JavaScript faz isso com a função global eval():

```
avaliação("3 // => 5  
+2")
```

A avaliação dinâmica de strings de código-fonte é um recurso de linguagem poderoso que quase nunca é necessário na prática. Se você estiver usando eval(), pense cuidadosamente se realmente precisa usá-lo. Em particular, eval() pode ser uma falha de segurança e você nunca deve passar qualquer string derivada da entrada do usuário para eval(). Com uma linguagem tão complicada como JavaScript, não há como limpar a entrada do usuário para torná-la segura para uso com eval(). Devido a esses problemas de segurança, alguns servidores web usam o cabeçalho HTTP “Content-Security-Policy” para desabilitar eval() para um site inteiro.

As subseções a seguir explicam o uso básico de eval() e explicam duas versões restritas dele que têm menos impacto no otimizador.

EVAL() É UMA FUNÇÃO OU UM OPERADOR?

`eval()` é uma função, mas está incluída neste capítulo sobre expressões porque realmente deveria ser um operador. As primeiras versões da linguagem definiam uma função `eval()` e, desde então, os designers de linguagem e escritores de intérpretes têm colocado restrições a ela que a tornam cada vez mais semelhante a um operador. Os intérpretes JavaScript modernos realizam muitas análises de código e

otimização. De modo geral, se uma função chamar `eval()`, o interpretador não poderá otimizar essa função. O problema de definir `eval()` como uma função é que ela pode receber outros nomes:

```
seja f = avaliação;
seja g = f;
```

Se isso for permitido, o intérprete não poderá saber com certeza quais funções chamam `eval()` e, portanto, não poderá otimizar agressivamente. Este problema poderia ter sido evitado se `eval()` fosse um operador (e uma palavra reservada). Aprenderemos (em §4.12.2 e §4.12.3) sobre as restrições impostas a `eval()` para torná-lo mais parecido com um operador.

4.12.1 avaliação()

`eval()` espera um argumento. Se você passar qualquer valor diferente de uma string, ele simplesmente retornará esse valor. Se você passar uma string, ele tentará analisá-la como código JavaScript, lançando um `SyntaxError` se falhar. Se analisar a string com sucesso, ele avaliará o código e retornará o valor da última expressão ou instrução na string ou

indefinido se a última expressão ou instrução não tiver valor. Se a string avaliada lançar uma exceção, essa exceção se propagará a partir da chamada para `eval()`.

A principal coisa sobre `eval()` (quando invocado desta forma) é que ele usa o ambiente variável do código que o chama. Ou seja, ele procura os valores das variáveis e define novas variáveis e funções da mesma forma que o código local. Se uma função definir uma variável local `x` e então chamar `eval("x")`, ela obterá o valor da variável local. Se chamar `eval("x=1")`, altera o valor da variável local. E se a função chamar `eval("var y = 3")`, ela

declara uma nova variável local y. Por outro lado, se a string avaliada usar let ou const, a variável ou constante declarada será local para a avaliação e não será definida no ambiente de chamada.

Da mesma forma, uma função pode declarar uma função local com código como este:

```
avaliar ("função f () {return x+1;}");
```

Se você ligar para o Código de nível superior, ele opera em variáveis globais e funções globais, é claro.

Observe que a string de código que você passa para avaliar () deve fazer sentido sintático por conta própria: você não pode usá-lo para colar fragmentos de código em uma função. Não faz sentido escrever Eval ("Return"), por exemplo, porque o retorno é apenas legal dentro das funções e o fato de a string avaliada usar o mesmo ambiente variável que a função de chamada não faz parte dessa função. Se sua string faria sentido como um script independente (mesmo muito curto como x = 0), é legal passar para avaliar (). Caso contrário, avaliar () () lançará um sintaxe.

4.12.2 Global Eval ()

É a capacidade de avaliar () alterar variáveis locais que são tão problemáticas para os otimizadores de JavaScript. Como uma solução alternativa, no entanto, os intérpretes simplesmente fazem menos otimização em qualquer função que chama Eval (). Mas o que um intérprete de javascript deve fazer, no entanto, se um script define um alias para avaliar () e depois chama essa função por

outro nome? A especificação JavaScript declara que quando eval() é invocado por qualquer nome diferente de “eval”, ele deve avaliar a string como se fosse um código global de nível superior. O código avaliado pode definir novas variáveis globais ou funções globais, e pode definir variáveis globais, mas não usará ou modificará quaisquer variáveis locais para a função de chamada e, portanto, não interferirá nas otimizações locais.

Uma “avaliação direta” é uma chamada à função eval() com uma expressão que usa o nome exato e não qualificado “eval” (que está começando a parecer uma palavra reservada). Chamadas diretas para eval() usam o ambiente variável do contexto de chamada. Qualquer outra chamada — uma chamada indireta — usa o objeto global como seu ambiente variável e não pode ler, escrever ou definir variáveis ou funções locais. (As chamadas diretas e indiretas podem definir novas variáveis somente com var. Os usos de let e const dentro de uma string avaliada criam variáveis e constantes que são locais para a avaliação e não alteram a chamada ou o ambiente global.)

O código a seguir demonstra:

```
const caso = avaliação; // Usar outro nome faz
uma avaliação global seja x =
&quot;global&quot;, y = &quot;global&quot;; //Duas variáveis globais
função f() { //Esta função faz um
    avaliação local
    deixe x = &quot;local&quot;; //Define uma variável local
    eval(&quot;x += &#39;alterado&#39;&quot;); // Avaliação direta define local
    variável
    retornar x; //Retorna o local alterado
    variável }

função g() { //Esta função faz um
    avaliação global
    deixe y = &quot;local&quot;; // Uma variável local
```

```

Geval (&quot;y += &#39;alterado&#39;;&quot;); // Conjuntos de avaliação indireta
variável global
    retornar y;                                // retorna local inalterado local
variável}

console.log (f (), x); // Variável local alterada: impressa
&quot;LocalChanged Global&quot;;
console.log (g (), y); // Variável global alterada: impressões
&quot;Local GlobalChanged&quot;;

```

Observe que a capacidade de fazer uma avaliação global não é apenas uma acomodação para as necessidades do otimizador; Na verdade, é um recurso tremendamente útil que permite executar strings de código como se fossem scripts independentes de nível superior. Conforme observado no início desta seção, é raro precisar realmente avaliar uma sequência de código. Mas se você achar necessário, é mais provável que você queira fazer uma avaliação global do que uma avaliação local.

4.12.3 Eval rigoroso ()

Modo rigoroso ([consulte o §5.6.3](#)) impõe restrições adicionais ao comportamento da função Eval () e mesmo sobre o uso do identificador "avaliar". Quando avaliar () () é chamado do código de modo rigoroso, ou quando a sequência de código a ser avaliada começa com uma diretiva de "uso rigorosa", o Eval () faz uma avaliação local com um ambiente de variável privada. Isso significa que, no modo rigoroso, o código avaliado pode consultar e definir variáveis locais, mas não pode definir novas variáveis ou funções no escopo local.

Além disso, o modo rigoroso torna o Eval () ainda mais semelhante ao operador, efetivamente transformando "avaliar" em uma palavra reservada. Você não tem permissão para substituir a função Eval () com um novo valor. E você não é

tem permissão para declarar uma variável, função, parâmetro de função ou parâmetro de bloco catch com o nome “eval”.

4.13 Operadores Diversos

JavaScript oferece suporte a vários outros operadores diversos, descritos nas seções a seguir.

4.13.1 O Operador Condicional (?:)

O operador condicional é o único operador ternário (três operandos) em JavaScript e às vezes é chamado de operador ternário. Às vezes, esse operador é escrito ?:, embora não apareça exatamente dessa forma no código. Como este operador possui três operandos, o primeiro vai antes do ?, o segundo vai entre o ? e o :, e o terceiro vai depois do :. É usado assim:

```
x &gt; 0? x: -x      //O valor absoluto de x
```

Os operandos do operador condicional podem ser de qualquer tipo. O primeiro operando é avaliado e interpretado como booleano. Se o valor do primeiro operando for verdadeiro, então o segundo operando é avaliado e seu valor é retornado. Caso contrário, se o primeiro operando for falso, então o terceiro operando é avaliado e seu valor é retornado. Apenas um do segundo e terceiro operandos é avaliado; nunca os dois.

Embora você possa obter resultados semelhantes usando a instrução if (§5.3.1), o operador ?: geralmente fornece um atalho útil. Aqui está um uso típico, que verifica se uma variável está definida (e tem um valor significativo e verdadeiro) e a utiliza em caso afirmativo ou fornece um valor padrão se

não:

```
cumprimentando = "hello" + (nome de usuário? nome de usuário: "lá");
```

Isso é equivalente a, mas mais compacto do que, o seguinte, if declaração:

```
saudação = "olá";
if (nome de usuário) {
    saudação += nome de usuário; } outro {
    saudação += "lá"; }
```

4.13.2 Primeiro definido (??)

O primeiro operador definido ?? Avalia o seu primeiro operando definido: se o operando esquerdo não for nulo e não indefinido, ele retornará esse valor. Caso contrário, ele retorna o valor do operando correto. Como o && e || operadores, ?? é curto-circuito: ele avalia apenas seu segundo operando se o primeiro operando avaliar como nulo ou indefinido. Se a expressão A não tiver efeitos colaterais, então a expressão A ?? B é equivalente a:

```
(a! == null && a! == indefinido)? A: b
```

? é uma alternativa útil a || (§4.10.2) Quando você deseja selecionar o primeiro operando definido em vez do primeiro operando verdadeiro. Embora || é nominalmente um operador lógico, ele também é usado idiomaticamente para selecionar o primeiro operando não-falsy com código como este:

```
// Se MaxWidth for verdade, use isso. Caso contrário,
procure um valor em
// o objeto de preferências. Se isso não for verdade, use um
```

```
constante codificada. deixe max = maxWidth || preferências.maxWidth ||  
500;
```

O problema com esse uso idiomático é que zero, a string vazia e false são todos valores falsos que podem ser perfeitamente válidos em algumas circunstâncias. Neste exemplo de código, se maxWidth for zero, esse valor será ignorado. Mas se mudarmos o || operador para ??, terminamos com uma expressão onde zero é um valor válido:

```
// Se maxWidth estiver definido, use-o. Caso contrário,  
procure um valor em  
// o objeto de preferências. Se isso não estiver definido,  
use uma constante codificada.  
deixe max = maxWidth ?? preferências.maxWidth ?? 500;
```

Aqui estão mais exemplos mostrando como ?? funciona quando o primeiro operando é falso. Se esse operando for falso, mas definido, então ?? devolve. Somente quando o primeiro operando é “nulo” (ou seja, nulo ou indefinido) que este operador avalia e retorna o segundo operando:

```
deixe opções = { tempo limite: 0, título: " ", detalhado:  
falso, n: nulo };  
opções.timeout ?? 1000           // => 0: conforme definido no objeto  
opções.título ?? "Untitled" // => "": conforme  
definido no objeto  
opções.verbose ?? verdadeiro // => false: conforme definido no  
opções de objeto.quiet ??  
falso                         // => false: propriedade não é  
opções  
definidas.n ?? 10             // => 10: propriedade é nula
```

Observe que o tempo limite, o título e as expressões detalhadas aqui teriam valores diferentes se usássemos || em vez de ??.

O ?? O operador é semelhante ao && e || Os operadores, mas não têm maior precedência ou menor precedência do que eles. Se você o usar em uma expressão com qualquer um desses operadores, deverá usar parênteses explícitos para especificar qual operação você deseja executar primeiro:

```
(A ?? B) || c      // ?? Primeiro, depois ||
A ?? (B || C)    // || Primeiro, então ??
A ?? b || c      // SyntaxError: Parênteses são necessários
```

O ?? O operador é definido pelo ES2020 e, no início de 2020, é recentemente suportado pelas versões atuais ou beta de todos os principais navegadores. Esse operador é formalmente chamado de operador de “coalescente nulo”, mas evito esse termo porque esse operador seleciona um de seus operandos, mas não os “coalesce” de nenhuma maneira que eu possa ver.

4.13.3 O operador TIPEOF

O TypeOf é um operador unário que é colocado antes do seu único operando, que pode ser de qualquer tipo. Seu valor é uma string que especifica o tipo de operando. A Tabela 4-3 especifica o valor do operador TIPOOF para qualquer valor JavaScript.

Tabela 4-3. Valores retornados pelo operador TypeOf

xTipo de x	
indefinido	"indefinido"
nulo "objeto"	
verdadeiro ou falso	"Booleano"
qualquer número ou nan	"número"

qualquer bigint	"bigint"
qualquer string	"corda"
qualquer símbolo	"símbolo"
qualquer função	"função"
qualquer objeto de não função	"objeto"

Você pode usar o operador TIPEOF em uma expressão como esta:

```
// Se o valor for uma string, embrulhe -o em cotações, caso  
contrário, converte  
(tipoof valor === &quot;string&quot;)? &quot;#39;&quot; &quot; +  
valor +&quot;#39; &quot;: value.toString ()
```

Observe que o tipo de retorna "objeto" se o valor do operando for nulo. Se você deseja distinguir os objetos nulos, precisará testar explicitamente esse valor de caso especial.

Embora as funções JavaScript sejam um tipo de objeto, o operador do tipo de considera as funções para serem suficientemente diferentes para que tenham seu próprio valor de retorno.

Como o tipo de avalia para "objeto" para todos os valores de objeto e matriz que não sejam funções, é útil apenas para distinguir objetos de outros tipos primitivos. Para distinguir uma classe de objeto de outra, você deve usar outras técnicas, como a instância do operador (consulte §4.9.4), o atributo de classe (consulte §14.4.3) ou a propriedade do construtor (consulte §9.2. 2 e §14.3).

4.13.4 O operador de exclusão

`delete` é um operador unário que tenta excluir a propriedade do objeto ou elemento da matriz especificado como seu operando. Assim como os operadores de atribuição, incremento e decremento, `delete` normalmente é usado pelo efeito colateral de exclusão de propriedade e não pelo valor que ele retorna. Alguns exemplos:

```
seja o = { x: 1, y: 2}; // Começa com um objeto delete
ox; // Exclui uma de suas propriedades "x"
em o // => false: a propriedade não existe mais

seja a = [1,2,3];           //Começa com um array
exclua um[2];               //Exclui o último elemento do
matriz
2 em um                     // => false: o elemento 2 do array não
existe mais a.length         // => 3: observe o comprimento do array
não muda, no entanto
```

Observe que uma propriedade ou elemento de matriz excluído não é apenas definido como um valor indefinido. Quando uma propriedade é excluída, a propriedade deixa de existir. A tentativa de ler uma propriedade inexistente retorna indefinido, mas você pode testar a existência real de uma propriedade com o operador `in` (§4.9.3). A exclusão de um elemento do array deixa um “buraco” no array e não altera seu comprimento. A matriz resultante é esparsa (§7.3).

`delete` espera que seu operando seja um lvalue. Se não for um lvalue, o operador não executa nenhuma ação e retorna verdadeiro. Caso contrário, `delete` tenta excluir o lvalue especificado. `delete` retornará verdadeiro se excluir com êxito o lvalue especificado. Porém, nem todas as propriedades podem ser excluídas: propriedades não configuráveis (§14.1) são imunes

De exclusão.

No modo rigoroso, a exclusão aumenta um sintaxe se o seu operando for um identificador não qualificado, como uma variável, função ou parâmetro de função: ele só funciona quando o operando é uma expressão de acesso à propriedade (§4.4). O modo rigoroso também especifica que o Delete levanta um TypeError se solicitado a excluir qualquer propriedade não confundível (ou seja, nãoolável). Fora do modo rigoroso, nenhuma exceção ocorre nesses casos, e excluir simplesmente retorna falsa para indicar que o operando não pôde ser excluído.

Aqui estão alguns exemplos de usos do operador de exclusão:

```
let o = {x: 1, y: 2};
delete o.x;      // excluir uma das propriedades do objeto; retorna
verdadeiro.
tipo de boi;    // A propriedade não existe; retorna
"undefined".
delete o.x;    // excluir uma propriedade inexistente; retorna verdadeiro.
excluir 1;      // Isso não faz sentido, mas apenas retorna
verdadeiro. // não pode excluir uma variável;
Retorna falsa, ou syntaxerror no modo rigoroso.
excluir o; // Propriedade nãoolável: retorna
false ou TypeError no modo rigoroso.

excluir object.prototype;
```

Veremos o operador Excluir novamente em §6.4.

4.13.5 O operador aguardado

Aguardar foi introduzido no ES2017 como uma maneira de tornar a programação assíncrona mais natural em JavaScript. Você precisará ler

Capítulo 13 para entender este operador. Resumidamente, entretanto, await espera um objeto Promise (representando uma computação assíncrona) como seu único operando, e faz com que seu programa se comporte como se estivesse aguardando a conclusão da computação assíncrona (mas faz isso sem realmente bloquear, e não faz isso). impedir que outras operações assíncronas ocorram ao mesmo tempo). O valor do operador await é o valor de cumprimento do objeto Promise. É importante ressaltar que await só é legal em funções que foram declaradas assíncronas com a palavra-chave `async`. Novamente, consulte o Capítulo 13 para obter detalhes completos.

4.13.6 O operador vazio

`void` é um operador unário que aparece antes de seu único operando, que pode ser de qualquer tipo. Este operador é incomum e raramente usado; ele avalia seu operando, descarta o valor e retorna indefinido. Como o valor do operando é descartado, usar o operador `void` só faz sentido se o operando tiver efeitos colaterais.

O operador `void` é tão obscuro que é difícil encontrar um exemplo prático de seu uso. Um caso seria quando você deseja definir uma função que não retorna nada, mas também usa a sintaxe de atalho da função de seta (consulte §8.1.3), onde o corpo da função é uma única expressão que é avaliada e retornada. Se você está avaliando a expressão apenas por seus efeitos colaterais e não deseja retornar seu valor, então a coisa mais simples é usar chaves ao redor do corpo da função. Mas, como alternativa, você também poderia usar o operador `void` neste caso:

```
deixe contador = 0;
```

```
const incremento = () => contador
vazio++; incremento() // => indefinido
contador           // => 1
```

4.13.7 O Operador vírgula (,)

O operador vírgula é um operador binário cujos operandos podem ser de qualquer tipo. Ele avalia seu operando esquerdo, avalia seu operando direito e então retorna o valor do operando direito. Assim, a seguinte linha:

```
i=0, j=1, k=2;
```

avalia como 2 e é basicamente equivalente a:

```
eu = 0; j = 1; k = 2;
```

A expressão à esquerda é sempre avaliada, mas seu valor é descartado, o que significa que só faz sentido usar o operador vírgula quando a expressão à esquerda tem efeitos colaterais. A única situação em que o operador vírgula é comumente usado é com um loop for (§5.4.3) que possui múltiplas variáveis de loop:

```
// A primeira vírgula abaixo faz parte da sintaxe da
instrução let
// A segunda vírgula é o operador vírgula: ela nos permite
escrever 2
// expressões (i++ e j--) em uma instrução (o loop for) que
espera 1.
for(deixe i=0,j=10; i < j; i++,j--) { console.log(i+j);
}
```

4.14 Resumo

Este capítulo abrange uma ampla variedade de tópicos, e há muito material de referência aqui que você pode querer reler no futuro à medida que continua aprendendo JavaScript. Alguns pontos-chave a serem lembrados, no entanto, são estes:

- Expressões são as frases de um programa JavaScript.
- Qualquer expressão pode ser avaliada em um valor JavaScript.
- As expressões também podem ter efeitos colaterais (como atribuição variável), além de produzir um valor.
- Expressões simples, como literais, referências variáveis e acessos de propriedade, podem ser combinados com os operadores para produzir expressões maiores.
- O JavaScript define operadores para aritmética, comparações, lógica booleana, atribuição e manipulação de bits, juntamente com alguns operadores diversos, incluindo o operador condicional ternário.
- O operador JavaScript + é usado para adicionar números e concatenar strings.
- Os operadores lógicos && e || Tenha um comportamento especial de “curto circuito” e às vezes avalia apenas um de seus argumentos. Idiomas JavaScript comuns exigem que você entenda o comportamento especial desses operadores.

Capítulo 5. Declarações

O Capítulo 4 descreveu expressões como frases JavaScript. Por essa analogia, as instruções são sentenças ou comandos JavaScript. Assim como as sentenças em inglês são terminadas e separadas umas das outras por pontos, as instruções em JavaScript são terminadas por ponto e vírgula (§2.6). As expressões são avaliadas para produzir um valor, mas as instruções são executadas para fazer algo acontecer.

Uma forma de “fazer algo acontecer” é avaliar uma expressão que tenha efeitos colaterais. Expressões com efeitos colaterais, como atribuições e invocações de funções, podem ser consideradas instruções isoladas e, quando usadas dessa forma, são conhecidas como instruções de expressão. Uma categoria semelhante de instruções são as declarações que declaram novas variáveis e definem novas funções.

Os programas JavaScript nada mais são do que uma sequência de instruções a serem executadas. Por padrão, o interpretador JavaScript executa estas declarações uma após a outra na ordem em que são escritas. Outra maneira de “fazer algo acontecer” é alterar essa ordem padrão de execução, e o JavaScript tem uma série de instruções ou estruturas de controle que fazem exatamente isso:

Condicionais

Instruções como if e switch que fazem o interpretador JavaScript executar ou pular outras instruções dependendo do valor

de uma expressão

Loops

Declarações como While e para isso executam outras declarações repetidamente

Saltos

Declarações como Break, Return That que fazem com que o intérprete salte para outra parte do programa

As seções a seguir descrevem as várias declarações em JavaScript

e explique sua sintaxe. A Tabela 5-1, no final do capítulo, resume a sintaxe. Um programa JavaScript é simplesmente uma sequência de declarações, separadas uma da outra com semicolons; portanto, depois de familiarizar -se com as declarações de JavaScript, você pode começar a escrever programas JavaScript.

5.1 declarações de expressão

Os tipos mais simples de declarações no JavaScript são expressões que têm efeitos colaterais. Esse tipo de afirmação foi mostrado no Capítulo 4. As declarações de atribuição são uma categoria principal de declarações de expressão. Por exemplo:

```
saudação = "hello" + nome; i *= 3;
```

Os operadores de incremento e decréscimos, ++ e -, estão relacionados a declarações de atribuição. Eles têm o efeito colateral de alterar um valor variável, como se uma tarefa tivesse sido realizada:

```
contador ++;
```

O operador de exclusão tem o importante efeito colateral de excluir uma propriedade de objeto. Assim, quase sempre é usado como uma declaração, e não como parte de uma expressão maior:

```
delete o.x;
```

As chamadas de função são outra importante categoria de declarações de expressão. Por exemplo:

```
console.log (DebugMessage); displayspinner (); // Uma função hipotética para exibir um spinner em um aplicativo da web.
```

Essas chamadas de função são expressões, mas têm efeitos colaterais que afetam o ambiente do host ou o estado do programa e são usados aqui como declarações. Se uma função não tiver efeitos colaterais, não há sentido em chamá-la, a menos que faça parte de uma expressão maior ou uma declaração de atribuição. Por exemplo, você não calculava apenas um cosseno e descartaria o resultado:

```
Math.cos (x);
```

Mas você pode calcular o valor e atribuí-lo a uma variável para uso futuro:

```
cx = math.cos (x);
```

Observe que cada linha de código em cada um desses exemplos é encerrada com um ponto de vírgula.

5.2 Declarações Compostas e Vazias

Assim como o operador vírgula (§4.13.7) combina múltiplas expressões em uma única expressão, um bloco de instruções combina múltiplas instruções em uma única instrução composta. Um bloco de instruções é simplesmente uma sequência de instruções entre chaves. Assim, as linhas a seguir atuam como uma única instrução e podem ser usadas em qualquer lugar onde o JavaScript espera uma única instrução:

```
{  
    x = Matemática.PI; cx = Math.cos(x);  
    console.log("cos(π) = " +  
        cx);  
}
```

Há algumas coisas a serem observadas sobre esse bloco de instruções. Primeiro, não termina com ponto e vírgula. As instruções primitivas dentro do bloco terminam em ponto e vírgula, mas o bloco em si não. Segundo, as linhas dentro do bloco são recuadas em relação às chaves que as envolvem. Isso é opcional, mas torna o código mais fácil de ler e entender.

Assim como as expressões geralmente contêm subexpressões, muitas instruções JavaScript contêm subinstruções. Formalmente, a sintaxe JavaScript geralmente permite uma única subafirmação. Por exemplo, a sintaxe do loop while inclui uma única instrução que serve como corpo do loop. Usando um bloco de instruções, você pode colocar qualquer número de instruções dentro desta única subinstrução permitida.

Uma instrução composta permite usar múltiplas instruções onde a sintaxe JavaScript espera uma única instrução. A declaração vazia é

O oposto: permite que você não inclua declarações onde se espera. A declaração vazia é assim:

```
;
```

O intérprete JavaScript não toma medidas quando executa uma instrução vazia. A declaração vazia é ocasionalmente útil quando você deseja criar um loop que tenha um corpo vazio. Considere o seguinte para o loop (para loops será coberto no §5.4.3):

```
// initialize uma matriz A para (vamos i = 0; i <a.Length; a [i ++] = 0);
```

Nesse loop, todo o trabalho é feito pela expressão `a [i ++] = 0`, e nenhum corpo de loop é necessário. A sintaxe do JavaScript requer uma declaração como um corpo de loop, no entanto, uma declaração vazia - apenas um semicolon nu - é usado.

Observe que a inclusão acidental de um ponto e vírgula após o parêntese correto de um loop for, enquanto o loop, ou se a declaração pode causar erros frustrantes que são difíceis de detectar. Por exemplo, o código a seguir provavelmente não faz o que o autor pretendia:

```
if ((a === 0) || (b === 0));      // opa! Esta linha faz
nada...
o = null;                      // e esta linha é sempre
executado.
```

Quando você usa intencionalmente a declaração vazia, é uma boa ideia comentar seu código de uma maneira que deixa claro que você está fazendo isso de propósito. Por exemplo:

```
for(deixe i = 0; i < a.length; a[i++] = 0) /* vazio */ ;
```

5.3 Condicionais

Instruções condicionais executam ou ignoram outras instruções dependendo do valor de uma expressão especificada. Essas declarações são os pontos de decisão do seu código e também são conhecidas como “ramificações”. Se você imaginar um intérprete JavaScript seguindo um caminho através do seu código, as instruções condicionais são os locais onde o código se ramifica em dois ou mais caminhos e o intérprete deve escolher qual caminho seguir.

As subseções a seguir explicam a condicional básica do JavaScript, a instrução if/else, e também abordam switch, uma instrução de ramificação multidirecional mais complicada.

5.3.1 se

A instrução if é a instrução de controle fundamental que permite ao JavaScript tomar decisões ou, mais precisamente, executar instruções condicionalmente. Esta declaração tem duas formas. O primeiro é:

```
se (expressão)
    declaração
```

Nesta forma, a expressão é avaliada. Se o valor resultante for verdadeiro, a instrução será executada. Se a expressão for falsa, a instrução não será executada. (Veja §3.4 para uma definição de valores verdadeiros e falsos.) Por exemplo:

```
if (nome de usuário == nulo)          // Se o nome de usuário for nulo ou
indefinido,
    nome de usuário = "John"        //defini-lo
    "Doe";
```

Ou da mesma forma:

```
// Se o nome de usuário for nulo, indefinido, falso, 0,  
" ou nan, dê um novo valor  
if (! Nome de usuário) nome de usuário = "John Doe";;
```

Observe que os parênteses em torno da expressão são uma parte necessária da sintaxe para a instrução IF.

A sintaxe do JavaScript requer uma única instrução após a palavra -chave IF e a expressão entre parênteses, mas você pode usar um bloco de instrução para combinar várias instruções em uma. Portanto, a declaração se também pode ser assim:

```
if (! endereço) {  
    endereço = ";; mensagem = "Especifique um  
    endereço de correspondência.;;  
}
```

A segunda forma da instrução IF introduz uma cláusula else que é executada quando a expressão é falsa. Sua sintaxe é:

```
If (expressão)  
instrução1  
outro  
    Declaração2
```

Esta forma da declaração executa a instrução1 se a expressão for verdadeira e executa a instrução2 se a expressão for falsamente. Por exemplo:

```
if (n === 1)  
    console.log ("Você tem 1 nova mensagem."); outro
```

```
console.log(`Você tem ${n} novas mensagens.`);
```

Quando você aninha instruções if com cláusulas else, é necessário algum cuidado para garantir que a cláusula else acompanhe a instrução if apropriada. Considere as seguintes linhas:

```
eu = j = 1;
k = 2; se
(eu === j)
    se (j === k)
        console.log("i é igual a k");
outro
    console.log("i não é igual a           // ERRADO! !
j");
```

Neste exemplo, a instrução if interna forma a única instrução permitida pela sintaxe da instrução if externa. Infelizmente, não está claro (exceto pela dica dada pelo recuo) qual é o else. E neste exemplo, o recuo está errado, porque um interpretador JavaScript na verdade interpreta o exemplo anterior como:

```
se (eu === j) {
    se (j === k)
        console.log("i é igual a k");
    outro
        console.log("i não é igual a           //OPS!
j");
}
```

A regra em JavaScript (como na maioria das linguagens de programação) é que, por padrão, uma cláusula else faz parte da instrução if mais próxima. Para tornar este exemplo menos ambíguo e mais fácil de ler, entender, manter e depurar, você deve usar chaves:

```
se (eu === j) {  
    se (j === k) {  
        console.log("i é igual a k"); } } else  
{ // Que diferença faz a localização de uma  
chave!  
  
    console.log("i não é igual a j"); }
```

Muitos programadores têm o hábito de colocar os corpos das instruções if e else (bem como de outras instruções compostas, como loops while) entre chaves, mesmo quando o corpo consiste em apenas uma única instrução. Fazer isso de forma consistente pode evitar o tipo de problema que acabamos de mostrar, e aconselho você a adotar essa prática. Neste livro impresso, dou valor a manter o código de exemplo verticalmente compacto e nem sempre sigo meus próprios conselhos sobre esse assunto.

5.3.2 senão se

A instrução if/else avalia uma expressão e executa um dos dois trechos de código, dependendo do resultado. Mas e quando você precisa executar um dos vários trechos de código? Uma maneira de fazer isso é com uma instrução else if. else if não é realmente uma instrução JavaScript, mas simplesmente uma linguagem de programação usada com frequência que resulta quando instruções if/else repetidas são usadas:

```
se (n === 1) {  
    // Executa o bloco de código #1 } else if (n === 2) {  
  
    // Executa o bloco de código #2 } else if (n === 3) {  
  
    // Executa o bloco de código #3 } else {  
  
    // Se tudo mais falhar, execute o bloco #4
```

```
}
```

Não há nada de especial nesse código. É apenas uma série de declarações if, onde cada uma se segue se faz parte da cláusula else da declaração anterior. Usando o caso, se o idioma for preferível e mais legível do que, escrevendo essas declarações em sua forma sintaticamente equivalente e totalmente aninhada:

```
if (n === 1) {
    // Execute o bloco de código #1}

else {if (n === 2) {

    // Executar o bloco de código #2}

    else {if (n === 3) {

        // Executar o bloco de código #3}

        else {// se tudo mais falhar, execute o bloco #4

    } }

}
```

5.3.3 Switch

Uma instrução IF faz com que uma ramificação no fluxo da execução de um programa e você possa usar o outro se o idioma executar uma ramificação de várias via. Esta não é a melhor solução, no entanto, quando todos os ramos dependem do valor da mesma expressão. Nesse caso, é um desperdício avaliar repetidamente essa expressão em múltiplas declarações IF.

A declaração do Switch lida exatamente com essa situação. O interruptor

palavra-chave é seguida por uma expressão entre parênteses e um bloco de código entre chaves:

```
mudar(expressão) {  
    declarações }
```

No entanto, a sintaxe completa de uma instrução switch é mais complexa do que isso. Vários locais no bloco de código são rotulados com a palavra-chave case seguida por uma expressão e dois pontos. Quando uma opção é executada, ela calcula o valor da expressão e, em seguida, procura um rótulo de caso cuja expressão seja avaliada com o mesmo valor (onde a mesma é determinada pelo operador ===). Se encontrar um, ele começa a executar o bloco de código na instrução rotulada pelo case. Se não encontrar um caso com um valor correspondente, ele procura uma instrução denominada default:. Se não houver rótulo default:, a instrução switch ignora completamente o bloco de código.

switch é uma declaração confusa para explicar; seu funcionamento fica muito mais claro com um exemplo. A seguinte instrução switch é equivalente às instruções if/else repetidas mostradas na seção anterior:

```
mudar (n) {caso 1:  
    // Comece aqui se n === 1  
    // Executa o bloco de código #1. quebrar; // Pare aqui  
  
    caso 2:                      // Comece aqui se n === 2  
    // Executa o bloco de código #2. quebrar; // Pare aqui  
  
    caso 3:                      // Comece aqui se n === 3  
    // Executa o bloco de código #3. quebrar; // Pare aqui
```

```
padrão:                                // Se tudo mais falhar...
//Executa o bloco de código #4. quebrar; // Pare aqui
}

}
```

Observe a palavra-chave break usada no final de cada caso neste código. A instrução break, descrita posteriormente neste capítulo, faz com que o interpretador pule para o final (ou “quebre”) da instrução switch e continue com a instrução que a segue. As cláusulas case em uma instrução switch especificam apenas o ponto inicial do código desejado; eles não especificam nenhum ponto final. Na ausência de instruções break, uma instrução switch começa a executar seu bloco de código no rótulo case que corresponde ao valor de sua expressão e continua executando instruções até atingir o final do bloco. Em raras ocasiões, é útil escrever código como esse que “passa” de um rótulo de caso para o outro, mas em 99% das vezes você deve ter o cuidado de encerrar cada caso com uma instrução break. (Ao usar switch dentro de uma função, entretanto, você pode usar uma instrução return em vez de uma instrução break. Ambas servem para encerrar a instrução switch e evitar que a execução passe para o próximo caso.)

Aqui está um exemplo mais realista da instrução switch; ele converte um valor em uma string de uma forma que depende do tipo do valor:

```
função convert(x) {
switch(typeof x) { case
"número":           // Converte o número em um
inteiro hexadecimal
    retornar x.toString(16);
caso "string":     //Retorna a string incluída
entre aspas
```

```
        return '' + x + ''; padrão: // converte
        qualquer outro tipo
        da maneira usual
    retornar string (x); }

}
```

Observe que, nos dois exemplos anteriores, as palavras -chave do caso são seguidas por literais de número e string, respectivamente. É assim que a instrução Switch é mais frequentemente usada na prática, mas observe que o padrão ECMAScript permite que cada caso seja seguido por uma expressão arbitrária.

A instrução Switch primeiro avalia a expressão que segue a palavra -chave Switch e, em seguida, avalia as expressões de caso, na ordem em que aparecem, até encontrar um valor que corresponda. O caso correspondente é determinado usando o operador de identidade ===, não o operador de igualdade ==, portanto as expressões devem corresponder sem qualquer conversão de tipo.

Como nem todas as expressões de caso são avaliadas sempre que a instrução Switch for executada, você deve evitar o uso de expressões de caso que contêm efeitos colaterais, como chamadas ou atribuições de função. O curso mais seguro é simplesmente limitar suas expressões de caso a expressões constantes.

Conforme explicado anteriormente, se nenhuma das expressões de caso corresponder à expressão do interruptor, a instrução Switch começa a executar seu corpo na declaração marcada com o padrão:. Se não houver padrão: etiqueta, a instrução Switch ignora completamente seu corpo. Observe que no

Exemplos mostrados, a etiqueta padrão: aparece no final do corpo do interruptor, seguindo todos os rótulos da caixa. Este é um lugar lógico e comum, mas pode aparecer em qualquer lugar dentro do corpo da declaração.

5.4 Loops

Para entender as declarações condicionais, imaginamos o intérprete JavaScript após um caminho de ramificação através do seu código -fonte. As declarações de loop são aquelas que dobram esse caminho de volta para repetir partes do seu código. O JavaScript possui cinco declarações de loop: enquanto, enquanto, enquanto, para/de (e sua variante para/aguardam) e para/in. As seguintes subseções explicam cada uma por sua vez. Um uso comum para loops é iterar sobre os elementos de uma matriz. §7.6 discute esse tipo de loop em detalhes e abrange métodos especiais de loops definidos pela classe da matriz.

5.4.1 enquanto

Assim como a instrução IF é condicional básica do JavaScript, a declaração do tempo é o loop básico do JavaScript. Tem a seguinte sintaxe:

```
enquanto (expressão)
    declaração
```

Para executar uma declaração de tempo, o intérprete primeiro avalia a expressão. Se o valor da expressão for falsamente, o intérprete ignora a declaração que serve como corpo de loop e seguir para a próxima declaração no programa. Se, por outro lado, a expressão é verdadeira, o intérprete executa a declaração e repete, saltando

de volta ao topo do loop e avaliando a expressão novamente. Outra maneira de dizer isso é que o intérprete executa a declaração repetidamente enquanto a expressão é verdadeira. Observe que você pode criar um loop infinito com a sintaxe enquanto (true).

Geralmente, você não deseja que o JavaScript execute exatamente a mesma operação repetidamente. Em quase todos os loops, uma ou mais variáveis mudam a cada iteração do loop. Como as variáveis mudam, as ações executadas pela execução da declaração podem diferir cada vez por meio do loop. Além disso, se a variável ou variáveis alteradas estiver envolvida na expressão, o valor da expressão pode ser diferente a cada vez através do loop. Isso é importante; Caso contrário, uma expressão que começa a verdade nunca mudaria, e o loop nunca terminaria! Aqui está um exemplo de um loop de tempo que imprime os números de 0 a 9:

```
deixe count = 0; while  
(contagem < 10) {  
  
    console.log (contagem);  
    contagem++;  
}
```

Como você pode ver, a contagem de variáveis começa em 0 e é incrementada cada vez que o corpo do loop é executado. Depois que o loop é executado 10 vezes, a expressão se torna falsa (ou seja, a contagem de variáveis não é mais inferior a 10), o fim da instrução termina, e o intérprete pode passar para a próxima instrução no programa. Muitos loops têm uma variável contra a contagem. Os nomes das variáveis I, J e K são comumente usados como contadores de loop, embora você deva usar nomes mais descritivos se facilitar o entendimento do seu código.

5.4.2 fazer/enquanto

O loop do/while é como um loop while, exceto que a expressão do loop é testada na parte inferior do loop e não no topo. Isto significa que o corpo do loop é sempre executado pelo menos uma vez. A sintaxe é:

```
do  
instrução while  
(expressão);
```

O loop do/while é menos comumente usado que seu primo while — na prática, é um tanto incomum ter certeza de que você deseja que um loop seja executado pelo menos uma vez. Aqui está um exemplo de loop do/while:

```
função printArray(a) { deixe len =  
a.length, i = 0; if (len === 0) {  
  
    console.log("Matriz Vazia"); } outro {  
  
    do {  
        console.log(a[i]); } while(++i < len);  
  
} }
```

Existem algumas diferenças sintáticas entre o loop do/while e o loop while comum. Primeiro, o loop do requer a palavra-chave do (para marcar o início do loop) e a palavra-chave while (para marcar o fim e introduzir a condição do loop). Além disso, o loop do deve sempre terminar com ponto e vírgula. O loop while não precisa de ponto e vírgula se o corpo do loop estiver entre chaves.

5.4.3 para

A instrução for fornece uma construção em loop que geralmente é mais conveniente do que a declaração do tempo. A declaração for simplifica os loops que seguem um padrão comum. A maioria dos loops tem algum tipo de variável. Essa variável é inicializada antes do início do loop e ser testado antes de cada iteração do loop. Finalmente, a variável do contador é incrementada ou atualizada no final do corpo do loop, pouco antes de a variável ser testada novamente. Nesse tipo de loop, a inicialização, o teste e a atualização são os três cruciais

manipulações de uma variável de loop. A declaração for codifica cada uma dessas três manipulações como uma expressão e torna essas expressões uma parte explícita da sintaxe do loop:

```
para (inicializar; teste; incremento)
    declaração
```

Inicializar, teste e incremento são três expressões (separadas por semicolons) que são responsáveis por inicializar, testar e incrementar a variável de loop. Colocá -los todos na primeira linha do loop facilita o entendimento do que um loop está fazendo e evita erros, como esquecer de inicializar ou incrementar a variável do loop.

A maneira mais simples de explicar como funciona um loop é mostrar o equivalente enquanto o loop:

```
inicializar;
while (teste) {
    incremento da
    declaração;
}
```

Em outras palavras, a expressão de inicialização é avaliada uma vez, antes do início do loop. Para ser útil, esta expressão deve ter efeitos colaterais (geralmente uma atribuição). JavaScript também permite que inicialize seja uma instrução de declaração de variável para que você possa declarar e inicializar um contador de loop ao mesmo tempo. A expressão de teste é avaliada antes de cada iteração e controla se o corpo do loop é executado. Se test for avaliado como um valor verdadeiro, a instrução que é o corpo do loop será executada. Finalmente, a expressão de incremento é avaliada. Novamente, esta deve ser uma expressão com efeitos colaterais para ser útil. Geralmente, é uma expressão de atribuição ou usa os operadores `++` ou `--`.

Podemos imprimir os números de 0 a 9 com um loop `for` como o seguinte. Compare-o com o loop `while` equivalente mostrado na seção anterior:

```
for(deixe contar = 0; contar < 10; contar++) {  
    console.log(contar);  
}
```

Os loops podem se tornar muito mais complexos do que este exemplo simples, é claro, e às vezes diversas variáveis mudam a cada iteração do loop. Esta situação é o único lugar onde o operador vírgula é comumente usado em JavaScript; ele fornece uma maneira de combinar múltiplas expressões de inicialização e incremento em uma única expressão adequada para uso em um loop `for`:

```
seja i, j, soma = 0; for(i = 0, j = 10 ; i < 10 ; i++, j--) {  
    soma += i * j; }
```

Em todos os nossos exemplos de loop até agora, a variável de loop tem sido numérica. Isso é bastante comum, mas não é necessário. O código a seguir usa um loop for para atravessar uma estrutura de dados da lista vinculada e retornar o último objeto na lista (ou seja, o primeiro objeto que não possui uma próxima propriedade):

```
Função Tail (o) {  
    cauda de lista vinculada o  
    para (; o.next; o = o.next) / * vazio * /; //  
    atravessar enquanto o.Next é o retorno da verdade;  
}
```

Observe que este código não possui expressão inicializa. Qualquer uma das três expressões pode ser omitida de um loop for, mas os dois semicolons são necessários. Se você omitir a expressão do teste, o loop se repete para sempre e para (;;) é outra maneira de escrever um loop infinito, como (verdadeiro).

5.4.4 para/de

ES6 define uma nova declaração de loop: for/of. Esse novo tipo de loop usa a palavra -chave, mas é um tipo completamente diferente de loop do que o regular para loop. (Também é completamente diferente do mais antigo para/em loop que descreveremos no §5.4.5.)

O loop for/of trabalha com objetos iteráveis. Explicaremos exatamente o que isso significa para um objeto ser iterável no capítulo 12, mas para este capítulo, basta saber que matrizes, cordas, conjuntos e mapas são iteráveis: eles representam uma sequência ou conjunto de elementos que você pode fazer um loop ou iterar através do uso de um loop para/de.

Aqui, por exemplo, podemos usar for/of para percorrer os elementos de uma matriz de números e calcular sua soma:

```
deixe dados = [1, 2, 3, 4, 5, 6, 7, 8, 9], soma = 0; for (deixe o
elemento de dados) {
    soma += elemento;
}
soma      // => 45
```

Superficialmente, a sintaxe parece um loop for regular: a palavra-chave for é seguida por parênteses que contêm detalhes sobre o que o loop deve fazer. Neste caso, os parênteses contêm uma declaração de variável (ou, para variáveis que já foram declaradas, simplesmente o nome da variável) seguida pela palavra-chave of e uma expressão que avalia um objeto iterável, como o array de dados neste caso . Como acontece com todos os loops, o corpo de um loop for/of segue os parênteses, normalmente entre chaves.

No código mostrado, o corpo do loop é executado uma vez para cada elemento da matriz de dados. Antes de cada execução do corpo do loop, o próximo elemento da matriz é atribuído à variável do elemento. Os elementos da matriz são iterados em ordem, do primeiro ao último.

Matrizes são iteradas “ao vivo” – alterações feitas durante a iteração podem afetar o resultado da iteração. Se modificarmos o código anterior adicionando a linha data.push(sum); dentro do corpo do loop, criamos um loop infinito porque a iteração nunca pode atingir o último elemento do array.

PARA/DE COM OBJETOS

Os objetos não são (por padrão) iterable. Tentando usar para/de um objeto regular lança um TypeError no tempo de execução:

```
Seja o = {x: 1, y: 2, z: 3}; para (deixe o elemento  
de 0) { // lança TypeError porque 0 não é iterável  
  
console.log (elemento); }
```

Se você deseja iterar através das propriedades de um objeto, poderá usar o loop for/in (introduzido no §5.4.5) ou usar/de com o método object.keys ():

```
Seja o = {x: 1, y: 2, z: 3}; Let Keys = " ";  
  
para (deixe k de object.keys (o)) {chaves += k;  
  
}  
teclas // > "xyz";
```

Isso funciona porque object.Keys () retorna uma variedade de nomes de propriedades para um objeto, e as matrizes são iteráveis com/de. Observe também que essa iteração das teclas de um objeto não está ativa como o exemplo da matriz acima foi - as mudanças para o objeto O feito no corpo do loop não terão efeito na iteração. Se você não se importa com as chaves de um objeto, também pode iterar através dos valores correspondentes como este:

```
deixe soma = 0; para (deixe v de object.values (o)) {  
  
Soma += v; }  
  
soma // > 6
```

E se você estiver interessado nas chaves e nos valores das propriedades de um objeto, você pode usar for/of com Object.entries() e atribuição de desestruturação:

```
deixe pares = ""; for(deixe [k, v] de Object.entries(o))  
{  
    pares += k + v; }  
  
pares // => "x1y2z3"
```

Object.entries() retorna um array de arrays, onde cada array interno representa um par chave/valor para uma propriedade do objeto. Usamos atribuição de desestruturação neste exemplo de código para descompactar essas matrizes internas em duas variáveis individuais.

PARA/DE COM CORDAS

Strings são iteráveis caractere por caractere no ES6:

```
deixe frequênciia = {}; for(deixe a letra de "mississippi")  
{  
    if (frequênciia[letra]) { frequênciia[letra]++;  
  
    } outro {  
        frequênciia[letra] = 1; }  
  
}  
frequênciia // => {m: 1, i: 4, s: 4, p: 2}
```

Observe que as strings são iteradas pelo codepoint Unicode, não pelo caractere UTF-16. A string “I ❤” tem .length igual a 5 (porque os dois caracteres emoji requerem, cada um, dois caracteres UTF-16 para serem representados). Mas se você iterar essa string com for/of, o corpo do loop será executado três vezes, uma para cada um dos três pontos de código “I”, “❤” e “.”



Para/de com set e mapa

As classes de conjunto e mapa do ES6 embutidas são iteráveis. Quando você itera um conjunto com/de, o corpo do loop funciona uma vez para cada elemento do conjunto. Você pode usar código como este para imprimir as palavras exclusivas em uma sequência de texto:

```
Não deixe terminar ==. Permite outro Nandines = AAT New  
20) (Exterrit (0))); Oito sobre o não agrupado Agu = [];
```

```
para (deixe o word of wordset)  
{exclusivo.push (word);  
} hack // => ["in", "e", "Tamman"];
```

Os mapas são um caso interessante, porque o iterador para um objeto de mapa não itera as teclas do mapa ou os valores do mapa, mas os pares de chave/valor. Cada vez, através da iteração, o iterador retorna uma matriz cujo primeiro elemento é uma chave e cujo segundo elemento é o valor correspondente. Dado um mapa m, você pode iterar e destruir seus pares de chave/valor como este:

```
Seja M = novo mapa ([[1, "One"]]);  
para (let [chave, valor] de m) {  
    chave // => 1  
    valor // => "um"  
}
```

Iteração assíncrona com para/aguardar

O ES2018 apresenta um novo tipo de iterador, conhecido como iterador assíncrono, e uma variante no loop for/of, conhecida como o loop for/wait que funciona com iteradores assíncronos.

Você precisará ler os capítulos 12 e 13 para entender o loop for/wait, mas aqui está como ele fica no código:

```
// leia pedaços de um fluxo de maneira assíncrona e imprimi-los
Função assíncrona printStream (stream) {
  para aguardar (Let Chunk of Stream) {
    console.log (pedaço); }

}
```

5.4.5 para/in

A for/in loop se parece muito com um loop para/de uma palavra -chave alterada para in. O loop for/of é novo no ES6, mas para/in faz parte do JavaScript desde o início (e é por isso que tem a sintaxe de som mais natural).

O FOR/in INCLUI VOLTA POR os nomes de propriedades de um objeto especificado. A sintaxe se parece com a seguinte:

```
para (variável no objeto)
  declaração
```

A variável normalmente nomeia uma variável, mas pode ser uma declaração variável ou qualquer coisa adequada como o lado esquerdo de uma expressão de atribuição. Objeto é uma expressão que avalia para um objeto. Como sempre, a declaração é a declaração ou o bloco de declaração que serve como o corpo do loop.

E você pode usar um loop para/em um loop assim:

```
for(seja p em o) {           // Atribui nomes de propriedades de o para  
    variável p  
    console.log(o[p]); // Imprime o valor de cada propriedade }
```

Para executar uma instrução for/in, o interpretador JavaScript primeiro avalia a expressão do objeto. Se for avaliado como nulo ou indefinido, o interpretador pula o loop e passa para a próxima instrução. O interpretador agora executa o corpo do loop uma vez para cada propriedade enumerável do objeto. Antes de cada iteração, entretanto, o interpretador avalia a expressão da variável e atribui a ela o nome da propriedade (um valor de string).

Observe que a variável no loop for/in pode ser uma expressão arbitrária, desde que seja avaliada como algo adequado para o lado esquerdo de uma atribuição. Essa expressão é avaliada a cada vez no loop, o que significa que ela pode ser avaliada de maneira diferente a cada vez. Por exemplo, você pode usar um código como o seguinte para copiar os nomes de todas as propriedades do objeto em um array:

```
seja o = { x: 1, y: 2, z: 3 }; seja a = [], i = 0;  
for(a[i++] in o) /* vazio */;
```

Arrays JavaScript são simplesmente um tipo especializado de objeto, e índices de array são propriedades de objetos que podem ser enumeradas com um loop for/in. Por exemplo, seguir o código anterior com esta linha enumera os índices da matriz 0, 1 e 2:

```
for(deixe i em a) console.log(i);
```

Acho que uma fonte comum de bugs em meu próprio código é o uso acidental de/in com matrizes quando eu pretendia usar para/de. Ao trabalhar com matrizes, você quase sempre deseja usar para/de em vez de para/in.

O loop for/in não enumora todas as propriedades de um objeto. Não enumora propriedades cujos nomes são símbolos. E das propriedades cujos nomes são strings, ele apenas atravessa as propriedades enumeráveis (ver §14.1). Os vários métodos internos definidos pelo Core JavaScript não são enumeráveis. Todos os objetos possuem um método `toString()`, por exemplo, mas o loop for/in não enumora essa propriedade `ToString`. Além dos métodos internos, muitas outras propriedades dos objetos embutidos não são entusiasmados. Todas as propriedades e métodos definidos pelo seu código são enumeráveis, por padrão. (Você pode torná-los que não são digitais usando técnicas explicadas no §14.1.)

Propriedades herdadas enumeráveis (ver §6.3.2) também são enumeradas pelo loop for/in. Isso significa que, se você usar para/em loops e também usar o código que define propriedades herdadas por todos os objetos, seu loop poderá não se comportar da maneira que você espera. Por esse motivo, muitos programadores preferem usar um loop for/de `object.keys()` em vez de um loop for/in.

Se o corpo de A for/in loop excluir uma propriedade que ainda não foi enumerada, essa propriedade não será enumerada. Se o corpo do loop definir novas propriedades no objeto, essas propriedades podem ou não ser enumeradas. Consulte §6.6.1 para obter mais informações sobre o pedido em

que para/em enumera as propriedades de um objeto.

5.5 saltos

Outra categoria de declarações de JavaScript são as declarações de salto. Como o nome indica, isso faz com que o intérprete JavaScript entre em um novo local no código -fonte. A declaração de quebra faz o intérprete saltar para o final de um loop ou outra declaração. Continuar faz o intérprete pular o restante do corpo de um loop e voltar para o topo de um loop para iniciar uma nova iteração. O JavaScript permite que as declarações sejam nomeadas ou rotuladas, e quebrar e continuar podem identificar o loop de destino ou outro rótulo de instrução.

A instrução Return faz com que o intérprete salva de uma invocação de função de volta ao código que o invocou e também fornece o valor para a invocação. A declaração de arremesso é uma espécie de retorno intermediário de uma função de gerador. A declaração de arremesso aumenta, ou joga, uma exceção e foi projetada para trabalhar com a instrução Try/Catch/Finalmente, que estabelece um bloco de código de manipulação de exceção. Este é um tipo complicado de declaração de salto: quando uma exceção é lançada, o intérprete salta para o manipulador de exceção de anexo mais próximo, que pode estar na mesma função ou subir a pilha de chamadas em uma função de invocação.

Detalhes sobre cada uma dessas declarações de salto estão nas seções a seguir.

5.5.1 Declarações rotuladas

Qualquer instrução pode ser rotulada precedendo-a com um identificador e dois pontos:

```
identificador: declaração
```

Ao rotular uma instrução, você atribui a ela um nome que pode ser usado para se referir a ela em outro lugar do programa. Você pode rotular qualquer instrução, embora seja útil apenas rotular instruções que possuem corpos, como loops e condicionais. Ao dar um nome a um loop, você pode usar instruções break e continue dentro do corpo do loop para sair do loop ou para pular diretamente para o topo do loop para iniciar a próxima iteração. break e continue são as únicas instruções JavaScript que usam rótulos de instrução; eles são abordados nas subseções a seguir. Aqui está um exemplo de um loop while rotulado e uma instrução continue que usa o rótulo.

```
mainloop: while(token !== null) { // Código omitido...
    continue mainloop; // Salta para a próxima iteração do
    laço nomeado
    // Mais código omitido... }
```

O identificador que você usa para rotular uma instrução pode ser qualquer identificador JavaScript legal que não seja uma palavra reservada. O namespace para rótulos é diferente do namespace para variáveis e funções, portanto você pode usar o mesmo identificador como um rótulo de instrução e como um nome de variável ou função. Os rótulos das declarações são definidos apenas dentro da declaração à qual se aplicam (e dentro de suas subafirmações, é claro). Uma instrução pode não ter o mesmo rótulo que uma instrução que a contém, mas duas instruções podem ter o mesmo rótulo, desde que nenhuma delas esteja aninhada.

dentro do outro. As declarações rotuladas podem elas próprias ser rotuladas. Efetivamente, isso significa que qualquer declaração pode ter vários rótulos.

5.5.2 pausa

A instrução break, usada sozinha, faz com que o loop mais interno ou a instrução switch saiam imediatamente. Sua sintaxe é simples:

```
quebrar;
```

Como causa a saída de um loop ou switch, esta forma de instrução break é válida apenas se aparecer dentro de uma dessas instruções.

Você já viu exemplos da instrução break dentro de uma instrução switch. Em loops, normalmente é usado para sair prematuramente quando, por qualquer motivo, não há mais necessidade de completar o loop. Quando um loop tem condições de terminação complexas, geralmente é mais fácil implementar algumas dessas condições com instruções break, em vez de tentar expressá-las todas em uma única expressão de loop. O código a seguir pesquisa os elementos de uma matriz em busca de um valor específico. O loop termina normalmente quando atinge o final do array; ele termina com uma instrução break se encontrar o que procura no array:

```
for(deixe i = 0; i < a.length; i++) { if (a[i] === alvo) break;  
}
```

JavaScript também permite que a palavra-chave break seja seguida por um rótulo de instrução (apenas o identificador, sem dois pontos):

```
quebrar nome do rótulo;
```

Quando break é usado com um rótulo, ele salta para o final ou termina a instrução envolvente que possui o rótulo especificado. É um erro de sintaxe usar break neste formato se não houver nenhuma instrução delimitadora com o rótulo especificado. Com esta forma de instrução break, a instrução nomeada não precisa ser um loop ou switch: break pode “romper” qualquer instrução anexa. Esta instrução pode até ser um bloco de instruções agrupado entre chaves com o único propósito de nomear o bloco com um rótulo.

Uma nova linha não é permitida entre a palavra-chave break e o labelname. Isso é resultado da inserção automática de ponto-e-vírgula omitido no JavaScript: se você colocar um terminador de linha entre a palavra-chave break e o rótulo a seguir, o JavaScript assumirá que você pretendia usar a forma simples e sem rótulo da instrução e tratará o terminador de linha como um ponto-e-vírgula . (Ver §2.6.)

Você precisa da forma rotulada da instrução break quando deseja interromper uma instrução que não é o loop ou switch mais próximo. O código a seguir demonstra:

```
deixe matriz = getData();      // Obtém um array 2D de números de
em algum lugar // Agora some todos os números
da matriz. seja soma = 0, sucesso = falso;

// Comece com uma instrução rotulada da qual podemos interromper
// se ocorrerem erros
computaSoma: if (matriz) {
    for(deixe x = 0; x < matriz.comprimento;
        x++) { deixe linha = matriz[x];
        if (!row) quebrar computaSum;
```

```
        for(deixe y = 0; y < comprimento da
            linha; y++) { deixa célula = linha[y];
                if (isNaN(célula)) quebrar computaSum; soma += célula;
            } } sucesso =
verdadeiro;

} // As instruções break saltam aqui. Se
chegarmos aqui com sucesso == false
// então havia algo errado com a matriz que nos foi dada.
// Caso contrário, sum contém a soma de todas as células
da matriz.
```

Finalmente, observe que uma instrução `break`, com ou sem rótulo, não pode transferir o controle através dos limites da função. Você não pode rotular uma instrução de definição de função, por exemplo, e depois usar esse rótulo dentro da função.

5.5.3 continuar

A instrução `continue` é semelhante à instrução `break`. Em vez de sair de um loop, entretanto, `continue` reinicia um loop na próxima iteração. A sintaxe da instrução `continue` é tão simples quanto a da instrução `break`:

```
continuar;
```

A instrução `continue` também pode ser usada com um rótulo:

```
continue nomedarótulo;
```

A instrução `continue`, tanto na forma rotulada quanto na não rotulada, pode

ser usado apenas dentro do corpo de um loop. Usá -lo em qualquer outro lugar, causa um erro de sintaxe.

Quando a declaração continua é executada, a iteração atual do loop de anexo é encerrada e a próxima iteração começa. Isso significa coisas diferentes para diferentes tipos de loops:

- Em um tempo, o loop, a expressão especificada no início do loop é testada novamente e, se for verdade, o corpo do loop é executado a partir do topo.
- Em um loop DO/enquanto, a execução pula para a parte inferior do loop, onde a condição do loop é testada novamente antes de reiniciar o loop na parte superior.
- Em um loop for, a expressão de incremento é avaliada e a expressão do teste é testada novamente para determinar se outra iteração deve ser feita.
- Em um para/de ou para/em loop, o loop começa com o próximo valor iterado ou o próximo nome da propriedade sendo atribuído à variável especificada.

Observe a diferença no comportamento da declaração de continuação no tempo e nos loops: há um tempo o loop retorna diretamente à sua condição, mas um loop for primeiro avalia sua expressão de incremento e depois retorna à sua condição. Anteriormente, consideramos o comportamento do loop for em termos de um "equivalente" enquanto loop. Como a declaração continua se comporta de maneira diferente para esses dois loops, no entanto, não é possível simular perfeitamente um loop com um loop sozinho.

O exemplo a seguir mostra uma instrução continue sem rótulo sendo usada para pular o resto da iteração atual de um loop quando ocorre um erro:

```
for(deixe i = 0; i < data.length; i++) { if (!data[i]) continue;  
//Não é possível prosseguir com indefinido  
dados  
total += dados[i]; }
```

Assim como a instrução break, a instrução continue pode ser usada em sua forma rotulada dentro de loops aninhados quando o loop a ser reiniciado não é o loop imediatamente envolvente. Além disso, como acontece com a instrução break, quebras de linha não são permitidas entre a instrução continue e seu labelname.

5.5.4 retorno

Lembre-se de que as invocações de funções são expressões e que todas as expressões possuem valores. Uma instrução return dentro de uma função especifica o valor das invocações dessa função. Aqui está a sintaxe da instrução return:

```
expressão de retorno;
```

Uma instrução return pode aparecer apenas dentro do corpo de uma função. É um erro de sintaxe aparecer em qualquer outro lugar. Quando a instrução return é executada, a função que a contém retorna o valor da expressão ao seu chamador. Por exemplo:

```
função quadrado(x) { return x*x; } // Uma função que possui uma  
instrução de retorno
```

```
quadrado (2)// => 4
```

Sem instrução de retorno, uma invocação de função simplesmente executa cada uma das instruções no corpo da função até chegar ao final da função e então retornar ao seu chamador. Neste caso, a expressão de invocação é avaliada como indefinida. A instrução return geralmente aparece como a última instrução em uma função, mas não precisa ser a última: uma função retorna ao seu chamador quando uma instrução return é executada, mesmo se houver outras instruções restantes no corpo da função.

A instrução return também pode ser usada sem uma expressão para fazer a função retornar indefinida ao seu chamador. Por exemplo:

```
function displayObject(o) { // Retorna imediatamente se o argumento for
  nulo ou
  indefinido.
  se (!o) retornar; // Resto da função vai aqui...
}
```

Devido à inserção automática de ponto e vírgula do JavaScript (§2.6), você não pode incluir uma quebra de linha entre a palavra-chave return e a expressão que a segue.

5.5.5 rendimento

A instrução yield é muito parecida com a instrução return, mas é usada apenas em funções geradoras ES6 (consulte §12.3) para produzir o próximo valor na sequência de valores gerada sem realmente retornar:

```
// Uma função geradora que produz um intervalo de inteiros
```

```
função* intervalo(de, até) {  
    for(deixe i = de; i <= para; i++) { rendimento i;  
}
```

Para entender o rendimento, você deve entender os iteradores e geradores, que não serão abordados até o Capítulo 12. Entretanto, o rendimento está incluído aqui para completar. (Tecnicamente, porém, o rendimento é um operador e não uma declaração, conforme explicado em §12.4.2.)

5.5.6 Lançamento

Uma exceção é um sinal que indica que ocorreu algum tipo de condição ou erro excepcional. Lançar uma exceção é sinalizar tal erro ou condição excepcional. Capturar uma exceção é tratá-la – tomar quaisquer ações necessárias ou apropriadas para se recuperar da exceção. Em JavaScript, exceções são lançadas sempre que ocorre um erro de tempo de execução e sempre que o programa lança um erro explicitamente usando a instrução `throw`. Exceções são capturadas com o

`instrução try/catch/finally, que é descrita na próxima seção.`

A instrução `throw` tem a seguinte sintaxe:

```
lançar expressão;
```

expressão pode ser avaliada como um valor de qualquer tipo. Você pode lançar um número que representa um código de erro ou uma string que contém uma mensagem de erro legível por humanos. A classe `Error` e suas subclasses são usadas quando o próprio interpretador JavaScript gera um erro, e você pode usar

eles também. Um objeto de erro possui uma propriedade de nome que especifica o tipo de erro e uma propriedade de mensagem que contém a string passada para a função do construtor. Aqui está uma função de exemplo que lança um objeto de erro quando invocado com um argumento inválido:

```
função factorial (x) { // Se o argumento de entrada for inválido,  
    // faça uma exceção! se (x < 0) lançar um novo erro ("x não  
    // deve ser negativo"); // caso contrário, calcule um valor e  
    // retorne normalmente  
    deixe f; for (f = 1; x > 1; f *=  
        x, x--) / *vazio * /; retornar f;  
  
} Fatorial (4)  
// => 24
```

Quando uma exceção é lançada, o intérprete JavaScript imediatamente interrompe a execução normal do programa e salta para o manipulador de exceção mais próximo. Os manipuladores de exceção são escritos usando a cláusula de captura da instrução Try/Catch/Finalmente, descrita na próxima seção. Se o bloco de código no qual a exceção foi lançada não tiver uma cláusula de captura associada, o intérprete verifica o próximo bloco de código de gabinetes mais alto para ver se ele possui um manipulador de exceção associado a ele. Isso continua até que um manipulador seja encontrado. Se uma exceção for lançada em uma função que não contenha um

Tente/Catch/finalmente a instrução para lidar com isso, a exceção se propaga até o código que invocou a função. Dessa forma, as exceções se propagam através da estrutura lexical dos métodos JavaScript e subindo a pilha de chamadas. Se nenhum manipulador de exceção for encontrado, a exceção é tratada como um erro e é relatada ao usuário.

5.5.7 Tente/Catch/Finalmente

A instrução try/catch/finally é o mecanismo de tratamento de exceções do JavaScript. A cláusula try desta instrução simplesmente define o bloco de código cujas exceções devem ser tratadas. O bloco try é seguido por uma cláusula catch, que é um bloco de instruções invocadas quando ocorre uma exceção em qualquer lugar do bloco try. A cláusula catch é seguida por um bloco final contendo código de limpeza que tem execução garantida, independentemente do que acontecer no bloco try. Os blocos catch e finalmente são opcionais, mas um bloco try deve ser acompanhado por pelo menos um desses blocos. Os blocos try, catch e finalmente começam e terminam com chaves. Essas chaves são uma parte obrigatória da sintaxe e não podem ser omitidas, mesmo que uma cláusula contenha apenas uma única instrução.

O código a seguir ilustra a sintaxe e o propósito da instrução try/catch/finally:

```
try { // Normalmente, esse código é executado do topo do bloco até  
    // fundo  
    // sem problemas. Mas às vezes pode gerar uma exceção,  
    // diretamente, com uma instrução throw, ou  
    // indiretamente, chamando  
    // um método que lança uma exceção. }  
  
pegar (e) {  
    // As instruções neste bloco são executadas se, e somente se, o  
    // try  
    // bloco lança uma exceção. Essas instruções podem usar a  
    // variável local  
    // e para se referir ao objeto Error ou outro valor que foi  
    // lançado.  
    // Este bloco pode tratar a exceção de alguma forma, pode  
    // ignorar o  
    // exceção sem fazer nada, ou pode relançar a exceção  
    // com throw.
```

```
    } finalmente
    {
        // Este bloco contém instruções que são sempre
        executadas, independentemente de
        // o que acontece no bloco try. Eles são executados se o
        try
            // o bloco termina: // 1) normalmente, após atingir o final
            // do bloco // 2) devido a uma instrução break, continue ou
            return // 3) com uma exceção que é tratada por um catch

        cláusula acima
        //    4) com uma exceção não detectada que ainda é
        propagando }
```

Observe que a palavra-chave `catch` geralmente é seguida por um identificador entre parênteses. Este identificador é como um parâmetro de função. Quando uma exceção é capturada, o valor associado à exceção (um objeto `Error`, por exemplo) é atribuído a este parâmetro. O identificador associado a uma cláusula `catch` tem escopo de bloco – ele é definido apenas dentro do bloco `catch`.

Aqui está um exemplo realista da instrução `try/catch`. Ele usa o método `factorial()` definido na seção anterior e os métodos JavaScript do lado do cliente `prompt()` e `alert()` para entrada e saída:

```
try { // Peça ao usuário para inserir um número let n =
Number(prompt("Por favor, insira um número inteiro
positivo"));
// Calcula o fatorial do número, assumindo que a entrada é
válida
    seja f = fatorial (n); //
    Exibe o resultado alert(n
    + "!" + f);
}
```

```
pegar(ex) {          // Se a entrada do usuário não for válida, finalizamos  
    aqui em cima  
    alerta (ex); //Informa ao usuário qual é o erro  
}
```

Este exemplo é uma instrução try/catch sem cláusula final. Embora finalmente não seja usado com tanta frequência quanto catch, pode ser útil. No entanto, seu comportamento requer explicação adicional. A cláusula finalmente será executada se qualquer parte do bloco try for executada, independentemente de como o código no bloco try for concluído. Geralmente é usado para limpar o código na cláusula try.

No caso normal, o interpretador JavaScript chega ao final do bloco try e depois segue para o bloco final, que executa qualquer limpeza necessária. Se o intérprete deixou o bloco try por causa de uma instrução return, continue ou break, o bloco finalmente será executado antes que o intérprete salte para seu novo destino.

Se ocorrer uma exceção no bloco try e houver um bloco catch associado para tratar a exceção, o interpretador primeiro executa o bloco catch e depois o bloco finalmente. Se não houver nenhum bloco catch local para tratar a exceção, o interpretador primeiro executa o bloco finalmente e depois salta para a cláusula catch mais próxima.

Se um bloco finalmente causar um salto com uma instrução return, continue, break ou throw, ou chamando um método que lança uma exceção, o interpretador abandona qualquer salto que estava pendente e executa o novo salto. Por exemplo, se uma cláusula finalmente lançar um

exceção, essa exceção substitui qualquer exceção que estava em processo de lançamento. Se uma cláusula finalmente emitir uma instrução return, o método retornará normalmente, mesmo que uma exceção tenha sido lançada e ainda não tenha sido tratada.

try e finalmente podem ser usados juntos sem uma cláusula catch. Nesse caso, o bloco final é simplesmente um código de limpeza com execução garantida, independentemente do que acontecer no bloco try. Lembre-se de que não podemos simular completamente um loop for com um loop while porque a instrução continue se comporta de maneira diferente para os dois loops. Se adicionarmos uma instrução try/finalmente, podemos escrever um loop while que funciona como um loop for e que lida com instruções continue corretamente:

```
// Simula for(initialize ; test ;increment ) body;  
inicializar;  
while(teste) {tente {corpo;  
} finalmente {incremento; }  
}
```

Observe, no entanto, que um corpo que contém uma instrução break se comporta de maneira ligeiramente diferente (causando um incremento extra antes de sair) no loop while e no loop for, portanto, mesmo com a cláusula finalmente, não é possível simular completamente o for loop com while.

CLÁUSULAS DE CAPTURA BARE

Ocasionalmente, você pode usar uma cláusula catch apenas para detectar e interromper a propagação de uma exceção, mesmo que não se importe com o tipo ou o valor da exceção. No ES2019 e posterior, você pode omitir os parênteses e o identificador e usar a palavra-chave catch simples neste caso. Aqui está um exemplo:

```
// Como JSON.parse(), mas retorna indefinido em vez de lançar uma função
de erro parseJSON(s) {
    tente {retornar JSON.parse(s);

    } pegar {
        // Algo deu errado, mas não nos importamos com o que
        foi return undefined;
    } }
```

5.6 Declarações Diversas

Esta seção descreve as três instruções JavaScript restantes —with, debugger e "use strict";.

5.6.1 com

A instrução with executa um bloco de código como se as propriedades de um objeto especificado fossem variáveis no escopo desse código. Possui a seguinte sintaxe:

```
com instrução
(objeto)
```

Esta instrução cria um escopo temporário com as propriedades do objeto como variáveis e então executa a instrução dentro desse escopo.

A instrução with é proibida no modo estrito (ver §5.6.3) e deve ser considerada obsoleta no modo não estrito: evite usá-la sempre que possível. O código JavaScript usado com é difícil de otimizar e provavelmente será executado significativamente mais lentamente do que o código equivalente escrito sem a instrução with.

O uso comum da instrução `with` é facilitar o trabalho com hierarquias de objetos profundamente aninhadas. Em JavaScript do lado do cliente, por exemplo, pode ser necessário digitar expressões como esta para acessar elementos de um formulário HTML:

```
documento.forms[0].endereço.valor
```

Se precisar escrever expressões como esta várias vezes, você pode usar a instrução `with` para tratar as propriedades do objeto de formulário como variáveis:

```
com(document.forms[0]) {  
    // Acesse os elementos do formulário diretamente aqui. Por exemplo:  
    nome.valor = " ";  
    endereço.valor = " ";  
    email.valor = " ";  
}
```

Isso reduz a quantidade de digitação necessária: você não precisa mais prefixar cada nome de propriedade do formulário com `document.forms[0]`. É igualmente simples, claro, evitar a instrução `with` e escrever o código anterior assim:

```
deixe f = document.forms[0];  
f.nome.valor = " ";  
f.endereço.value = " ";  
f.email.value = " ";
```

Observe que se você usar `const` ou `let` ou `var` para declarar uma variável ou constante dentro do corpo de uma instrução `with`, isso criará uma variável comum e não definirá uma nova propriedade dentro do objeto especificado.

5.6.2 depurador

A instrução do depurador normalmente não faz nada. Se, no entanto, um programa depurador estiver disponível e em execução, então uma implementação poderá (mas não é obrigatória) executar algum tipo de ação de depuração. Na prática, esta instrução atua como um ponto de interrupção: a execução do código JavaScript é interrompida e você pode usar o depurador para imprimir valores de variáveis, examinar a pilha de chamadas e assim por diante. Suponha, por exemplo, que você esteja recebendo uma exceção em sua função `f()` porque ela está sendo chamada com um argumento indefinido e você não consegue descobrir de onde vem essa chamada. Para ajudá-lo a depurar esse problema, você pode alterar `f()` para que comece assim:

```
função f(o) {
    if (o === indefinido) depurador;      //Linha temporária para
    fins de depuração
    ...
    //O resto da função
    vai aqui. }
```

Agora, quando `f()` for chamado sem argumento, a execução será interrompida e você poderá usar o depurador para inspecionar a pilha de chamadas e descobrir de onde vem essa chamada incorreta.

Observe que não é suficiente ter um depurador disponível: a instrução `debugger` não iniciará o depurador para você. Se você estiver usando um navegador da Web e tiver o console de ferramentas do desenvolvedor aberto, essa instrução causará um ponto de interrupção.

5.6.3 “uso estrito”

`"Use Strict"` é uma diretiva introduzida no ES5. As diretrizes não são declarações (mas estão próximas o suficiente para que `"uso rigoroso"` esteja documentado aqui). Existem duas diferenças importantes entre a diretiva `"Use Strict"` e declarações regulares:

- Ele não inclui nenhuma palavra-chave de idioma: a diretiva é apenas uma declaração de expressão que consiste em uma string especial literal (em cotações únicas ou duplas).
- Ele pode aparecer apenas no início de um script ou no início de um corpo de função, antes que qualquer declaração real apareça.

O objetivo de uma diretiva `"uso rigoroso"` é indicar que o código a seguir (no script ou função) é um código rigoroso. O código de nível superior (não função) de um script é um código rigoroso se o script tiver uma diretiva `"usar rigorosa"`. Um corpo de função é um código rigoroso se for definido no código rigoroso ou se tiver uma diretiva `"usar rigorosa"`. O código passado para o método `Eval()` é o código rigoroso se `Eval()` for chamado do código rigoroso ou se a string de código incluir uma diretiva `"use rigorosa"`. Além do código declarado explicitamente como rigoroso, qualquer código em um corpo de classe (capítulo 9) ou em um módulo ES6 (§10.3) é automaticamente um código rigoroso. Isso significa que, se todo o seu código JavaScript for escrito como módulos, tudo será automaticamente rigoroso e você nunca precisará usar uma diretiva explícita `"usar rigorosa"`.

O código rigoroso é executado no modo rigoroso. O modo rigoroso é um subconjunto restrito do idioma que corrige deficiências importantes da linguagem e fornece uma verificação de erros mais forte e maior segurança. Como o modo rigoroso não é o código JavaScript antigo e padrão que ainda usa os recursos deficientes do idioma, continuarão sendo executados corretamente. As diferenças

Entre o modo rigoroso e o modo não estrito, os seguintes (os três primeiros são particularmente importantes):

- A declaração com com o modo não é permitido no modo rigoroso.
- No modo rigoroso, todas as variáveis devem ser declaradas: um `referenceError` é lançado se você atribuir um valor a um identificador que não é uma variável declarada, função, parâmetro de função, parâmetro de cláusula de captura ou propriedade do objeto global. (No modo não estrito, isso declara implicitamente uma variável global adicionando uma nova propriedade ao objeto global.)
- No modo rigoroso, as funções invocadas como funções (e não como métodos) têm um valor desse valor indefinido. (No modo não rigoroso, as funções invocadas como funções são sempre passadas no objeto global como esse valor.) Além disso, no modo rigoroso, quando uma função é invocada com chamada () ou aplicar () (§8.7.4), o Este valor é exatamente o valor aprovado como o primeiro argumento a chamar () ou aplicar (). (No modo não estrito, valores nulos e indefinidos são substituídos pelos valores globais de objeto e não-objeto são convertidos em objetos.)
- No modo rigoroso, as atribuições para propriedades não escritas e tentativas de criar novas propriedades em objetos não extensíveis lançam um `TypeError`. (No modo não estrito, essas tentativas falham em silêncio.)
- No modo rigoroso, o código passado para avaliar () não pode declarar variáveis ou definir funções no escopo do chamador, como pode no modo não rito. Em vez disso, as definições de variáveis e funções vivem em um novo escopo criado para o `Eval` (). Esse escopo é descartado quando o `EVAL` () retorna.
- No modo rigoroso, o objeto de argumentos (§8.3.3) em uma função contém uma cópia estática dos valores passados para a função. No modo não rigoroso, o objeto de argumentos tem comportamento "mágico" em

quais elementos da matriz e parâmetros de função nomeados referem-se ao mesmo valor.

- No modo estrito, um SyntaxError será lançado se o operador delete for seguido por um identificador não qualificado, como uma variável, função ou parâmetro de função. (No modo não estrito, essa expressão de exclusão não faz nada e é avaliada como falsa.)
- No modo estrito, uma tentativa de excluir uma propriedade não configurável gera um TypeError. (No modo não estrito, a tentativa falha e a expressão de exclusão é avaliada como falsa.)
- No modo estrito, é um erro de sintaxe um objeto literal definir duas ou mais propriedades com o mesmo nome. (No modo não estrito, nenhum erro ocorre.)
- No modo estrito, é um erro de sintaxe uma declaração de função ter dois ou mais parâmetros com o mesmo nome. (No modo não estrito, nenhum erro ocorre.)
- No modo estrito, literais inteiros octais (começando com 0 que não é seguido por x) não são permitidos. (No modo não estrito, algumas implementações permitem literais octais.)
- No modo estrito, os identificadores eval e argumentos são tratados como palavras-chave e você não tem permissão para alterar seus valores. Você não pode atribuir um valor a esses identificadores, declará-los como variáveis, usá-los como nomes de funções, usá-los como nomes de parâmetros de funções ou usá-los como identificador de um bloco catch.
- No modo estrito, a capacidade de examinar a pilha de chamadas é restrito. arguments.caller e argumentscallee lançam um TypeError dentro de uma função de modo estrito. As funções de modo estrito também possuem propriedades de chamador e argumentos que lançam TypeError quando lidas.

(Algumas implementações definem essas propriedades fora do padrão em funções não rigíveis.)

5.7 declarações

As palavras -chave const, let, var, function, classe, importação e exportação não são tecnicamente declarações, mas se parecem muito com declarações, e este livro se refere informalmente a elas como declarações, para que mereçam uma menção neste capítulo.

Essas palavras -chave são descritas com mais precisão como declarações, em vez de declarações. Dissemos no início deste capítulo que declarações "fazem algo acontecer". As declarações servem para definir novos valores e dar a eles nomes que podemos usar para nos referir a esses valores. Eles não fazem muito acontecer, mas fornecendo nomes para valores, em um sentido importante, definem o significado das outras declarações do seu programa.

Quando um programa é executado, são as expressões do programa que estão sendo avaliadas e as declarações do programa que estão sendo executadas. As declarações em um programa não "correm" da mesma maneira: em vez disso, elas definem a estrutura do próprio programa. Pouco, você pode pensar nas declarações como as partes do programa que são processadas antes que o código comece a ser executado.

As declarações de JavaScript são usadas para definir constantes, variáveis, funções e classes e para importar e exportar valores entre os módulos. As próximas subseções dão exemplos de todos esses declarações. Todos estão cobertos com muito mais detalhes em outros lugares em

este livro.

5.7.1 const, let e var

As declarações Const, Let e VAR são cobertas em §3.10. Em ES6 e posterior, Const declara constantes e deixe declara variáveis. Antes do ES6, a palavra -chave VAR era a única maneira de declarar variáveis e não havia como declarar constantes. As variáveis declaradas com VAR são escopo para a função contendo, em vez do bloco contendo. Isso pode ser uma fonte de bugs e, no javascript moderno, não há realmente nenhuma razão para usar o VAR em vez de LET.

```
const tau = 2*math.pi;  
Deixe o raio = 3;  
var circunferência = tau * raio;
```

5.7.2 Função

A declaração de função é usada para definir funções, que são cobertas em detalhes no capítulo 8 (também vimos função no §4.3, onde foi usada como parte de uma expressão de função em vez de uma declaração de função.) Uma declaração de função se parece com esta :

```
área de função (raio) {return math.pi * raio * raio;  
}
```

Uma declaração de função cria um objeto de função e o atribui ao nome especificado - áreas neste exemplo. Em outras partes do nosso programa, podemos nos referir à função - e executar o código dentro dele - usando esse nome. As declarações de função em qualquer bloco de código JavaScript são

processado antes da execução do código e os nomes das funções são vinculados aos objetos de função em todo o bloco. Dizemos que as declarações de função são “içadas” porque é como se todas tivessem sido movidas para o topo de qualquer escopo em que estejam definidas. O resultado é que o código que invoca uma função pode existir no seu programa antes do código que declara a função.

§12.3 descreve um tipo especial de função conhecida como gerador. As declarações do gerador usam a palavra-chave `function`, mas seguem-na com um asterisco. §13.3 descreve funções assíncronas, que também são declaradas usando a palavra-chave `function`, mas são prefixadas com a palavra-chave `async`.

5.7.3 aula

No ES6 e posteriores, a declaração de classe cria uma nova classe e dá a ela um nome que podemos usar para nos referirmos a ela. As classes são descritas em detalhes no Capítulo 9. Uma simples declaração de classe pode ser assim:

```
class Circle { construtor (raio) { this.r = raio;
} area() { return Math.PI * this.r * this.r; }
circunferênci() { return 2 * Math.PI * this.r; }

}
```

Ao contrário das funções, as declarações de classe não são elevadas e você não pode usar uma classe declarada dessa forma no código que aparece antes da declaração.

5.7.4 importação e exportação

As declarações de importação e exportação são utilizadas em conjunto para fazer

Valores definidos em um módulo de código JavaScript disponível em outro módulo. Um módulo é um arquivo de código JavaScript com seu próprio espaço de nome global, completamente independente de todos os outros módulos. A única maneira de um valor (como função ou classe) definido em um módulo pode ser usado em outro módulo é se o módulo definidor o exportar com exportação e o uso do módulo o importar com importação. Os módulos são o assunto do capítulo 10, e a importação e a exportação são abordadas em detalhes no §10.3.

As diretivas de importação são usadas para importar um ou mais valores de outro arquivo de código JavaScript e fornecer nomes no módulo atual. As diretrizes de importação vêm em algumas formas diferentes. Aqui estão alguns exemplos:

```
círculo de importação de './geometry/circle.js'; importar {pi, tau} de './geometry/constants.js'; importar {magnitude como hipotenuse} de './vectors/utils.js';
```

Os valores dentro de um módulo JavaScript são privados e não podem ser importados para outros módulos, a menos que tenham sido exportados explicitamente. A Diretiva de Exportação faz isso: declara que um ou mais valores definidos no módulo atual são exportados e, portanto, disponíveis para importação por outros módulos. A Diretiva de Exportação possui mais variantes do que a Diretiva de Importação. Aqui está um deles:

```
// geometria/constants.js
const pi = math.pi; const
tau = 2 * pi; exportação
{pi, tau};
```

A palavra -chave de exportação às vezes é usada como modificador em outros

Declarações, resultando em um tipo de declaração composta que define uma constante, variável, função ou classe e a exporta ao mesmo tempo. E quando um módulo exporta apenas um valor, isso geralmente é feito com o padrão de exportação de formulário especial:

```
exportar const tau = 2 * math.pi; magnitude da função  
de exportação (x, y) {return math.sqrt (x*x + y*y); }
```

```
Exportar o círculo de classe padrão { /* definição de classe  
omitida aqui */ }
```

5.8 Resumo das declarações JavaScript

Este capítulo introduziu cada uma das declarações da linguagem JavaScript, resumidas na Tabela 5-1.

Tabela 5-1. Declaração JavaScript Sintaxe

Declaração	Propósito
quebrar	Saia do loop ou interruptor mais íntimo ou da declaração de anexo nomeado
caso	Rotule uma declaração dentro de um interruptor
aula	Declarar uma aula
const	Declarar e inicializar uma ou mais constantes
continuar	Comece a próxima iteração do loop mais interno ou o loop nomeado
Depurador	Ponto de interrupção do depurador
padrão	Rotule a instrução padrão em um interruptor
faça/while	Uma alternativa ao loop while

exportar	Declarar valores que podem ser importados para outros módulos
para	Um loop fácil de usar
para/aguardar	Iteram de forma assíncrona os valores de um iterador assíncrono
para/in	Enumerar os nomes de propriedades de um objeto
para/de	Enumerar os valores de um objeto iterável, como uma matriz
função	Declarar uma função
se/else	Executar uma declaração ou outra dependendo de uma condição
importar	Declarar nomes para valores definidos em outros módulos
rótulo	Dê um nome para uso com quebra e continue
deixa r	Declare e inicialize uma ou mais variáveis escassas de blocos (nova sintaxe)
retornar	Retornar um valor de uma função
trocar	Filial Multiway para Case ou Padrão: Rótulos
lançar	Jogue uma exceção
Tente/captura/ final	Lidar com exceções e limpeza de código
“Use rigoroso”	Aplique restrições de modo rigoroso para script ou função
nosso	Declare e inicialize uma ou mais variáveis (sintaxe antiga)
enquanto	Uma construção de loop básico
com	Estender a cadeia de escopo (depreciada e proibida no modo rigoroso)
colheita	Fornecer um valor a ser iterado; usado apenas nas funções do gerador

¹ O fato de as expressões de caso serem avaliadas em tempo de execução torna a declaração JavaScript Switch muito diferente (e menos eficiente que) a instrução Switch

de C, C++ e Java. Nessas linguagens, as expressões case devem ser constantes de tempo de compilação do mesmo tipo, e as instruções switch muitas vezes podem ser compiladas em tabelas de salto altamente eficientes.

2 Quando considerarmos a instrução continue em §5.5.3, veremos que esse loop while não é um equivalente exato do loop for.

Capítulo 6. Objetos

Os objetos são o tipo de dados mais fundamentais do JavaScript, e você já os viu muitas vezes nos capítulos que precedem este. Como os objetos são muito importantes para a linguagem JavaScript, é importante que você entenda como eles funcionam em detalhes, e este capítulo fornece esse detalhe. Começa com uma visão geral formal dos objetos e depois mergulha em seções práticas sobre a criação de objetos e a consulta, definição, exclusão, teste e enumeração das propriedades dos objetos. Essas seções focadas na propriedade são seguidas por seções que explicam como estender, serializar e definir métodos importantes em objetos. Finalmente, o capítulo conclui com uma longa seção sobre a sintaxe literal de novos objetos no ES6 e versões mais recentes do idioma.

6.1 Introdução aos objetos

Um objeto é um valor composto: agrega vários valores (valores primitivos ou outros objetos) e permite armazenar e recuperar esses valores pelo nome. Um objeto é uma coleção não ordenada de propriedades, cada uma com um nome e um valor. Os nomes de propriedades geralmente são strings (embora, como veremos em §6.10.3, os nomes de propriedades também possam ser símbolos), para que possamos dizer que os objetos mapeiam strings para valores. Esse mapeamento de string-a-valor passa por vários nomes- você provavelmente já está familiarizado com a estrutura de dados fundamental sob o nome "hash", "hashtable", "dicionário" ou "matriz associativa". Um objeto é mais do que um mapa simples de string a valor, no entanto. Além de manter

seu próprio conjunto de propriedades, um objeto JavaScript também herda as propriedades de outro objeto, conhecido como seu “protótipo”. Os métodos de um objeto são normalmente propriedades herdadas, e essa “herança prototípica” é um recurso fundamental do JavaScript.

Os objetos JavaScript são dinâmicos – propriedades geralmente podem ser adicionadas e excluídas – mas podem ser usados para simular objetos estáticos e “estruturas” de linguagens de tipo estaticamente. Eles também podem ser usados (ignorando a parte do valor do mapeamento string-to-value) para representar conjuntos de strings.

Qualquer valor em JavaScript que não seja uma string, um número, um símbolo ou verdadeiro, falso, nulo ou indefinido é um objeto. E mesmo que strings, números e booleanos não sejam objetos, eles podem se comportar como objetos imutáveis.

Lembre-se da Seção 3.8 que os objetos são mutáveis e manipulados por referência e não por valor. Se a variável `x` se refere a um objeto e o código `deixa y = x;` é executado, a variável `y` contém uma referência ao mesmo objeto, não uma cópia desse objeto. Quaisquer modificações feitas no objeto através da variável `y` também são visíveis através da variável `x`.

As coisas mais comuns a fazer com objetos são criá-los e definir, consultar, excluir, testar e enumerar suas propriedades. Essas operações fundamentais são descritas nas seções iniciais deste capítulo. As seções seguintes cobrem tópicos mais avançados.

Uma propriedade tem um nome e um valor. Um nome de propriedade pode ser qualquer string, incluindo uma string vazia (ou qualquer símbolo), mas nenhum objeto pode

tem duas propriedades com o mesmo nome. O valor pode ser qualquer valor JavaScript ou pode ser uma função getter ou setter (ou ambas). Aprenderemos sobre funções getter e setter em §6.10.6.

Às vezes é importante ser capaz de distinguir entre propriedades definidas diretamente em um objeto e aquelas que são herdadas de um objeto protótipo. JavaScript usa o termo propriedade própria para se referir a propriedades não herdadas.

Além de seu nome e valor, cada propriedade possui três atributos de propriedade:

- O atributo gravável especifica se o valor da propriedade pode ser definido.
- O atributo enumerable especifica se o nome da propriedade é retornado por um loop for/in.
- O atributo configurável especifica se a propriedade pode ser excluída e se seus atributos podem ser alterados.

Muitos dos objetos internos do JavaScript possuem propriedades somente leitura, não enumeráveis ou não configuráveis. Por padrão, entretanto, todas as propriedades dos objetos criados são graváveis, enumeráveis e configuráveis. §14.1 explica técnicas para especificar valores de atributos de propriedade não padrão para seus objetos.

6.2 Criando Objetos

Os objetos podem ser criados com objetos literais, com a palavra-chave new e com a função Object.create(). As subseções abaixo

descreva cada técnica.

6.2.1 Literais de objeto

A maneira mais fácil de criar um objeto é incluir um objeto literal em seu código JavaScript. Em sua forma mais simples, um objeto literal é uma lista separada por vírgula de nome separado pelo colchão: pares de valor, fechados dentro de aparelhos encaracolados. Um nome de propriedade é um identificador JavaScript ou uma string literal (a string vazia é permitida). Um valor de propriedade é qualquer expressão de JavaScript; O valor da expressão (pode ser um valor primitivo ou um valor de objeto) se torna o valor da propriedade. Aqui estão alguns exemplos:

```
Seja vazio = {};
Propriedades Let Point = {x: 0, y: 0};           // um objeto sem
                                                // dois numéricos
Propriedades Seja P2 = {x: Point.x, y:
Point.y+1};                                     // mais complexo
valores deixam
o livro = {
    &quot;Titulo principal&quot;: &quot;JavaScript&quot;, // estas propriedades
os nomes incluem espaços,
&quot;Sub-título&quot;: &quot;O Guia Definitivo&quot;, // e Hifens,
então use literais de cordas.
    para: &quot;todos o público&quot;,           // for está reservado,
Mas sem citações. Autor: {                      // o valor disso
propriedade é
    primeiro nome: &quot;David&quot;,           // em si um objeto.
Sobrenome: &quot;Flanagan&quot;}                 // legal

};
```

Uma vírgula à direita após a última propriedade em um objeto literal é legal, e alguns estilos de programação incentivam o uso desses traseiros

Vírgulas, portanto, é menos provável que você cause um erro de sintaxe se adicionar uma nova propriedade no final do objeto literal em algum momento posterior.

Um objeto literal é uma expressão que cria e inicializa um objeto novo e distinto cada vez que é avaliado. O valor de cada propriedade é avaliado sempre que o literal é avaliado. Isso significa que um único objeto literal pode criar muitos novos objetos se aparecer dentro do corpo de um loop ou em uma função que é chamada repetidamente e que os valores da propriedade desses objetos podem diferir um do outro.

Os literais do objeto mostrados aqui usam sintaxe simples que tem sido legal desde as primeiras versões do JavaScript. As versões recentes do idioma introduziram uma série de novos recursos literais de objeto, abordados no §6.10.

6.2.2 Criando objetos com novo

O novo operador cria e inicializa um novo objeto. A nova palavra -chave deve ser seguida por uma invocação de função. Uma função usada dessa maneira é chamada de construtor e serve para inicializar um objeto recém -criado. O JavaScript inclui construtores para seus tipos internos. Por exemplo:

```
Seja o = new Object ();      // Crie um objeto vazio: o mesmo que {}.
deixe a = new Array ();      // Crie uma matriz vazia: o mesmo que [].
Seja d = new Date ();        // Crie um objeto de data representando
a hora atual, deixe
r = new Map ();              // Crie um objeto de mapa para chave/valor
mapeamento
```

Além desses construtores embutidos, é comum definir seu

As funções próprias do construtor para inicializar objetos recém - criados. Fazer isso é coberto no capítulo 9.

6.2.3 Protótipos

Antes de podermos cobrir a terceira técnica de criação de objetos, devemos fazer uma pausa por um momento para explicar protótipos. Quase todo objeto JavaScript possui um segundo objeto JavaScript associado a ele. Este segundo objeto é conhecido como protótipo, e o primeiro objeto herda as propriedades do protótipo.

Todos os objetos criados pelos literais do objeto têm o mesmo objeto de protótipo, e podemos nos referir a esse objeto de protótipo no código JavaScript como `object.prototype`. Objetos criados usando a nova palavra -chave e uma invocação de construtor usam o valor da propriedade `Prototype` da função do construtor como seu protótipo. Portanto, o objeto criado por `new Object()` herda do `object.prototype`, assim como o objeto criado por `{}` faz. Da mesma forma, o objeto criado por `new Array()` usa o `Array.prototype` como seu protótipo e o objeto criado por `new date()` usa `date.prototype` como seu protótipo. Isso pode ser confuso ao aprender o JavaScript pela primeira vez. Lembre -se: quase todos os objetos têm um protótipo, mas apenas um número relativamente pequeno de objetos possui uma propriedade de protótipo. São esses objetos com propriedades de protótipo que definem os protótipos para todos os outros objetos.

`Object.prototype` é um dos objetos raros que não possui protótipo: ele não herda nenhuma propriedade. Outros objetos de protótipo são objetos normais que possuem um protótipo. A maioria dos construtores embutidos (e a maioria

construtores definidos pelo usuário) possuem um protótipo que herda de Object.prototype. Por exemplo, Date.prototype herda propriedades de Object.prototype, portanto, um objeto Date criado por new Date() herda propriedades de Date.prototype e Object.prototype. Essa série vinculada de objetos protótipos é conhecida como cadeia de protótipos.

Uma explicação de como funciona a herança de propriedade está em §6.3.2. O Capítulo 9 explica a conexão entre protótipos e construtores com mais detalhes: mostra como definir novas “classes” de objetos escrevendo uma função construtora e definindo sua propriedade protótipo para o objeto protótipo a ser usado pelas “instâncias” criadas com esse construtor. E aprenderemos como consultar (e até alterar) o protótipo de um objeto na Seção 14.3.

6.2.4 Object.create()

Object.create() cria um novo objeto, usando seu primeiro argumento como protótipo desse objeto:

```
deixe o1 = Object.create({x: 1, y: 2});           //o1 herda
propriedades x e y.
o1.x + o1.y // => 3
```

Você pode passar null para criar um novo objeto que não tenha um protótipo, mas se você fizer isso, o objeto recém-criado não herdará nada, nem mesmo métodos básicos como toString() (o que significa que não funcionará com o + operador):

```
deixe o2 = Object.create(null);           // o2 herda não
adereços ou métodos.
```

Se você quiser criar um objeto vazio comum (como o objeto retornado por {} ou new Object()), passe Object.prototype:

```
deixe o3 = Object.create(Object.prototype); // o3 é como {} ou new Object().
```

A capacidade de criar um novo objeto com um protótipo arbitrário é poderosa, e usaremos Object.create() em vários lugares ao longo deste capítulo. (Object.create() também aceita um segundo argumento opcional que descreve as propriedades do novo objeto. Este segundo argumento é um recurso avançado abordado na Seção 14.1.)

Um uso para Object.create() é quando você deseja se proteger contra modificações não intencionais (mas não maliciosas) de um objeto por uma função de biblioteca sobre a qual você não tem controle. Em vez de passar o objeto diretamente para a função, você pode passar um objeto que herda dela. Se a função ler as propriedades desse objeto, ela verá os valores herdados. Se definir propriedades, entretanto, essas gravações não afetarão o objeto original.

```
deixe o = { x: "não altere este valor"; };
biblioteca.function(Object.create(o)); // Protege contra modificações acidentais
```

Para entender por que isso funciona, você precisa saber como as propriedades são consultadas e definidas em JavaScript. Esses são os tópicos da próxima seção.

6.3 Consultando e definindo propriedades

Para obter o valor de uma propriedade, use o ponto (.) ou colchete

([]) operadores descritos em §4.4. O lado esquerdo deve ser uma expressão cujo valor seja um objeto. Se estiver usando o operador ponto, o lado direito deverá ser um identificador simples que nomeie a propriedade. Se estiver usando colchetes, o valor entre colchetes deverá ser uma expressão avaliada como uma string que contém o nome da propriedade desejada:

```
deixe autor = livro.autor;           // Obtém a propriedade "autor";
do livro. deixe nome =
autor.sobrenome;                   // Obtém a propriedade "sobrenome";
do autor. deixe título = livro["título principal"];
// Obtenha a propriedade "título principal" do livro.
```

Para criar ou definir uma propriedade, use um ponto ou colchetes como faria para consultar a propriedade, mas coloque-os no lado esquerdo de uma expressão de atribuição:

```
livro.edição = 7;                  // Cria uma "edição";
propriedade do livro. livro["título
principal"] = "ECMAScript";      // Altera o "principal
propriedade "título".
```

Ao usar a notação de colchetes, dissemos que a expressão entre colchetes deve ser avaliada como uma string. Uma afirmação mais precisa é que a expressão deve ser avaliada como uma string ou um valor que pode ser convertido em uma string ou em um símbolo (§6.10.3). No Capítulo 7, por exemplo, veremos que é comum usar números entre colchetes.

6.3.1 Objetos como matrizes associativas

Conforme explicado na seção anterior, os dois JavaScript a seguir

expressões têm o mesmo valor:

```
objeto.property objeto["propriedade"]
```

A primeira sintaxe, usando o ponto e um identificador, é como a sintaxe usada para acessar um campo estático de uma estrutura ou objeto em C ou Java. A segunda sintaxe, usando colchetes e uma string, parece um acesso a array, mas para um array indexado por strings em vez de números. Esse tipo de array é conhecido como array associativo (ou hash, mapa ou dicionário). Objetos JavaScript são arrays associativos e esta seção explica por que isso é importante.

Em C, C++, Java e linguagens fortemente tipadas semelhantes, um objeto pode ter apenas um número fixo de propriedades, e os nomes dessas propriedades devem ser definidos antecipadamente. Como JavaScript é uma linguagem de tipo flexível, esta regra não se aplica: um programa pode criar qualquer número de propriedades em qualquer objeto. Quando você usa o `.` operador para acessar uma propriedade de um objeto, entretanto, o nome da propriedade é expresso como um identificador. Os identificadores devem ser digitados literalmente em seu programa JavaScript; eles não são um tipo de dados, portanto não podem ser manipulados pelo programa.

Por outro lado, quando você acessa uma propriedade de um objeto com a notação de array `[]`, o nome da propriedade é expresso como uma string. Strings são tipos de dados JavaScript, portanto podem ser manipulados e criados enquanto um programa está em execução. Assim, por exemplo, você pode escrever o seguinte código em JavaScript:

```
deixe addr = ";;
```

```
for(seja i = 0; i < 4; i++) { addr += cliente[`endereço${i}`] +  
"";  
}
```

Este código lê e concatena as propriedades address0, address1, address2 e address3 do objeto cliente.

Este breve exemplo demonstra a flexibilidade de usar notação de array para acessar propriedades de um objeto com expressões de string. Este código poderia ser reescrito usando a notação de ponto, mas há casos em que apenas a notação de array serve. Suponha, por exemplo, que você esteja escrevendo um programa que utiliza recursos de rede para calcular o valor atual dos investimentos do usuário no mercado de ações. O programa permite ao usuário digitar o nome de cada ação que possui, bem como a quantidade de ações de cada ação. Você pode usar um objeto chamado portfólio para armazenar essas informações. O objeto possui uma propriedade para cada ação. O nome da propriedade é o nome da ação e o valor da propriedade é o número de ações dessa ação. Assim, por exemplo, se um usuário detém 50 ações da IBM, a propriedade portfolio.ibm tem o valor 50.

Parte deste programa pode ser uma função para adicionar uma nova ação ao portfólio:

```
function addstock(carteira, stockname, ações) {  
    portfólio[stockname] = ações;  
}
```

Como o usuário insere os nomes das ações em tempo de execução, não há como saber os nomes das propriedades com antecedência. Já que você não pode saber o

Nomes de propriedades Quando você escreve o programa, não há como usar o. Operador para acessar as propriedades do objeto do portfólio. Você pode usar o operador [], no entanto, porque ele usa um valor de string (que é dinâmico e pode mudar em tempo de execução) em vez de um identificador (que é estático e deve ser codificado no programa) para nomear a propriedade.

No capítulo 5, introduzimos o loop for/in (e veremos novamente em breve, no §6.6). O poder desta declaração JavaScript fica claro quando você considera seu uso com matrizes associativas. Aqui está como você o usaria ao calcular o valor total de um portfólio:

```
function computeValue (portfólio) {let total = 0,0;

    para (deixe estoque no portfólio) {          // para cada estoque em
o portfólio:
        Let ações = portfólio [estoque];           // Obtenha o número de
ações
        Deixe o preço = getQuote (estoque);         // Procure compartilhar
preço
        Total += Ações * Preço;                   // Adicione o valor do estoque a
valor total
    } retornar total;                           // retorna total
valor. }
```

Os objetos JavaScript são comumente usados como matrizes associativas, como mostrado aqui, e é importante entender como isso funciona. No ES6 e posterior, no entanto, a classe de mapa descrita no §11.1.2 geralmente é uma escolha melhor do que usar um objeto simples.

6.3.2 Herança

Os objetos JavaScript possuem um conjunto de "propriedades próprias" e também herdam um conjunto de propriedades do objeto de protótipo. Para entender isso, devemos considerar o acesso à propriedade com mais detalhes. Os exemplos nesta seção usam a função `Object.create()` para criar objetos com protótipos especificados. Veremos no capítulo 9, no entanto, que toda vez que você cria uma instância de uma classe com nova, está criando um objeto que herda as propriedades de um objeto de protótipo.

Suponha que você consulte a propriedade `x` no objeto `o`. Se `O` não tiver uma propriedade própria com esse nome, o objeto de protótipo de `O` será consultado para a propriedade `x`. Se o objeto protótipo não tiver uma propriedade própria com esse nome, mas possui um protótipo, a consulta será realizada no protótipo do protótipo. Isso continua até que a propriedade `X` seja encontrada ou até que um objeto com um protótipo nulo seja pesquisado. Como você pode ver, o atributo de protótipo de um objeto cria uma cadeia ou lista vinculada a partir da qual as propriedades são herdadas:

```
Seja o = {};           // o herda métodos de objeto de
Object.prototype.ox = 1; // e agora tem uma propriedade própria
x. Seja p = Object.create(o); // p herda
propriedades de O e objeto.prototype
py = 2;               // e tem uma propriedade própria y.
Seja q = Object.create(p); // Q herda propriedades de P, O e ...
qz = 3;               // ... Object.prototype e tem um
Propriedade própria z.
Seja f = q.toString(); // ToString é herdado de
Object.prototype.QX + QY
// > 3; X e Y são herdados de
o e p
```

Agora, suponha que você atribua à propriedade X do objeto o. Se o O já possui uma propriedade própria (não herdada) chamada X, a tarefa simplesmente altera o valor desta propriedade existente. Caso contrário, a tarefa cria uma nova propriedade chamada x no objeto o. Se o herdou anteriormente a propriedade X, essa propriedade herdada agora está oculta pela propriedade própria criada com o mesmo nome.

A atribuição de propriedade examina a cadeia de protótipo apenas para determinar se a tarefa é permitida. Se o O herdar uma propriedade somente leitura chamada X, por exemplo, a atribuição não será permitida. (Detalhes sobre quando uma propriedade pode ser definida estão em §6.3.3.) Se a atribuição for permitida, no entanto, ela sempre cria ou define uma propriedade no objeto original e nunca modifica objetos na cadeia de protótipos. O fato de que a herança ocorre ao consultar propriedades, mas não quando configurá-las é um recurso essencial do JavaScript, pois nos permite substituir seletivamente as propriedades herdadas:

```
Seja unitcircle = {r: 1};           // um objeto para herdar
de let c = object.create (unitcircle); // c herda
a propriedade r

cx = 1; cy = 1;                   // c define dois
propriedades de seu próprio CR = 2; // c
substitui sua propriedade herdada

unitcircle.r                      // > 1: o protótipo é
não afetado
```

Há uma exceção à regra de que uma atribuição de propriedade falha ou cria ou define uma propriedade no objeto original. Se o herdar a propriedade x e essa propriedade é uma propriedade acessadora com um método de setter (ver §6.10.6), esse método do setter é chamado e não

criando uma nova propriedade x em o. Observe, no entanto, que o método do setter é chamado no objeto O, não no objeto de protótipo que define a propriedade; portanto, se o método do setter definir quaisquer propriedades, ele o fará em O e deixará novamente a cadeia de protótipo não modificada .

6.3.3 Erros de acesso à propriedade

Expressões de acesso à propriedade nem sempre retornam ou definem um valor. Esta seção explica as coisas que podem dar errado ao consultar ou definir uma propriedade.

Não é um erro consultar uma propriedade que não existe. Se a propriedade X não for encontrada como uma propriedade própria ou uma propriedade herdada de O, a expressão de acesso à propriedade Ox avaliará como indefinido. Lembre-se de que nosso objeto de livro possui uma propriedade de "subtítulo", mas não uma propriedade "legenda":

```
Book.subtitle // => indefinido: a propriedade não existe
```

É um erro, no entanto, tentar consultar uma propriedade de um objeto que não existe. Os valores nulos e indefinidos não têm propriedades e é um erro consultar propriedades desses valores. Continuando o exemplo anterior:

```
Seja len = book.subtitle.length; //! TypeError: indefinido não tem comprimento
```

As expressões de acesso à propriedade falharão se o lado esquerdo do é nulo ou indefinido. Portanto, ao escrever uma expressão como o livro. Aqui estão

duas maneiras de se proteger contra esse tipo de problema:

```
// Uma técnica detalhada e explícita let sobrenome = indefinido;  
  
se (livro) {  
    if (livro.autor) {  
        sobrenome = livro.autor.sobrenome; }  
  
}  
  
// Uma alternativa concisa e idiomática para obter sobrenome ou  
nulo ou indefinido  
sobrenome = livro && livro.autor && livro.autor.sobrenome;
```

Para entender por que essa expressão idiomática funciona para evitar exceções `TypeError`, você pode querer revisar o comportamento de curto-circuito do operador `&&` em §4.10.1.

Conforme descrito em §4.4.1, ES2020 suporta acesso condicional à propriedade com `?.`, o que nos permite reescrever a expressão de atribuição anterior como:

```
deixe sobrenome = livro?.autor?.sobrenome;
```

A tentativa de definir uma propriedade como nula ou indefinida também causa um `TypeError`. As tentativas de definir propriedades em outros valores nem sempre são bem-sucedidas: algumas propriedades são somente leitura e não podem ser definidas, e alguns objetos não permitem a adição de novas propriedades. No modo estrito (§5.6.3), um `TypeError` é lançado sempre que uma tentativa de definir uma propriedade falha. Fora do modo estrito, essas falhas geralmente são silenciosas.

As regras que especificam quando uma atribuição de propriedade é bem-sucedida e quando falha são intuitivas, mas difíceis de expressar de forma concisa. Uma tentativa de definir

uma propriedade p de um objeto o falha nestas circunstâncias:

- o possui uma propriedade própria p que é somente leitura: não é possível definir propriedades somente leitura.
- o possui uma propriedade herdada p que é somente leitura: não é possível ocultar uma propriedade herdada somente leitura com uma propriedade própria de mesmo nome.
- o não possui imóvel próprio p; o não herda uma propriedade p com um método setter, e o atributo extensível de o (ver §14.2) é falso. Como p ainda não existe em o, e se não houver nenhum método setter para chamar, então p deve ser adicionado a o. Mas se o não for extensível, então nenhuma nova propriedade poderá ser definida nele.

6.4 Excluindo Propriedades

O operador delete (§4.13.4) remove uma propriedade de um objeto. Seu único operando deve ser uma expressão de acesso à propriedade. Surpreendentemente, delete não opera no valor da propriedade, mas na própria propriedade:

```
excluir livro.autor;           // O objeto book agora não tem
propriedade do autor. deletar
livro["título principal"]; // Agora não tem "main
título"; também.
```

O operador delete exclui apenas propriedades próprias, não herdadas. (Para excluir uma propriedade herdada, você deve excluí-la do objeto protótipo no qual ela está definida. Isso afeta todos os objetos que herdam desse protótipo.)

Uma expressão de exclusão avalia a verdadeira se a exclusão for bem - sucedida ou se a exclusão não tiveram efeito (como excluir uma propriedade inexistente). O Exclue também avalia como verdadeiro quando usado (sem sentido) com uma expressão que não é uma expressão de acesso à propriedade:

```
Seja o = {x: 1};      // o tem propriedade própria x e herda
Propriedade ToString
Excluir ox           // => true: exclui a propriedade x
delete o.x           // => true: nada (x não existe)
Mas é verdade de qualquer
maneira excluir o.ToString
                           // => true: não faz nada (a tostragem não é
uma propriedade
própria) Exclua 1     // => true: absurdo, mas verdade de qualquer maneira
```

A exclusão não remove as propriedades que possuem um atributo configurável de false. Certas propriedades dos objetos internos não são confundíveis, assim como as propriedades do objeto global criado por declaração variável e declaração de função. No modo rigoroso, tentar excluir uma propriedade não configurável causa um TypeError. No modo não rigoroso, excluir simplesmente avalia como false neste caso:

```
// No modo rigoroso, todas essas deleções jogam TypeError
em vez de retornar falsas
excluir object.prototype // => false: propriedade não é configurável

era x = 1;              // declarar uma variável global
exclua globalthis.x     // => false: não posso excluir isso
Função da
propriedade F () {}     // declarar uma função global
exclua globalthis.f     // => false: não posso excluir isso
propriedade também
```

Ao excluir propriedades configuráveis do objeto global no modo não rigoroso, você pode omitir a referência ao objeto global e simplesmente seguir o operador de exclusão com o nome da propriedade:

```
globalThis.x = 1;           // Cria um global configurável
propriedade (sem let ou   // => true: esta propriedade pode ser
var) excluir x           // excluído
```

No modo estrito, entretanto, delete gera um SyntaxError se seu operando for um identificador não qualificado como x, e você precisa ser explícito sobre o acesso à propriedade:

```
excluir x;                // SyntaxError em modo estrito
exclua globalThis.x;       //Isso funciona
```

6.5 Propriedades de Teste

Os objetos JavaScript podem ser considerados conjuntos de propriedades e muitas vezes é útil poder testar a participação no conjunto – para verificar se um objeto possui uma propriedade com um determinado nome. Você pode fazer isso com o operador in, com hasOwnProperty() e

propertyIsEnumerable() ou simplesmente consultando a propriedade. Todos os exemplos mostrados aqui usam strings como nomes de propriedades, mas também funcionam com Símbolos (§6.10.3).

O operador in espera um nome de propriedade no lado esquerdo e um objeto no lado direito. Ele retorna verdadeiro se o objeto tiver uma propriedade própria ou uma propriedade herdada com esse nome:

```
seja o = { x: 1 };
" x " em o // => true: o possui uma propriedade própria " x ";
" y " em o // => false: o não possui a propriedade " y ";
" toString " // => true: o herda uma propriedade toString
; em o
```

O método HASOWNPROPERTY () de um objeto testa se esse objeto possui uma propriedade própria com o nome fornecido. Retorna falsa para propriedades herdadas:

```
Seja o = {x: 1}; o.HasownProperty  
(&quot;X&quot;) // =&gt; true: o tem um próprio  
Propriedade x o.HasownProperty  
(&quot;Y&quot;) // =&gt; false: o não tem um  
Propriedade y o.HasownProperty (&quot;ToString&quot;)  
// =&gt; False: ToString é uma propriedade herdada
```

O PropertyIsEnumerable () refina o teste HASOWNPROPERTY (). Ele retorna verdadeiro apenas se a propriedade nomeada for uma propriedade própria e seu atributo enumerável for verdadeiro. Certas propriedades embutidas não são enumeráveis. As propriedades criadas pelo código JavaScript normal são enumeráveis, a menos que você tenha usado uma das técnicas mostradas no §14.1 para torná-las que não são inumeráveis.

```
Seja o = {x: 1}; o.Propertyisenumerable  
(&quot;X&quot;) // =&gt; true: o tem um próprio  
propriedade enumerável x  
o.propertyisenumerable (&quot;toString&quot;) // =&gt; false: não é um  
Propriedade object.prototype.propertyisenumerable  
(&quot;toString&quot;) // =&gt; false: não enumerável
```

Em vez de usar o operador no IN, muitas vezes é suficiente simplesmente consultar a propriedade e usar! == para garantir que não seja indefinido:

```
Seja o = {x: 1};  
0x! == indefinido // =&gt; true: o tem uma propriedade x  
0y! == indefinido // =&gt; false: o não tem um  
propriedade y o.ToString! == indefinido // =&gt; true: o herda uma toque
```

propriedade

Há uma coisa que o operador `in` pode fazer que a propriedade simples técnica de acesso mostrada aqui não pode fazer. `in` pode distinguir entre propriedades que não existem e propriedades que existem, mas foram definidas como indefinidas. Considere este código:

```
deixe o = {x: indefinido};      // A propriedade está explicitamente definida como
boi indefinido! ==             // => false: a propriedade existe, mas
indefinido                      // não é definida
é indefinido oy! ==             // => false: propriedade nem sequer
indefinido                      // existe
&quot;x&quot; em o              // => true: a propriedade existe
&quot;y&quot; em o              // => false: a propriedade não
exist delete o.x;               //Exclui a propriedade x
&quot;x&quot; em o              // => false: não existe
não mais
```

6.6 Enumerando Propriedades

Em vez de testar a existência de propriedades individuais, às vezes queremos iterar ou obter uma lista de todas as propriedades de um objeto. Existem algumas maneiras diferentes de fazer isso.

O loop `for/in` foi abordado em §5.4.5. Ele executa o corpo do loop uma vez para cada propriedade enumerável (própria ou herdada) do objeto especificado, atribuindo o nome da propriedade à variável do loop. Os métodos internos que os objetos herdam não são enumeráveis, mas as propriedades que seu código adiciona aos objetos são enumeráveis por padrão. Por exemplo:

```
seja o = {x: 1, y: 2, z: 3};           //Três próprios enumeráveis
propriedades
```

```
o.propertyIsEnumerable('toString') // > false: não  
enumerável para (seja p em o) {  
    // Percorre o  
    propriedades  
    console.log(p);  
    // Imprime x, y e z,  
    mas não paraString }
```

Para evitar a enumeração de propriedades herdadas com for/in, você pode adicionar uma verificação explícita dentro do corpo do loop:

```
for(deixe p em o) { if  
    (!o.hasOwnProperty(p)) continuar; // Pular  
    propriedades herdadas }  
  
for(deixe p em o) { if (typeof o[p] === "function")  
    continue; //Pular tudo  
    métodos }
```

Como alternativa ao uso de um loop for/in, geralmente é mais fácil obter uma matriz de nomes de propriedades para um objeto e, em seguida, percorrer essa matriz com um loop for/of. Existem quatro funções que você pode usar para obter uma matriz de nomes de propriedades:

- `Object.keys()` retorna um array dos nomes das propriedades próprias enumeráveis de um objeto. Não inclui propriedades não enumeráveis, propriedades herdadas ou propriedades cujo nome é um Símbolo (ver §6.10.3).
- `Object.getOwnPropertyNames()` funciona como `Object.keys()` mas também retorna um array de nomes de propriedades próprias não enumeráveis, desde que seus nomes sejam strings.

- `Object.getOwnPropertySymbols ()` retorna propriedades próprias cujos nomes são símbolos, sejam eles enumeráveis ou não.
- `Reflect.ownKeys ()` retorna todos os nomes de propriedades, enumeráveis e não enumeráveis, e as cordas e o símbolo. (Veja §14.6.)

Existem exemplos do uso de `object.Keys ()` com um loop para/de §6.7.

6.6.1 Ordem de enumeração da propriedade

ES6 define formalmente a ordem em que as próprias propriedades de um objeto são enumerados. `Object.Keys ()`, `Object.getOwnPropertyNames ()`, `Object.getOwnPropertySymbols ()`, `Reflect.ownKeys ()` e métodos relacionados como `JSON.stringify ()` todas as propriedades da lista na seguinte ordem, sujeitas a suas próprias restrições adicionais sobre se Eles listam propriedades ou propriedades não enumeráveis cujos nomes são strings ou símbolos:

- Propriedades de string cujos nomes são números inteiros não negativos estão listados primeiro, em ordem numérica do menor ao maior. Esta regra significa que matrizes e objetos semelhantes a matrizes terão suas propriedades enumeradas em ordem.
- Depois que todas as propriedades que se parecem com índices de matriz estão listadas, todas as propriedades restantes com nomes de strings são listadas (incluindo propriedades que parecem números negativos ou números de ponto flutuante). Essas propriedades estão listadas na ordem em que foram adicionadas ao objeto. Para propriedades definidas em um

objeto literal, esta ordem é a mesma ordem em que aparecem no literal.

- Finalmente, as propriedades cujos nomes são objetos Símbolo são listadas na ordem em que foram adicionadas ao objeto.

A ordem de enumeração para o loop for/in não é tão bem especificada quanto para essas funções de enumeração, mas as implementações normalmente enumeram propriedades próprias na ordem descrita e, em seguida, percorrem a cadeia de protótipos enumerando propriedades na mesma ordem para cada objeto protótipo . Observe, entretanto, que uma propriedade não será enumerada se uma propriedade com o mesmo nome já tiver sido enumerada, ou mesmo se uma propriedade não enumerável com o mesmo nome já tiver sido considerada.

6.7 Estendendo Objetos

Uma operação comum em programas JavaScript é copiar as propriedades de um objeto para outro objeto. É fácil fazer isso com um código como este:

```
deixe alvo = {x: 1}, fonte = {y: 2, z: 3};  
for(deixe a chave de Object.keys(fonte)) {  
    alvo[chave] = origem[chave]; }  
  
alvo      // => {x: 1, y: 2, z: 3}
```

Mas como esta é uma operação comum, vários frameworks JavaScript definiram funções utilitárias, geralmente chamadas de extend(), para realizar esta operação de cópia. Finalmente, no ES6, essa capacidade chega ao núcleo da linguagem JavaScript na forma de Object.assign().

`Object.assign ()` espera dois ou mais objetos como seus argumentos. Ele modifica e retorna o primeiro argumento, que é o objeto de destino, mas não altera o segundo ou qualquer argumento subsequente, que são os objetos de origem. Para cada objeto de origem, ele copia as próprias propriedades próprias desse objeto (incluindo aqueles cujos nomes são símbolos) no objeto de destino. Ele processa os objetos de origem na ordem da lista de argumentos, para que as propriedades na primeira fonte substituam as propriedades do mesmo nome no objeto de destino e propriedades no segundo objeto de origem (se houver uma) substituição de substituição com o mesmo nome na primeira fonte objeto.

`Object.assign ()` copia propriedades com o `Ordind Property Get Operations`, por isso, se um objeto de origem tiver um método `getter` ou o objeto de destino tiver um método `Setter`, eles serão invocados durante a cópia, mas não serão copiados.

Um motivo para atribuir propriedades de um objeto a outro é quando você tem um objeto que define valores padrão para muitas propriedades e deseja copiar essas propriedades padrão para outro objeto se uma propriedade com esse nome ainda não existir nesse objeto. Usando `object.assign ()` ingenuamente não fará o que você deseja:

```
object.assign (o, padrões);           // substitui tudo em o
com padrões
```

Em vez disso, o que você pode fazer é criar um novo objeto, copiar os padrões nele e, em seguida, substituir esses padrões com as propriedades em O:

```
o = object.assign ({}, padrões, o);
```

Veremos no §6.10.4 que você também pode expressar esta operação de copiar e substituir o objeto usando o ... Spread Operator como este:

```
o = {... padrão, ... o};
```

Também poderíamos evitar a sobrecarga da criação e cópia extra de objetos escrevendo uma versão do object.assign () que copia propriedades apenas se elas estiverem faltando:

```
// como object.assign (), mas não substitui as
propriedades existentes
// (e também não lida com as propriedades do símbolo) Função
(Target, ... fontes) {for (deixe a fonte de fontes) {

    para (deixe a chave do object.Keys (fonte)) {if (! (chave no
    alvo)) {// isso é diferente de
object.assign ()

    Target [key] = fonte [chave]; }

}} retornar
o destino;

} Object.assign ({x: 1}, {x: 2, y: 2}, {y: 3, z:
4}) // => {x:
2, y: 3, z: 4} mescle ({x: 1}, {x: 2, y: 2}, {y: 3,
z: 4}) // => {x:
1, y: 2, z: 4}
```

É simples escrever outros utilitários de manipulação de propriedades, como esta função mescle (). Uma função RESTRINCI () pode excluir propriedades de um objeto se não aparecerem em outro objeto de modelo, por exemplo. Ou uma função subtract () pode remover todas as propriedades de um objeto de outro objeto.

6.8 Serializando Objetos

A serialização de objetos é o processo de conversão do estado de um objeto em uma string a partir da qual ele pode ser restaurado posteriormente. As funções

`JSON.stringify()` e `JSON.parse()` serializam e restauram objetos JavaScript. Essas funções usam o formato de intercâmbio de dados JSON. JSON significa “JavaScript Object Notation” e sua sintaxe é muito semelhante à dos objetos JavaScript e literais de array:

```
deixe o = {x: 1, y: {z: [falso, nulo, ""]}}; //Definir um
objeto de teste
vamos s = JSON.stringify(o);      //s == '"x":1,"y":[
[falso,null,""]}&   {'z'::
#39; let p = JSON.parse(s); // p == {x: 1, y: {z: [falso,
nulo, ""]}}
```

A sintaxe JSON é um subconjunto da sintaxe JavaScript e não pode representar todos os valores JavaScript. Objetos, matrizes, strings, números finitos, verdadeiro, falso e nulo são suportados e podem ser serializados e restaurados. NaN, Infinity e -Infinity são serializados como nulos. Os objetos de data são serializados em strings de data no formato ISO (consulte a função `Date.toJSON()`), mas `JSON.parse()` os deixa no formato de string e não restaura o objeto Date original. Os objetos Function, RegExp e Error e o valor indefinido não podem ser serializados ou restaurados. `JSON.stringify()` serializa apenas as propriedades próprias enumeráveis de um objeto. Se um valor de propriedade não puder ser serializado, essa propriedade será simplesmente omitida da saída stringificada. Tanto `JSON.stringify()` quanto `JSON.parse()` aceitam segundos argumentos opcionais que podem ser usados para personalizar o processo de serialização e/ou restauração, especificando uma lista de propriedades a serem serializadas, por exemplo, ou convertendo determinados valores durante a serialização ou

processo de estringificação. A documentação completa para essas funções está em §11.6.

6.9 Métodos de Objeto

Conforme discutido anteriormente, todos os objetos JavaScript (exceto aqueles criados explicitamente sem um protótipo) herdam propriedades de

`Objeto.protótipo`. Essas propriedades herdadas são principalmente métodos e, por estarem universalmente disponíveis, são de particular interesse para programadores JavaScript. Já vimos os métodos `hasOwnProperty()` e `propertyIsEnumerable()`, por exemplo. (E também já cobrimos algumas funções estáticas definidas no construtor `Object`, como

`Object.create()` e `Object.keys()`.) Esta seção explica alguns métodos de objetos universais que são definidos em

`Object.prototype`, mas que se destinam a ser substituídos por outras implementações mais especializadas. Nas seções a seguir, mostramos exemplos de definição desses métodos em um único objeto. No Capítulo 9, você aprenderá como definir esses métodos de maneira mais geral para uma classe inteira de objetos.

6.9.1 O método `toString()`

O método `toString()` não aceita argumentos; ele retorna uma string que de alguma forma representa o valor do objeto no qual é invocado. JavaScript invoca esse método de um objeto sempre que precisa converter o objeto em uma string. Isso ocorre, por exemplo, quando você usa o operador `+` para concatenar uma string com um objeto ou quando você passa um objeto para um método que espera uma string.

O método `toString()` padrão não é muito informativo (embora seja útil para determinar a classe de um objeto, como veremos na Seção 14.4.3). Por exemplo, a seguinte linha de código simplesmente é avaliada como a string “[object Object]”:

```
deixe s = { x: 1, y: 1 }.toString();      // s == "Object [object Object]"
```

Como esse método padrão não exibe muitas informações úteis, muitas classes definem suas próprias versões de `toString()`. Por exemplo, quando um array é convertido em uma string, você obtém uma lista dos elementos do array, cada um deles convertido em uma string, e quando uma função é convertida em uma string, você obtém o código-fonte da função. Você pode definir seu próprio método `toString()` assim:

```
deixe ponto = { x: 1, y: 2, toString: function()
{ return `(${this.x}, ${this.y})` }

};

String(ponto)    // > "(1, 2)": toString() é usado para
conversões de string
```

6.9.2 O método `toLocaleString()`

Além do método `toString()` básico, todos os objetos possuem um `toLocaleString()`. O objetivo deste método é retornar uma representação de string localizada do objeto. O método `toLocaleString()` definido por `Object` não faz nenhuma localização: ele simplesmente chama `toString()` e retorna esse valor. As classes `Data` e `Número` definem versões personalizadas de

`toLocalestring()` que tentam formatar números, datas e tempos de acordo com as convenções locais. Array define um

`TOLOCALESTRING()` Método que funciona como `ToString()`, exceto que formata os elementos da matriz chamando seus métodos `toLocalestring()` em vez de seus métodos de `toString()`. Você pode fazer a mesma coisa com um objeto de ponto como este:

```
Let Point = {x: 1000, y: 2000, toString:  
    function () {return `($ {this.x}, $ {this.y})`;  
  
},  
    Tolocalestring: function () {  
        retornar `($ {this.x.toLocalestring ()},  
$ {this.y.toLocalestring ()})`;  
    };  
    Point.ToString ()  
        // => "(1000, 2000)"  
    Point.Tolocalestring ()  
        // => "(1.000, 2.000)": nota  
        milhares de separadores
```

As classes de internacionalização documentadas no §11.7 podem ser úteis ao implementar um método `toLocalestring()`.

6.9.3 o método `ValueOf()`

O método `ValueOf()` é muito parecido com o método `ToString()`, mas é chamado quando o JavaScript precisa converter um objeto para algum tipo primitivo que não seja uma string - tipicamente, um número. O JavaScript chama esse método automaticamente se um objeto for usado em um contexto em que um valor primitivo é necessário. O método padrão de `valueof()` não faz nada interessante, mas algumas das classes internas definem seus próprios

método `valorOf()`. A classe `Date` define `valueOf()` para converter datas em números, e isso permite que objetos `Date` sejam comparados cronologicamente com `<` e `>`. Você poderia fazer algo semelhante com um objeto de ponto, definindo um método `valueOf()` que retorna a distância da origem até o ponto:

```
deixe ponto = { x: 3, y: 4, valorOf: function()
{ return Math.hypot(this.x, this.y);

} }; Número
(ponto)
    // > 5: valueOf() é usado para conversões para
ponto numérico
&gt; 4      // > verdadeiro
ponto &gt; 5  // > falso
ponto &lt; 6  // > verdadeiro
```

6.9.4 O método `toJSON()`

`Object.prototype` na verdade não define um método `toJSON()`, mas o método `JSON.stringify()` (ver §6.8) procura um

`toJSON()` em qualquer objeto que seja solicitado a serializar. Se este método existir no objeto a ser serializado, ele será invocado e o valor de retorno será serializado, em vez do objeto original. A classe `Date` (§11.4) define um método `toJSON()` que retorna uma representação de string serializável da data.

Poderíamos fazer o mesmo para nosso objeto `Point` assim:

```
deixe ponto = { x: 1, y: 2, toString:
function() { return `(${this.x}, ${this.y})`;

},
```

```
toJSON: function() { return this.toString(); } };

JSON.stringify([ponto])      // => '["(1, 2)"]'
```

6.10 Sintaxe Literal de Objeto Estendido

Versões recentes do JavaScript estenderam a sintaxe para objetos literais de diversas maneiras úteis. As subseções a seguir explicam essas extensões.

6.10.1 Propriedades abreviadas

Suponha que você tenha valores armazenados nas variáveis x e y e queira criar um objeto com propriedades denominadas x e y que contenham esses valores. Com a sintaxe literal de objeto básica, você acabaria repetindo cada identificador duas vezes:

```
seja x = 1, y = 2; seja o = {
  x: x, y: y
};
```

No ES6 e posterior, você pode eliminar os dois pontos e uma cópia do identificador e obter um código muito mais simples:

```
seja x = 1, y = 2;
seja o = {x, y};
boi + oi // => 3
```

6.10.2 Nomes de propriedades computadas

Às vezes você precisa criar um objeto com uma propriedade específica, mas o nome dessa propriedade não é uma constante de tempo de compilação que você possa

digite literalmente seu código-fonte. Em vez disso, o nome da propriedade necessária é armazenado em uma variável ou é o valor de retorno de uma função que você invoca. Você não pode usar um literal de objeto básico para esse tipo de propriedade. Em vez disso, você precisa criar um objeto e adicionar as propriedades desejadas como uma etapa extra:

```
const PROPERTY_NAME = "p1"; function computaPropertyName()
{ return "p" + 2; }

deixe o = {};
o[PROPERTY_NAME] = 1;
o[computaPropertyName()] = 2;
```

É muito mais simples configurar um objeto como este com um recurso ES6 conhecido como propriedades computadas que permite pegar os colchetes do código anterior e movê-los diretamente para o literal do objeto:

```
const PROPERTY_NAME = "p1"; função computaPropertyName() {
return "p" + 2; }

deixe p = {[PROPERTY_NAME]:
1, [computaPropertyName()]: 2

};

p.p1 + p.p2 // => 3
```

Com esta nova sintaxe, os colchetes delimitam uma expressão JavaScript arbitrária. Essa expressão é avaliada e o valor resultante (convertido em uma string, se necessário) é usado como nome da propriedade.

Uma situação em que você pode querer usar propriedades computadas é quando você tem uma biblioteca de código JavaScript que espera receber objetos com um conjunto específico de propriedades e os nomes desses objetos.

propriedades são definidas como constantes nessa biblioteca. Se você estiver escrevendo código para criar os objetos que serão passados para essa biblioteca, você poderá codificar os nomes das propriedades, mas correrá o risco de erros se digitar o nome da propriedade errado em qualquer lugar e correrá o risco de problemas de incompatibilidade de versão se um novo versão da biblioteca altera os nomes das propriedades necessárias. Em vez disso, você pode achar que seu código fica mais robusto ao usar a sintaxe de propriedade computada com as constantes de nome de propriedade definidas pela biblioteca.

6.10.3 Símbolos como Nomes de Propriedades

A sintaxe da propriedade computada habilita outro recurso literal de objeto muito importante. No ES6 e posterior, os nomes das propriedades podem ser strings ou símbolos. Se você atribuir um símbolo a uma variável ou constante, poderá usar esse símbolo como um nome de propriedade usando a sintaxe de propriedade computada:

```
const extension = Symbol("meu símbolo de extensão"); seja
o = {
  [extensão]: { /* dados de extensão armazenados
  neste objeto */ } };
o[extensão].x = 0; // Isso não entrará em conflito com outras
propriedades de o
```

Conforme explicado em §3.6, os símbolos são valores opacos. Você não pode fazer nada com eles além de usá-los como nomes de propriedades. Cada símbolo é diferente de todos os outros símbolos, entretanto, o que significa que os símbolos são bons para criar nomes de propriedades exclusivos. Crie um novo símbolo chamando a função de fábrica `Symbol()`. (Símbolos são valores primitivos, não objetos, então `Symbol()` não é um construtor

função que você chama com o novo.) O valor retornado pelo símbolo () não é igual a nenhum outro símbolo ou outro valor. Você pode passar uma string para símbolo () e essa string é usada quando seu símbolo é convertido em uma string. Mas este é apenas uma ajuda de depuração: dois símbolos criados com o mesmo argumento de string ainda são diferentes um do outro.

O ponto dos símbolos não é segurança, mas definir um mecanismo de extensão seguro para objetos JavaScript. Se você receber um objeto de código de terceiros que não controla e precisar adicionar algumas de suas próprias propriedades a esse objeto, mas deseja ter certeza de que suas propriedades não entrarão em conflito com nenhuma propriedade que já exista no objeto, você pode usar com segurança símbolos como nomes de propriedades. Se você fizer isso, também pode ter certeza de que o código de terceiros não alterará acidentalmente suas propriedades simbolicamente nomeadas. (Esse código de terceiros poderia, é claro, usar object.getOwnPropertySymbols () para descobrir os símbolos que você está usando e poderia alterar ou excluir seu

propriedades. É por isso que os símbolos não são um mecanismo de segurança.)

6.10.4 Operador de espalhamento

No ES2018 e mais tarde, você pode copiar as propriedades de um objeto existente em um novo objeto usando o "operador de espalhamento" ... dentro de um objeto literal:

```
deixe a posição = {x: 0, y: 0}; Let Dimensions = {Width: 100,  
Hight: 75}; Deixe rect = {... posicionar, ... dimensões};  
rect.x + ret.y + ret.width + ret.height // => 175
```

Neste código, as propriedades da posição e dimensões

Os objetos são "espalhados" para o objeto Rect literal como se tivessem sido escritos literalmente dentro desses aparelhos encaracolados. Observe que essa sintaxe é frequentemente chamada de operadora de spread, mas não é um verdadeiro operador JavaScript em nenhum sentido. Em vez disso, é uma sintaxe de caso especial disponível apenas nos literais de objetos. (Três pontos são usados para outros propósitos em outros contextos de JavaScript, mas os literais de objetos são o único contexto em que os três pontos causam esse tipo de interpolação de um objeto em outro.)

Se o objeto que está espalhado e o objeto em que ele está sendo espalhado tiver uma propriedade com o mesmo nome, o valor dessa propriedade será o que vem por último:

```
Seja o = {x: 1}; Seja p = {x: 0, ... o}; px // => 1: o valor do objeto o substitui o valor inicial
```

```
Seja q = {... o, x: 2}; qx // => 2: 0 valor 2 substitui o valor anterior de o.
```

Observe também que o operador de spread espalhe apenas as próprias propriedades de um objeto, e não herdadas:

```
Seja o = object.create ({x: 1}); // o herda a propriedade x deixa p = {... o}; px // => indefinido
```

Por fim, vale a pena notar que, embora o operador de spread seja apenas três pequenos pontos em seu código, ele pode representar uma quantidade substancial de trabalho para o intérprete JavaScript. Se um objeto tiver n propriedades, o processo de espalhar essas propriedades em outro objeto provavelmente será

uma operação O (n). Isso significa que, se você se encontrar usando ... dentro de um loop ou função recursiva como uma maneira de acumular dados em um objeto grande, você pode estar escrevendo um algoritmo O (n^2) ineficiente que não será escrito bem como n maior.

6.10.5 Métodos de abreviação

Quando uma função é definida como uma propriedade de um objeto, chamamos essa função de método (teremos muito mais a dizer sobre os métodos nos capítulos 8 e 9). Antes do ES6, você definiria um método em um objeto literal usando uma expressão de definição de função, assim como definiria qualquer outra propriedade de um objeto:

```
Let Square = {Area: function () {return  
this.side * thisside; }, lado: 10  
}; square.area () // => 100
```

No ES6, no entanto, a sintaxe literal do objeto (e também a sintaxe da definição de classe que veremos no Capítulo 9) foi estendida para permitir um atalho onde a palavra -chave da função e o colón são omitidos, resultando em código como este:

```
Let Square = {Area () {return  
this.side * thisside; }, lado: 10  
}; square.area () // => 100
```

Ambas as formas do código são equivalentes: ambos adicionam uma propriedade denominada área ao objeto literal, e ambos definem o valor dessa propriedade para o

função especificada. A sintaxe abreviada deixa mais claro que a área () é um método e não uma propriedade de dados como o lado.

Quando você escreve um método usando essa sintaxe abreviada, o nome da propriedade pode assumir qualquer um dos formulários que são legais em um objeto literal: além de um identificador javascript regular, como a área de nome acima, você também pode usar literais de string e nomes de propriedades computadas e nomes de propriedades , que pode incluir nomes de propriedades de símbolo:

```
const métod_name = "m";
const símbolo = símbolo ();
Deixe
WeirdMethods = {

    "Método com espaços" (x) {return x + 1; },
    [Method_Name] (x) {return x + 2; },
    [símbolo] (x) {return x + 3; };

}

WeirdMethods ["Método com espaços"] (1) // => 2
WeirdMethods [Method_Name] (1)           // => 3
Weirdmethods [símbolo] (1)              // => 4
```

Usar um símbolo como nome de método não é tão estranho quanto parece. Para tornar um objeto iterável (para que ele possa ser usado com um loop for/de), você deve definir um método com o símbolo de nome simbólico.iterator, e há exemplos de fazer exatamente isso no Capítulo 12.

6.10.6 Getters de propriedades e setters

Todas as propriedades do objeto que discutimos até agora neste capítulo foram propriedades de dados com um nome e um valor comum. O JavaScript também suporta propriedades de acessador, que não possuem um único valor, mas, em vez disso, possuem um ou dois métodos de acessórios: um getter e/ou um setter.

Quando um programa consulta o valor de uma propriedade do acessador, o JavaScript invoca o método getter (sem passar argumentos). O valor de retorno deste método torna-se o valor da expressão de acesso à propriedade. Quando um programa define o valor de uma propriedade do acessador, o JavaScript invoca o método setter, passando o valor do lado direito da atribuição. Este método é responsável por “definir”, em certo sentido, o valor da propriedade. O valor de retorno do método setter é ignorado.

Se uma propriedade tiver um método getter e um método setter, ela será uma propriedade de leitura/gravação. Se tiver apenas um método getter, é uma propriedade somente leitura. E se tiver apenas um método setter, é uma propriedade somente de gravação (algo que não é possível com propriedades de dados), e as tentativas de lê-la sempre são avaliadas como indefinidas.

As propriedades do acessador podem ser definidas com uma extensão para a sintaxe literal do objeto (ao contrário de outras extensões ES6 que vimos aqui, getters e setters foram introduzidos no ES5):

```
let o = { // Uma propriedade de
dados comum dataProp: value,
          // Uma propriedade do acessador definida como um par de
          // funções. obter accessorProp() { retornar this.dataProp; },
          definir accessorProp (valor) { this.dataProp = valor; } };
```

As propriedades do acessador são definidas como um ou dois métodos cujo nome é igual ao nome da propriedade. Eles se parecem com métodos comuns definidos usando a abreviação ES6, exceto que as definições de getter e setter são prefixadas com get ou set. (No ES6, você também pode usar

Nomes de propriedades calculados ao definir getters e setters. Basta substituir o nome da propriedade após o Get ou Set com uma expressão entre colchetes.)

Os métodos acessadores definidos acima simplesmente obtêm e definem o valor de uma propriedade de dados, e não há razão para preferir a propriedade Acessor à propriedade de dados. Mas, como um exemplo mais interessante, considere o seguinte objeto que representa um ponto cartesiano 2D. Possui propriedades de dados comuns para representar as coordenadas X e Y do ponto, e possui propriedades de acessórios que fornecem as coordenadas polares equivalentes ao ponto:

Seja p = { // x e y são propriedades de dados de leitura de leitura regulares. x: 1.0, y: 1.0,

```
// R é uma propriedade de acessador de leitura-write com getter e setter.  
// Não se esqueça de colocar uma vírgula após métodos de acessórios. obtenha r () {return math.hypot (this.x, this.y); },  
set r (newValue) {  
    Seja OldValue = Math.HyPot (this.x, this.y);  
    Deixe a proporção = newValue/OldValue;  
    this.x *= proporção;  
    this.y *= razão;  
},  
  
// Theta é uma propriedade acessadora somente leitura apenas com getter.  
obtenha theta () {return Math.atan2 (this.y, this.x); }; pr // => math.sqrt2  
  
p.theta // => math.pi / 4
```

Observe o uso da palavra -chave isso nos getters e Setter neste

exemplo. O JavaScript chama essas funções como métodos do objeto em que são definidos, o que significa que, dentro do corpo da função, isso se refere ao objeto Point P. Portanto, o método getter para a propriedade R pode se referir às propriedades X e Y como `this.x` e `this.y`. Os métodos e a palavra -chave são abordados em mais detalhes no §8.2.2.

As propriedades do acessador são herdadas, assim como as propriedades de dados, para que você possa usar o objeto P definido acima como um protótipo para outros pontos. Você pode dar aos novos objetos suas próprias propriedades X e Y, e eles herdarão as propriedades R e Theta:

```
Seja q = object.create (p); // Um novo objeto que herda getters e
setters
qx = 3; QY = 4;           // Crie propriedades de dados de Q
qr                         // => 5: o acessador herdado
Trabalho de propriedades Q.Theta // => Math.atan2 (4, 3)
```

O código acima usa propriedades de acessador para definir uma API que fornece duas representações (coordenadas cartesianas e coordenadas polares) de um único conjunto de dados. Outros motivos para usar as propriedades do acessador incluem a verificação da sanidade das gravações de propriedades e o retorno de valores diferentes em cada propriedade Leia:

```
// Este objeto gera números de série estritamente crescentes const
serialnum = {
    // Esta propriedade de dados contém o próximo número de
    // série. // _ no nome da propriedade sugere que é para
    // somente uso interno.
    _n: 0,
    // retorna o valor atual e o incremento, obtém a seguir () {return
    // this._n ++; },
```

```

// Defina um novo valor de n, mas apenas se for maior que o
atual
    Defina o próximo (n) {
        if (n> this._n) this._n = n; caso contrário, jogue
        novo erro ("o número de série só pode ser definido
para um valor maior ");
    } }; serialnum.next
= 10;                                // Defina o número de série inicial
serialnum.next                         // => 10
serialnum.next                         // => 11: Valor diferente a cada vez
nós chegamos a seguir

```

Finalmente, aqui está mais um exemplo que usa um método getter para implementar uma propriedade com comportamento "mágico":

```

// Este objeto possui propriedades acessadoras que retornam
números aleatórios.
// A expressão "Random.octet", por exemplo,
produz um número aleatório
// entre 0 e 255 cada vez que é avaliado. const aleatoriamente = {

    obtenha Octet () {return Math.floor (Math.random
    ()*256); }, obtenha uint16 () {return math.floor
    (math.random ()*65536); }, obtenha int16 () {retorna
    Math.Floor (Math.Random ()*65536) -32768; }};

```

6.11 Resumo

Este capítulo documentou objetos JavaScript em grandes detalhes, abrangendo tópicos que incluem:

- Terminologia básica de objetos, incluindo o significado de termos como propriedades enumeráveis e próprias.
- Sintaxe literal de objetos, incluindo muitos novos recursos no ES6

e mais tarde.

- Como ler, escrever, deletar, enumerar e verificar a presença das propriedades de um objeto.
- Como funciona a herança baseada em protótipo em JavaScript e como criar um objeto que herda de outro objeto com `Object.create()`.
- Como copiar propriedades de um objeto para outro com `Object.assign()`.

Todos os valores JavaScript que não são valores primitivos são objetos. Isso inclui arrays e funções, que são os tópicos dos próximos dois capítulos.

¹ Lembrar; quase todos os objetos possuem um protótipo, mas a maioria não possui uma propriedade chamada protótipo. A herança JavaScript funciona mesmo se você não puder acessar o objeto protótipo diretamente. Mas veja §14.3 se quiser aprender como fazer isso.

Capítulo 7. Matrizes

Este capítulo documenta arrays, um tipo de dados fundamental em JavaScript e na maioria das outras linguagens de programação. Uma matriz é uma coleção ordenada de valores. Cada valor é chamado de elemento e cada elemento possui uma posição numérica na matriz, conhecida como índice. Os arrays JavaScript não são digitados: um elemento do array pode ser de qualquer tipo e diferentes elementos do mesmo array podem ser de tipos diferentes. Os elementos do array podem até ser objetos ou outros arrays, o que permite criar estruturas de dados complexas, como arrays de objetos e arrays de arrays. As matrizes JavaScript são baseadas em zero e usam índices de 32 bits: o índice do primeiro elemento é 0 e o índice mais alto possível é 4294967294 ($2^{32} - 2$), para um tamanho máximo de matriz de 4.294.967.295 elementos. Os arrays JavaScript são dinâmicos: eles aumentam ou diminuem conforme necessário e não há necessidade de declarar um tamanho fixo para o array ao criá-lo ou de realocá-lo quando o tamanho muda. Matrizes JavaScript podem ser esparsas: os elementos não precisam ter índices contíguos e podem haver lacunas. Cada array JavaScript possui uma propriedade `length`. Para matrizes não esparsas, esta propriedade especifica o número de elementos na matriz. Para matrizes esparsas, o comprimento é maior que o índice mais alto de qualquer elemento.

Matrizes JavaScript são uma forma especializada de objeto JavaScript, e os índices de matriz são, na verdade, pouco mais do que nomes de propriedades que são inteiros. Falaremos mais sobre as especializações de arrays em outras partes deste capítulo. As implementações normalmente otimizam matrizes para que o acesso aos elementos da matriz indexados numericamente seja geralmente significativo.

mais rápido que o acesso a propriedades regulares de objetos.

As matrizes herdam propriedades do `Array.prototype`, que define um rico conjunto de métodos de manipulação de matriz, cobertos no §7.8. A maioria desses métodos é genérica, o que significa que eles funcionam corretamente não apenas para matrizes verdadeiras, mas para qualquer “objeto semelhante a uma matriz”. Discutiremos objetos semelhantes a matrizes no §7.9. Finalmente, as cordas JavaScript se comportam como matrizes de personagens, e discutiremos isso no §7.10.

O ES6 apresenta um conjunto de novas classes de matriz conhecidas coletivamente como “matrizes digitadas”. Ao contrário das matrizes JavaScript regulares, as matrizes digitadas têm um comprimento fixo e um tipo de elemento numérico fixo. Eles oferecem acesso de alto desempenho e bytes a dados binários e são abordados no §11.2.

7.1 Criando matrizes

Existem várias maneiras de criar matrizes. As subseções a seguir explicam como criar matrizes com:

- Matriz literais
- O ... Spread Operator em um objeto iterável
- O construtor da matriz ()
- Os métodos de fábrica do `Array.Of()` e `Array.From()`

7.1.1 Literais da matriz

De longe, a maneira mais simples de criar uma matriz é com uma matriz literal, que é simplesmente uma lista separada por vírgula de elementos de matriz entre colchetes. Por exemplo:

```
deixe vazio = []; // Um array sem elementos
sejam primos = [2, 3, 5, 7, 11]; // Um array com 5 numéricos
elementos let misc = [1.1, true, "a", ,];
// 3 elementos de vários tipos + vírgula final
```

Os valores em um array literal não precisam ser constantes; eles podem ser expressões arbitrárias:

```
seja base = 1024; deixe tabela = [base, base+1, base+2, base+3];
```

Literais de array podem conter literais de objeto ou outros literais de array:

```
seja b = [[1, {x: 1, y: 2}], [2, {x: 3, y: 4}]];
```

Se um array literal contém múltiplas vírgulas em uma linha, sem nenhum valor entre eles, o array é esparso (veja §7.3). Os elementos da matriz para os quais os valores são omitidos não existem, mas parecem indefinidos se você os consultar:

```
vamos contar = [1,,3]; // Elementos nos índices 0 e 2. Nenhum elemento no índice 1 let undefs = [,,];
// Um array sem elementos, mas com comprimento 2
```

A sintaxe literal da matriz permite uma vírgula final opcional, portanto [,,] tem comprimento de 2, não de 3.

7.1.2 O Operador de Spread

No ES6 e posteriores, você pode usar o “operador spread,” ..., para incluir os elementos de um array dentro de um literal de array:

```
seja a = [1, 2, 3];
seja b = [0, ...a, 4]; // b == [0, 1, 2, 3, 4]
```

Os três pontos “espalham” o array a para que seus elementos se tornem elementos dentro do literal do array que está sendo criado. É como se ...a fosse substituído pelos elementos do array a, listados literalmente como parte do literal do array envolvente. (Observe que, embora chamemos esses três pontos de operador de propagação, este não é um operador verdadeiro porque só pode ser usado em literais de array e, como veremos mais adiante neste livro, em invocações de funções.)

O operador spread é uma maneira conveniente de criar uma cópia (superficial) de um array:

```
deixe original = [1,2,3]; deixe copiar =
[...original]; copiar[0] = 0; //Modificar a
cópia não altera o original

original[0] // => 1
```

O operador spread funciona em qualquer objeto iterável. (Objetos iteráveis são sobre os quais o loop for/of itera; nós os vimos pela primeira vez em §5.4.4 e veremos muito mais sobre eles no Capítulo 12.) Strings são iteráveis, então você pode usar um operador spread para transformar qualquer string em uma matriz de strings de um único caractere:

```
deixe dígitos = [..."0123456789ABCDEF"];
dígitos // =>
["0","1","2","3","4","5","6","7","8","9","A","B","C","D","E","F"]
```

Objetos de conjunto (§11.1.1) são iteráveis, portanto, uma maneira fácil de remover elementos duplicados de um array é converter o array em um conjunto e então

Converta imediatamente o conjunto em uma matriz usando o operador de espalhamento:

```
Deixe letras = [... "Hello World"]; [... novo conjunto  
(letras)] // => ["h", "e", "l",  
"o", " ", "w", "r",  
"d"]
```

7.1.3 O construtor Array ()

Outra maneira de criar uma matriz é com o construtor Array (). Você pode invocar este construtor de três maneiras distintas:

- Chame sem argumentos:

```
deixe a = new Array ();
```

Este método cria uma matriz vazia sem elementos e é equivalente à matriz literal [].

- Chame com um único argumento numérico, que especifica um comprimento:

```
deixe a = nova matriz (10);
```

Esta técnica cria uma matriz com o comprimento especificado. Esta forma do construtor Array () pode ser usada para pré -alocar uma matriz quando você souber com antecedência quantos elementos serão necessários. Observe que nenhum valores é armazenado na matriz e as propriedades do índice da matriz "0", "1" e assim por diante nem são definidas para a matriz.

- Especifique explicitamente dois ou mais elementos de matriz ou um único elemento não numérico para a matriz:

```
deixe a = nova matriz (5, 4, 3, 2, 1,
```

```
"testando, testando");
```

Nesta forma, os argumentos do construtor tornam-se os elementos do novo array. Usar um array literal é quase sempre mais simples do que usar o construtor Array().

7.1.4 Matriz.de()

Quando a função construtora Array() é invocada com um argumento numérico, ela usa esse argumento como um comprimento de array. Mas quando invocado com mais de um argumento numérico, trata esses argumentos como elementos do array a ser criado. Isso significa que o construtor Array() não pode ser usado para criar um array com um único elemento numérico.

No ES6, a função Array.of() resolve este problema: é um método de fábrica que cria e retorna um novo array, usando seus valores de argumento (independentemente de quantos deles existam) como os elementos do array:

```
Matriz.de()          // => []; retorna array vazio sem
                     argumentos
Array.of(10)        // => [10]; pode criar matrizes com um único
                     argumento numérico
Array.of(1,2,3)     // => [1, 2, 3]
```

7.1.5 Array.from()

Array.from é outro método de fábrica de array introduzido no ES6. Ele espera um objeto iterável ou semelhante a um array como seu primeiro argumento e retorna um novo array que contém os elementos desse objeto. Com um argumento iterável, Array.from(iterable) funciona como o operador spread

[... iterable] Faz. Também é uma maneira simples de fazer uma cópia de uma matriz:

```
Seja copy = Array.From (original);
```

Array.From () também é importante porque define uma maneira de criar uma cópia verdadeira de um objeto semelhante a uma matriz. Objetos semelhantes a matrizes são objetos que não são de materiais que possuem uma propriedade de comprimento numérico e têm valores armazenados com propriedades cujos nomes são inteiros. Ao trabalhar com JavaScript do lado do cliente, os valores de retorno de alguns métodos do navegador da Web são semelhantes a matrizes, e pode ser mais fácil trabalhar com eles se você os converter primeiro em matrizes verdadeiras:

```
Deixe Truearray = Array.From (matriz);
```

Array.From () também aceita um segundo argumento opcional. Se você passar uma função como o segundo argumento, à medida que a nova matriz está sendo construída, cada elemento do objeto de origem será passado para a função que você especificar, e o valor de retorno da função será armazenado na matriz em vez do valor original. (Este é muito parecido com o método do mapa de array () que será introduzido posteriormente no capítulo, mas é mais eficiente realizar o mapeamento enquanto a matriz está sendo construída do que para construir a matriz e depois mapeá -la para outro novo variedade.)

7.2 Elementos de matriz de leitura e escrita

Você acessa um elemento de uma matriz usando o operador []. Uma referência à matriz deve aparecer à esquerda dos colchetes. Uma expressão arbitrária que possui um valor inteiro não negativo deve estar dentro do

colchetes. Você pode usar esta sintaxe para ler e escrever o valor de um elemento de uma matriz. Portanto, a seguir estão todas as declarações legais de JavaScript:

```
deixe a = ["mundo"]; //Começa com um array de um elemento
deixe valor = a[0];      //Lê o elemento 0
a[1] = 3,14;            // Escreve o elemento 1
seja i = 2;
uma[eu] = 3;             // Escreve o elemento 2
a[i + 1] = "olá";    // Escreve o elemento 3
uma[uma[i]] = uma[0];   //Lê os elementos 0 e 2, escreve
elemento 3
```

O que há de especial nos arrays é que quando você usa nomes de propriedades que são números inteiros e não negativos menores que $2 - 1$, o array mantém automaticamente o valor da propriedade `length` para você. No exemplo anterior, criamos um array `a` com um único elemento. Em seguida, atribuímos valores aos índices 1, 2 e 3. A propriedade `length` do array mudou conforme fizemos, então:

```
a.comprimento // => 4
```

Lembre-se de que arrays são um tipo especializado de objeto. Os colchetes usados para acessar os elementos do array funcionam exatamente como os colchetes usados para acessar as propriedades do objeto. JavaScript converte o índice da matriz numérica que você especifica em uma string – o índice 1 se torna a string “1” – e então usa essa string como um nome de propriedade. Não há nada de especial na conversão do índice de um número para uma string: você também pode fazer isso com objetos regulares:

```
deixe o = {}; // Cria um objeto simples
o[1] = "one"; // Indexa-o com um inteiro
```

```
o["1"]           // > "One"; nomes numéricos e de propriedades de string  
são iguais
```

É útil distinguir claramente um índice de matriz de um nome de propriedade do objeto. Todos os índices são nomes de propriedades, mas apenas nomes de propriedades que são inteiros entre 0 e 2-2 são índices. Todas as matrizes são objetos e você pode criar propriedades de qualquer nome nelas. Se você usar propriedades que são índices de matriz, no entanto, as matrizes têm o comportamento especial de atualizar sua propriedade de comprimento, conforme necessário.

Observe que você pode indexar uma matriz usando números negativos ou que não sejam inteiros. Quando você faz isso, o número é convertido em uma string e essa string é usada como o nome da propriedade. Como o nome não é um número inteiro não negativo, ele é tratado como uma propriedade de objeto regular, não um índice de matriz. Além disso, se você indexar uma matriz com uma string que seja um número inteiro não negativo, ela se comporta como um índice de matriz, não uma propriedade de objeto. O mesmo acontece se você usar um número de ponto flutuante que é o mesmo que um número inteiro:

```
a [-1.23] = true;    // Isso cria uma propriedade chamada "-1,23";
a ["1000"] = 0; // Este é o 1001º elemento da matriz
a [1.000] = 1;      // índice de matriz 1. O mesmo que a [1] = 1;
```

O fato de os índices de matriz serem simplesmente um tipo especial de nome da propriedade de objeto significa que as matrizes JavaScript não têm noção de um erro "fora dos limites". Quando você tenta consultar uma propriedade inexistente de qualquer objeto, você não recebe um erro; Você simplesmente fica indefinido. Isso é tão verdadeiro para matrizes quanto para objetos:

```
deixe um = [verdadeiro, falso]; // Esta matriz tem elementos nos índices  
0 e 1
```

```
A [2]          // >=; indefinido; nenhum elemento nisso
índice.
A [-1]         // >=; indefinido; Sem propriedade com isso
nome.
```

7.3 Matrizes esparsas

Uma matriz esparsa é aquela em que os elementos não possuem índices contíguos a partir de 0. Normalmente, a propriedade de comprimento de uma matriz especifica o número de elementos na matriz. Se a matriz for escassa, o valor da propriedade do comprimento será maior que o número de elementos. Matrizes esparsas podem ser criadas com o construtor Array () ou simplesmente atribuindo a um índice de matriz maior que o comprimento da matriz atual.

```
deixe a = nova matriz (5); // sem elementos, mas
a.length é 5. a = []; // Crie uma matriz sem elementos e
comprimento = 0. a [1000] = 0; // A atribuição adiciona
um elemento, mas define o comprimento para 1001.
```

Veremos mais tarde que você também pode fazer uma matriz escassa com o operador de exclusão.

Matrizes suficientemente escassas são normalmente implementadas de uma maneira mais lenta e com eficiência de memória do que as matrizes densas, e procurar elementos para tal matriz levará tanto tempo quanto a pesquisa regular de propriedades.

Observe que quando você omite um valor em uma matriz literal (usando vírgulas repetidas como em [1,,3]), a matriz resultante é escassa e os elementos omitidos simplesmente não existem:

```
seja a1 = [,];           // Este array não possui elementos e
comprimento 1 deixa a2 =
[indefinido];           // Este array tem um indefinido
elemento
0 em a1                 // => false: a1 não possui elemento com
índice 0
0 em a2                 // => true: a2 tem o indefinido
valor no índice 0
```

Compreender arrays esparsos é uma parte importante para compreender a verdadeira natureza dos arrays JavaScript. Na prática, porém, a maioria dos arrays JavaScript com os quais você trabalhará não serão esparsos. E, se você tiver que trabalhar com um array esparso, seu código provavelmente irá tratá-lo da mesma forma que trataria um array não esparso com elementos indefinidos.

7.4 Comprimento da matriz

Cada array tem uma propriedade `length`, e é essa propriedade que torna os arrays diferentes dos objetos JavaScript normais. Para arrays densos (ou seja, não esparsos), a propriedade `length` especifica o número de elementos no array. Seu valor é um a mais que o índice mais alto da matriz:

```
[].comprimento          // => 0: o array não possui elementos
["a","b"] .comprimento // => 3: o índice mais alto é 2, o comprimento é
3
```

Quando um array é esparso, a propriedade `length` é maior que o número de elementos, e tudo o que podemos dizer sobre isso é que o comprimento é garantido como maior que o índice de cada elemento do array. Ou, dito de outra forma, um array (esparso ou não) nunca terá um elemento cujo índice seja maior ou igual ao seu comprimento. A fim de

Para manter essa invariante, os arrays têm dois comportamentos especiais. A primeira descrevemos acima: se você atribuir um valor a um elemento do array cujo índice *i* é maior ou igual ao comprimento atual do array, o valor da propriedade *length* é definido como *i*+1.

O segundo comportamento especial que os arrays implementam para manter o comprimento invariante é que, se você definir a propriedade *length* como um número inteiro não negativo *n* menor que seu valor atual, quaisquer elementos do array cujo índice seja maior ou igual a *n* serão deletados. da matriz:

```
uma = [1,2,3,4,5];           // Comece com um array de 5 elementos.  
a.comprimento = 3;          // a é agora [1,2,3].  
a.comprimento = 0;          // Exclui todos os elementos. a é [].  
a.comprimento = 5;          // O comprimento é 5, mas nenhum elemento, como  
nova matriz (5)
```

Você também pode definir a propriedade *length* de uma matriz com um valor maior que seu valor atual. Na verdade, fazer isso não adiciona novos elementos ao array; simplesmente cria uma área esparsa no final do array.

7.5 Adicionando e Excluindo Elementos do Array

Já vimos a maneira mais simples de adicionar elementos a um array: basta atribuir valores a novos índices:

```
deixe a = [];           // Comece com um array vazio.  
a[0] = "zero"; // E adicione elementos a ele.  
a[1] = "um";
```

Você também pode usar o método *push()* para adicionar um ou mais valores ao

Fim de uma matriz:

```
deixe A = [];           // Comece com uma matriz vazia
a.push ("zero");      // Adicione um valor no final.   a =
["zero"] a.push ("um",
"dois"); // Adicione mais dois valores.   a = ["zero",
"One", "Two"]
```

Empurrar um valor para uma matriz A é o mesmo que atribuir o valor a um [A.Length]. Você pode usar o método `Netfift()` (descrito no §7.8) para inserir um valor no início de uma matriz, mudando os elementos da matriz existente para índices mais altos. O método `pop()` é o oposto de `push()`: remove o último elemento da matriz e o retorna, reduzindo o comprimento de uma matriz em 1. Da mesma forma, o método `shift()` remove e retorna o primeiro elemento da matriz, reduzindo o comprimento em 1 e mudando todos os elementos para um índice um menor que o índice atual. Veja §7.8 para saber mais sobre esses métodos.

Você pode excluir elementos de matriz com o operador Excluir, assim como você pode excluir propriedades do objeto:

```
Seja a = [1,2,3];
exclua um [2]; // agora não tem elemento no índice 2
2 em um        // => false: nenhum índice de matriz é definido
A. Length       // => 3: Excluir não afeta o comprimento da matriz
```

A exclusão de um elemento de matriz é semelhante a (mas sutilmente diferente de) atribuir indefinido a esse elemento. Observe que o uso de exclusão em um elemento de matriz não altera a propriedade `Length` e não muda os elementos com índices mais altos para preencher a lacuna que é deixada pela propriedade excluída. Se você excluir um elemento de uma matriz, a matriz

torna-se esparsa.

Como vimos acima, você também pode remover elementos do final de um array simplesmente definindo a propriedade length para o novo comprimento desejado.

Finalmente, splice() é o método de uso geral para inserir, excluir ou substituir elementos de array. Ele altera a propriedade length e muda os elementos da matriz para índices mais altos ou mais baixos conforme necessário. Consulte §7.8 para obter detalhes.

7.6 Iterando Matrizes

A partir do ES6, a maneira mais fácil de percorrer cada um dos elementos de um array (ou qualquer objeto iterável) é com o loop for/of, que foi abordado em detalhes em §5.4.4:

```
deixe letras = [..."Olá mundo"]; //Um array de letras
deixe string = "";
for(deixe letra de letras) {
    string += letra;
}
corda // => "Olá mundo"; remontamos o texto original
```

O iterador de array integrado que o loop for/of usa retorna os elementos de um array em ordem crescente. Não possui comportamento especial para matrizes esparsas e simplesmente retorna indefinido para quaisquer elementos da matriz que não existam.

Se você quiser usar um loop for/of para um array e precisar saber o índice de cada elemento do array, use o método Entry() do array,

junto com a atribuição de desestruturação, assim:

```
deixe todos = " "; for(let [índice, letra] de
letras.entries()) {
  if (índice % 2 === 0) todos os outros += letra;    // letras em
índices pares }

todos os outros // => "Hlowrd"
```

Outra boa maneira de iterar arrays é com `forEach()`. Esta não é uma nova forma de loop `for`, mas um método de array que oferece uma abordagem funcional para iteração de array. Você passa uma função para o método `forEach()` de um array, e `forEach()` invoca sua função uma vez em cada elemento do array:

```
deixe maiúscula = " ";
letras.forEach(letra => {      // Observe a sintaxe da função de seta
aqui
maiúscula += letra.toUpperCase(); });

maiúscula // => "OLÁ MUNDO"
```

Como seria de esperar, `forEach()` itera o array em ordem e, na verdade, passa o índice do array para sua função como um segundo argumento, o que ocasionalmente é útil. Ao contrário do loop `for/of`, o `forEach()` reconhece arrays esparsos e não invoca sua função para elementos que não existem.

[§7.8.1](#) documenta o método `forEach()` com mais detalhes. Essa seção também cobre métodos relacionados, como `map()` e `filter()`, que executam tipos especializados de iteração de array.

Você também pode percorrer os elementos de um array com um bom e antigo loop for (§5.4.3):

```
deixe vogais = ""; for(let i = 0; i < letras.length; i++) { // Para cada índice no array

    deixe letra = letras[i];                      //Pegue o elemento
    naquele índice
    if (/^[aeiou]/.test(letra)) {                //Use um normal
        teste de expressão
        vogais += letra;                          //Se for um
        vogal, lembre-se disso
    }
}
vogais
// => "áéíóú"
```

Em loops aninhados ou outros contextos onde o desempenho é crítico, às vezes você pode ver esse loop básico de iteração de array escrito de forma que o comprimento do array seja consultado apenas uma vez, em vez de em cada iteração. Ambas as formas de loop for a seguir são idiomáticas, embora não sejam particularmente comuns, e com os intérpretes JavaScript modernos, não está claro se eles têm algum impacto no desempenho:

```
// Salva o comprimento do array em uma variável local
for(let i = 0, len = letras.length; i < len; i++) {
    // o corpo do loop permanece o mesmo }

// Itera para trás do final do array até o início for(let i =
letras.length-1; i >= 0; i--) {
    // o corpo do loop permanece o mesmo }
```

Estes exemplos assumem que a matriz é densa e que todos os elementos contêm dados válidos. Se este não for o caso, você deve testar os elementos do array antes de usá-los. Se você quiser pular indefinido e

Elementos inexistentes, você pode escrever:

```
para (vamos i = 0; i < A.Length; i++) {if (a [i] === indefinido)
    continue; // Saltar indefinido +
    elementos inexistentes
    // corpo de loop aqui}
```

7.7 Matrizes multidimensionais

O JavaScript não suporta verdadeiras matrizes multidimensionais, mas você pode aproximar -las com matrizes de matrizes. Para acessar um valor em uma matriz de matrizes, basta usar o operador [] duas vezes. Por exemplo, suponha que a matriz variável seja uma matriz de matrizes de números. Todo elemento na matriz [x] é uma variedade de números. Para acessar um número específico nessa matriz, você escreveria a matriz [x] [y]. Aqui está um exemplo concreto que usa uma matriz bidimensional como uma tabela de multiplicação:

```
// Crie uma matriz multidimensional Let Table
= nova matriz (10); // 10 linhas da tabela

para (vamos i = 0; i < table.length; i++) {tabela [i] = nova
matriz (10); // Cada linha tem 10
colunas}

// Inicialize a matriz para (Let Rhe = 0; Row < table.Length; Row++)
{
    for (let col = 0; col < tabela [linha] .Length; col++)
        {tabela [linha] [col] = linha*col;
    }
}

// Use a matriz multidimensional para calcular 5*7 Tabela [5] [7] //=> 35
```

7.8 Métodos de Matriz

As seções anteriores focaram na sintaxe básica do JavaScript para trabalhar com arrays. Em geral, porém, são os métodos definidos pela classe Array que são os mais poderosos. As próximas seções documentam esses métodos. Ao ler sobre esses métodos, lembre-se de que alguns deles modificam o array ao qual são chamados e outros deixam o array inalterado. Vários métodos retornam um array: às vezes, este é um novo array e o original permanece inalterado. Outras vezes, um método modificará o array no local e também retornará uma referência ao array modificado.

Cada uma das subseções a seguir cobre um grupo de métodos de array relacionados:

- Os métodos iteradores percorrem os elementos de uma matriz, normalmente invocando uma função que você especifica em cada um desses elementos.
- Os métodos de pilha e fila adicionam e removem elementos do array desde o início e o fim de um array.
- Os métodos de subarray servem para extrair, excluir, inserir, preencher e copiar regiões contíguas de um array maior.
- Os métodos de pesquisa e classificação servem para localizar elementos em uma matriz e para classificar os elementos de uma matriz.

As subseções a seguir também cobrem os métodos estáticos da classe Array e alguns métodos diversos para concatenar arrays e converter arrays em strings.

7.8.1 Métodos de iterador de matriz

Os métodos descritos nesta seção iteram sobre as matrizes passando os elementos da matriz, para uma função que você fornece, e eles fornecem maneiras convenientes de iterar, mapear, filtrar, testar e reduzir as matrizes.

Antes de explicarmos os métodos em detalhes, no entanto, vale a pena fazer algumas generalizações sobre eles. Primeiro, todos esses métodos aceitam uma função como seu primeiro argumento e invocam essa função uma vez para cada elemento (ou alguns elementos) da matriz. Se a matriz for escassa, a função que você passa não será invocada para elementos inexistentes. Na maioria dos casos, a função que você fornece é invocada com três argumentos: o valor do elemento da matriz, o índice do elemento da matriz e a própria matriz. Muitas vezes, você só precisa do primeiro desses valores de argumento e pode ignorar o segundo e o terceiro valores.

A maioria dos métodos iteradores descritos nas subseções a seguir aceita um segundo argumento opcional. Se especificado, a função é invocada como se fosse um método deste segundo argumento. Ou seja, o segundo argumento que você passa se torna o valor dessa palavra -chave dentro da função que você passa como o primeiro argumento. O valor de retorno da função que você passa geralmente é importante, mas métodos diferentes lidam com o valor de retorno de maneiras diferentes. Nenhum dos métodos descritos aqui modifica a matriz em que eles são chamados (embora a função que você passa possa modificar a matriz, é claro).

Cada uma dessas funções é invocada com uma função como seu primeiro argumento, e é muito comum definir essa função embutida como parte da expressão de invocação do método em vez de usar uma função existente que

é definido em outro lugar. A sintaxe da função de seta (ver §8.1.3) funciona particularmente bem com esses métodos e a usaremos nos exemplos a seguir.

FOREACH()

O método `forEach()` itera através de um array, invocando uma função que você especifica para cada elemento. Conforme descrevemos, você passa a função como primeiro argumento para `forEach()`. `forEach()` então invoca sua função com três argumentos: o valor do elemento do array, o índice do elemento do array e o próprio array. Se você se preocupa apenas com o valor do elemento do array, você pode escrever uma função com apenas um parâmetro – os argumentos adicionais serão ignorados:

```
deixe dados = [1,2,3,4,5], soma = 0; // Calcula a
soma dos elementos do array data.forEach(value
=> { sum += value; }); //soma == 15

// Agora incremente cada elemento do array data.forEach(function(v,
i, a) { a[i] = v + 1; }); // dados == [2,3,4,5,6]
```

Observe que `forEach()` não fornece uma maneira de encerrar a iteração antes que todos os elementos tenham sido passados para a função. Ou seja, não há equivalente à instrução `break` que você possa usar com um loop `for` regular.

MAPA()

O método `map()` passa cada elemento do array no qual ele é invocado para a função que você especifica e retorna um array contendo os valores retornados pela sua função. Por exemplo:

```
seja a = [1, 2, 3];
a.map(x => x*x) // => [1, 4, 9]; a função recebe a entrada x
e retorna x*x
```

A função que você passa para map() é invocada da mesma forma que uma função passada para forEach(). Para o método map(), entretanto, a função que você passa deve retornar um valor. Observe que map() retorna um novo array: ele não modifica o array no qual é invocado. Se esse array for esparsa, sua função não será chamada para os elementos ausentes, mas o array retornado será esparsa da mesma forma que o array original: terá o mesmo comprimento e os mesmos elementos ausentes.

FILTRO()

O método filter() retorna um array contendo um subconjunto dos elementos do array no qual é invocado. A função que você passa para ele deve ser predicada: uma função que retorna verdadeiro ou falso. O predicado é invocado da mesma forma que forEach() e map(). Se o valor de retorno for verdadeiro, ou um valor que seja convertido em verdadeiro, então o elemento passado para o predicado será membro do subconjunto e será adicionado à matriz que se tornará o valor de retorno. Exemplos:

```
seja a = [5, 4, 3, 2, 1];
a.filtra(x => x < 3)    // => [2, 1]; valores inferiores a 3
a.filter((x,i) => i%2 === 0) // => [5, 3, 1]; qualquer outro valor
```

Observe que filter() ignora elementos ausentes em arrays esparsos e que seu valor de retorno é sempre denso. Para fechar as lacunas em uma matriz esparsa, você pode fazer o seguinte:

```
deixe denso = sparse.filter(() => true);
```

E para fechar lacunas e remover elementos indefinidos e nulos, você pode usar um filtro, assim:

```
a = a.filter(x => x !== undefined && x !== null);
```

ENCONTRAR() E ENCONTRAR ÍNDICE()

Os métodos `find()` e `findIndex()` são como `filter()` no sentido de que iteram por seu array procurando por elementos para os quais sua função de predicado retorna um valor verdadeiro. Ao contrário de `filter()`, entretanto, esses dois métodos param de iterar na primeira vez que o predicado encontra um elemento. Quando isso acontece, `find()` retorna o elemento correspondente e `findIndex()` retorna o índice do elemento correspondente. Se nenhum elemento correspondente for encontrado, `find()` retornará indefinido e `findIndex()` retornará -1:

```
seja a = [1,2,3,4,5];
a.findIndex(x => x === 3) // => 2; o valor 3 aparece em
índice 2 a.findIndex(x
=> x < 0) // => -1; não há números negativos em
a matriz a.find(x => x %
5 === 0) // => 5: este é um múltiplo de 5
a.encontrar(x => x % 7 === 0) // => indefinido: sem múltiplos de 7
na matriz
```

CADA() E ALGUNS()

Os métodos `every()` e `some()` são predicados de array: eles aplicam uma função de predicado que você especifica aos elementos do array e então retornam verdadeiro ou falso.

O método `every()` é como o quantificador matemático “para todos” \forall : ele retorna verdadeiro se e somente se sua função de predicado retornar verdadeiro para

todos os elementos da matriz:

```
seja a = [1,2,3,4,5];
a.cada(x => x < 10)    // => verdadeiro: todos os valores são < 10.
a.every(x => x % 2 === 0) // => false: nem todos os valores são pares.
```

O método some() é como o quantificador matemático “existe” \exists : ele retorna verdadeiro se existir pelo menos um elemento na matriz para o qual o predicado retorna verdadeiro e retorna falso se e somente se o predicado retorna falso para todos os elementos de a matriz:

```
seja a = [1,2,3,4,5];
a.algum(x => x%2==0)  // => verdadeiro; a tem alguns números pares.
a.algum(éNaN)          // => falso; a não tem não-números.
```

Observe que tanto every() quanto some() param de iterar os elementos do array como assim que souberem qual valor retornar. some() retorna verdadeiro na primeira vez que seu predicado retorna verdadeiro e apenas itera por todo o array se seu predicado sempre retornar falso. every() é o oposto: retorna falso na primeira vez que seu predicado retorna falso e só itera todos os elementos se seu predicado sempre retornar verdadeiro. Observe também que, por convenção matemática, every() retorna true e some retorna false quando invocado em um array vazio.

REDUZIR() E REDUZIR DIREITA()

Os métodos reduzir() e reduzirRight() combinam os elementos de um array, usando a função que você especifica, para produzir um único valor. Esta é uma operação comum em programação funcional

e também atende pelos nomes “injetar” e “dobrar”.

Exemplos ajudam a ilustrar como funciona:

```
seja a = [1,2,3,4,5];
a.reduzir((x,y) => x+y, 0)           // => 15; a soma do
valores a.reduce((x,y) => x*y,
1)                                     // => 120; o produto de
os valores a.reduce((x,y) => (x > y) ? x
: y) // => 5; o maior dos valores
```

`reduzir()` leva dois argumentos. A primeira é a função que realiza a operação de redução. A tarefa desta função de redução é combinar ou reduzir de alguma forma dois valores em um único valor e retornar esse valor reduzido. Nos exemplos que mostramos aqui, as funções combinam dois valores somando-os, multiplicando-os e escolhendo o maior. O segundo argumento (opcional) é um valor inicial a ser passado para a função.

As funções usadas com `reduzir()` são diferentes das funções usadas com `forEach()` e `map()`. Os valores familiares de valor, índice e matriz são passados como o segundo, terceiro e quarto argumentos. O primeiro argumento é o resultado acumulado da redução até agora. Na primeira chamada para a função, este primeiro argumento é o valor inicial que você passou como segundo argumento para `reduzir()`. Nas chamadas subsequentes, é o valor retornado pela invocação anterior da função. No primeiro exemplo, a função de redução é chamada primeiro com os argumentos 0 e 1. Ela os soma e retorna 1. Em seguida, é chamada novamente com os argumentos 1 e 2 e retorna 3. Em seguida, ela calcula $3+3=6$, depois $6+4=10$ e finalmente $10+5=15$. Este valor final, 15, torna-se o valor de retorno de `reduzir()`.

Você deve ter notado que a terceira chamada para `reduzir()` neste exemplo possui apenas um único argumento: não há valor inicial especificado. Quando você invoca `reduzir()` assim sem valor inicial, ele usa o primeiro elemento do array como valor inicial. Isso significa que a primeira chamada à função de redução terá o primeiro e o segundo elementos da matriz como primeiro e segundo argumentos. Nos exemplos de soma e produto, poderíamos ter omitido o argumento do valor inicial.

Chamar `reduzir()` em um array vazio sem argumento de valor inicial causa um `TypeError`. Se você chamá-lo com apenas um valor – seja um array com um elemento e nenhum valor inicial ou um array vazio e um valor inicial – ele simplesmente retorna aquele valor sem nunca chamar a função de redução.

`reduzRight()` funciona exatamente como `reduzir()`, exceto que processa a matriz do índice mais alto para o mais baixo (da direita para a esquerda), em vez de do mais baixo para o mais alto. Talvez você queira fazer isso se a operação de redução tiver associatividade da direita para a esquerda, por exemplo:

```
// Calcula 2^(3^4).      A exponenciação tem da direita para a esquerda
precedência seja a = [2, 3, 4];
a.reduceRight((acc, val) => Math.pow(val, acc))
// => 2.4178516392292583e+24
```

Observe que nem `reduzir()` nem `reduzirRight()` aceitam um argumento opcional que especifica o valor `this` no qual a função de redução deve ser invocada. O argumento opcional do valor inicial toma o seu lugar. Consulte o método `Function.bind()` (§8.7.5) se precisar que sua função de redução seja invocada como um método de um determinado

objeto.

Os exemplos mostrados até agora foram numéricos para simplificar, mas reduzir() e reduzirRight() não se destinam apenas a cálculos matemáticos. Qualquer função que possa combinar dois valores (como dois objetos) em um valor do mesmo tipo pode ser usada como função de redução. Por outro lado, algoritmos expressos usando reduções de array podem rapidamente se tornar complexos e difíceis de entender, e você pode achar mais fácil ler, escrever e raciocinar sobre seu código se usar construções de loop regulares para processar seus arrays.

7.8.2 Achatando arrays com flat() e flatMap()

No ES2019, o método flat() cria e retorna um novo array que contém os mesmos elementos do array em que é chamado, exceto que quaisquer elementos que sejam arrays são “achatados” no array retornado. Por exemplo:

```
[1, [2, 3]].plano()      // => [1, 2, 3]
[1, [2, [3]]].flat()     // => [1, 2, [3]]
```

Quando chamado sem argumentos, flat() nivela um nível de aninhamento. Os elementos da matriz original que são eles próprios matrizes são nivelados, mas os elementos da matriz dessas matrizes não são nivelados. Se você quiser nivelar mais níveis, passe um número para flat():

```
seja a = [1, [2, [3, [4]]]];
a.flat(1) // => [1, 2, [3,
[4]]] a.flat(2) // => [1, 2, 3,
[4]] a.flat(3) // => [1, 2, 3,
4] a.flat(4) // => [1, 2, 3, 4]
```

O método flatMap() funciona exatamente como o método map() (veja “map()”), exceto que o array retornado é automaticamente nivelado como se fosse passado para flat(). Ou seja, chamar a.flatMap(f) é o mesmo que (mas mais eficiente que) a.map(f).flat():

```
deixe frases = ["olá mundo", "o guia definitivo"];
deixe palavras = frases.flatMap(frase => frase.split(" "));
palavras // => ["olá", "mundo", "o", "definitivo", "guia"];
```

Você pode pensar em flatMap() como uma generalização de map() que permite que cada elemento do array de entrada seja mapeado para qualquer número de elementos do array de saída. Em particular, flatMap() permite mapear elementos de entrada para um array vazio, que se reduz a nada no array de saída:

```
// Mapeia números não negativos para suas raízes quadradas [-2, -1, 1, 2].flatMap(x => x < 0 ? [] : Math.sqrt(x)) // => [1, 2 **0,5]
```

7.8.3 Adicionando arrays com concat()

O método concat() cria e retorna um novo array que contém os elementos do array original no qual concat() foi invocado, seguido por cada um dos argumentos para concat(). Se algum desses argumentos for um array, então são os elementos do array que são concatenados, e não o array em si. Observe, entretanto, que concat() não nivelava recursivamente matrizes de matrizes. concat() não modifica o array no qual é invocado:

```
seja a = [1,2,3];
a.concat(4, 5)           // => [1,2,3,4,5]
a.concat([4,5],[6,7])    // => [1,2,3,4,5,6,7]; matrizes são
```

```
A.CONCAT achatado (4,  
[5, [6,7]])) // => [1,2,3,4,5, [6,7]]; mas não  
matrizes aninhadas a // => [1,2,3]; A matriz  
original não é modificada
```

Observe que concat () faz uma nova cópia da matriz que ela é chamada. Em muitos casos, essa é a coisa certa a fazer, mas é uma operação cara. Se você estiver escrevendo código como A = A.Concat (X), pense em modificar sua matriz no lugar com push () ou Splice () em vez de criar um novo.

7.8.4 pilhas e filas com push (), pop (), shift () e não dividido ()

Os métodos push () e pop () permitem trabalhar com matrizes como se fossem pilhas. O método push () anexa um ou mais novos elementos ao final de uma matriz e retorna o novo comprimento da matriz. Ao contrário do concat (), push () não acha os argumentos da matriz. O método pop () faz o inverso: ele exclui o último elemento de uma matriz, diminui o comprimento da matriz e retorna o valor que removeu. Observe que ambos os métodos modificam a matriz no lugar. A combinação de push () e pop () permite que você use uma matriz de javascript para

Implemente uma pilha de primeira entrada e saída. Por exemplo:

```
deixe pilha = [] // Stack == []  
stack.push (1,2); // pilha == [1,2];  
stack.pop (); // pilha == [1]; Retorna 2  
stack.push (3); // Stack == [1,3]  
stack.pop (); // pilha == [1]; retorna 3  
stack.push ([4,5]); // Stack == [1, [4,5]]  
stack.pop (); // pilha == [1]; Retorna [4,5]  
stack.pop (); // pilha == []; retorna 1
```

O método push () não acha uma matriz que você passa para ele, mas se você quiser empurrar todos os elementos de uma matriz para outra matriz, poderá usar o operador de espalhamento (§8.3.4) para achatá-lo explicitamente:

```
a.push (... valores);
```

Os métodos não () e shift () se comportam como push () e pop (), exceto que eles inserem e removem elementos do início de uma matriz e não do final. O não -definido () adiciona um elemento ou elementos ao início da matriz, muda os elementos da matriz existente para índices mais altos para abrir espaço e retorna o novo comprimento da matriz. Shift () remove e retorna o primeiro elemento da matriz, mudando todos os elementos subsequentes para baixo para ocupar o espaço recém -vago no início da matriz. Você pode usar o DENLING () e o shift () para implementar uma pilha, mas seria menos eficiente do que usar push () e pop () porque os elementos da matriz precisam ser deslocados para cima ou para baixo toda vez que um elemento é adicionado ou removido em o início da matriz. Em vez disso, porém, você pode implementar uma estrutura de dados da fila usando push () para adicionar elementos no final de uma matriz e shift () para removê-los do início da matriz:

```
Seja q = [];
q.push (1,2);
q.shift ();
q.push (3)
q.shift ()
q.shift ()
```

// q == []
// q == [1,2]
// q == [2]; retorna 1
// q == [2, 3]
// q == [3]; Retorna 2
// q == []; retorna 3

Há uma característica do não -quei () que vale a pena chamar porque você pode achar surpreendente. Ao passar vários argumentos para

Desenvolver (), eles são inseridos de uma só vez, o que significa que acabam na matriz em uma ordem diferente da que seriam se você os inserisse um de cada vez:

```
deixe A = [];           // a == []
A.UnShift (1)          // a == [1]
A.UnShift (2)          // a == [2, 1]
a = [];                // a == []
A.UnShift (1,2)         // a == [1, 2]
```

7.8.5 Subarra -se com Slice (), Splice (), Fill () e CopyWithin ()

As matrizes definem uma série de métodos que funcionam em regiões contíguas, ou subarrays ou "fatias" de uma matriz. As seções a seguir descrevem métodos para extrair, substituir, preencher e copiar fatias.

FATIAR()

O método Slice () retorna uma fatia, ou subarray, da matriz especificada. Seus dois argumentos especificam o início e o final da fatia a serem devolvidos. A matriz retornada contém o elemento especificado pelo primeiro argumento e todos os elementos subsequentes até, mas não incluindo o elemento especificado pelo segundo argumento. Se apenas um argumento for especificado, a matriz retornada contém todos os elementos da posição inicial até o final da matriz. Se um argumento for negativo, especifica um elemento de matriz em relação ao comprimento da matriz. Um argumento de -1, por exemplo, especifica o último elemento na matriz e um argumento de -2 especifica o elemento antes desse. Observe que o slice () não modifica a matriz em que é invocada. Aqui estão alguns exemplos:

```
seja a = [1,2,3,4,5];
a.fatia(0,3);      // Retorna [1,2,3]
uma.fatia(3);      //Retorna [4,5]
uma.fatia(1,-1);   // Retorna [2,3,4]
uma.fatia(-3,-2);  //Retorna [3]
```

EMENDA()

`splice()` é um método de uso geral para inserir ou remover elementos de um array. Ao contrário de `slice()` e `concat()`, `splice()` modifica o array no qual é invocado. Observe que `splice()` e `slice()` têm nomes muito semelhantes, mas executam operações substancialmente diferentes.

`splice()` pode excluir elementos de um array, inserir novos elementos em um array ou realizar ambas as operações ao mesmo tempo. Os elementos do array que vêm após o ponto de inserção ou exclusão têm seus índices aumentados ou diminuídos conforme necessário para que permaneçam contíguos ao restante do array. O primeiro argumento para `splice()` especifica a posição do array na qual a inserção e/ou exclusão deve começar. O segundo argumento especifica o número de elementos que devem ser excluídos (separados) do array. (Observe que esta é outra diferença entre esses dois métodos. O segundo argumento para `slice()` é uma posição final. O segundo argumento para `splice()` é um comprimento.) Se este segundo argumento for omitido, todos os elementos da matriz do elemento inicial até o final da matriz são removidos. `splice()` retorna um array de elementos excluídos ou um array vazio se nenhum elemento foi excluído. Por exemplo:

```
seja a = [1,2,3,4,5,6,7,8]; a.splice(4) // => [5,6,7,8]; a é
agora [1,2,3,4]
```

```
uma.splice(1,2) // => [2,3]; a é agora [1,4]
uma.splice(1,1) // => [4]; a é agora [1]
```

Os primeiros dois argumentos para splice() especificam quais elementos do array devem ser excluídos. Esses argumentos podem ser seguidos por qualquer número de argumentos adicionais que especificam os elementos a serem inseridos no array, começando na posição especificada pelo primeiro argumento. Por exemplo:

```
seja a = [1,2,3,4,5];
uma.splice(2,0,"a","b") // => []; a agora é [1,2,"a","b",3,4,5]
uma.splice(2,2,[1,2],3) // => ["a","b","c"]; a é agora [1,2,
[1,2],3,3,4,5]
```

Observe que, diferentemente de concat(), splice() insere os próprios arrays, não os elementos desses arrays.

PREENCHER()

O método fill() define os elementos de um array, ou uma fatia de um array, para um valor especificado. Ele altera o array em que é chamado e também retorna o array modificado:

```
deixe a = novo Array(5); //Começa sem elementos e comprimento
5h.preencher(0)          // => [0,0,0,0,0]; preencha a matriz
com zeros a.fill(9, 1)   // => [0,9,9,9,9]; preencha com 9
começando no índice 1    // => [0,9,8,8,9]; preencha com 8 em
a.fill(8, 2, -1)         índices 2, 3
```

O primeiro argumento para fill() é o valor para definir os elementos do array. O segundo argumento opcional especifica o índice inicial. Se omitido, o preenchimento começa no índice 0. O terceiro argumento opcional especifica o final

índice — elementos da matriz até, mas não incluindo, este índice serão preenchidos. Se este argumento for omitido, o array será preenchido do índice inicial ao final. Você pode especificar índices relativos ao final do array passando números negativos, assim como faz para slice().

COPIAR DENTRO()

copyWithin() copia uma fatia de um array para uma nova posição dentro do array. Ele modifica o array no local e retorna o array modificado, mas não alterará o comprimento do array. O primeiro argumento especifica o índice de destino para o qual o primeiro elemento será copiado. O segundo argumento especifica o índice do primeiro elemento a ser copiado. Se este segundo argumento for omitido, 0 será usado. O terceiro argumento especifica o final da fatia de elementos a ser copiada. Se omitido, o comprimento da matriz será usado. Elementos do índice inicial até, mas não incluindo, o índice final serão copiados. Você pode especificar índices relativos ao final do array passando números negativos, assim como você pode fazer com slice():

```
seja a = [1,2,3,4,5];
a.copyWithin(1)           // => [1,1,2,3,4]: copia os elementos do array
up one a.copyWithin(2, 3, 5) // => [1,1,3,4,4]: copia os últimos 2 elementos para o índice 2
a.copyWithin(0, -2)        // => [4,4,3,4,4]: deslocamentos negativos
trabalho também
```

copyWithin() pretende ser um método de alto desempenho que é particularmente útil com arrays digitados (veja §11.2). Ele é modelado a partir da função memmove() da biblioteca padrão C. Observe que a cópia funcionará corretamente mesmo se houver sobreposição entre a fonte e

regiões de destino.

7.8.6 Métodos de pesquisa e classificação de matrizes

Matrizes implementam `indexOF ()`, `LastIndexOf ()` e inclui
() métodos semelhantes aos métodos de strings com o
mesmo nome. Também existem métodos de classificação
(`)` e `reverse ()` para reordenar os elementos de uma matriz.
Esses métodos são descritos nas subseções a seguir.

`IndexOf ()` e `LastIndexOf ()`

`Index0f ()` e `LastIndex0f ()` pesquisam uma matriz
por um elemento com um valor especificado e
retorne o índice do primeiro elemento
encontrado, ou `-1` se nenhum for encontrado.
`index0f ()` pesquisa a matriz do começo ao fim, e
as pesquisas `lastIndex0F ()` do final ao começo:

```
Seja a = [0,1,2,1,0];
A.IndexOf (1)           // => 1: a [1] é 1
A.LastIndex0F (1)        // => 3: a [3] é 1
A.IndexOf (3)           // => -1: nenhum elemento tem valor 3
```

`Index0f ()` e `LastIndex0F ()` compararam seu argumento com os
elementos da matriz usando o equivalente ao operador `==`.
Se sua matriz contiver objetos em vez de valores
primitivos, esses métodos verificam se duas referências se
referem exatamente ao mesmo objeto. Se você deseja
realmente olhar para o conteúdo de um objeto, tente usar o
método `find ()` com sua própria função de predicado
personalizado.

`indexof ()` e `lastIndexof ()` tomam um segundo opcional

argumento que especifica o índice da matriz no qual iniciar a pesquisa. Se este argumento for omitido, indexOf() começa no início e lastIndexOf() começa no final. Valores negativos são permitidos para o segundo argumento e são tratados como um deslocamento do final do array, assim como para o método slice(): um valor de -1, por exemplo, especifica o último elemento do array.

A função a seguir pesquisa um array em busca de um valor especificado e retorna um array de todos os índices correspondentes. Isso demonstra como o segundo argumento de indexOf() pode ser usado para encontrar correspondências além do primeiro.

```
// Encontra todas as ocorrências de um valor x em um array a e
// retorna um array
// de índices correspondentes
function findall(a, x) { let
results = [],                                //O array de índices
nós retornaremos
    len = a.comprimento,                      //O comprimento da matriz
    ser pesquisado
    posição = 0;                            //A posição a ser pesquisada
de
    while(pos < len) {                      // Embora mais elementos para
procurar...
        pos = a.indexOf(x, pos); // Pesquisa if (pos === -1) break;
        // Se nada for encontrado, estamos
feito.
        resultados.push(pos);                // Caso contrário, armazena o índice em
variedade
        pos = pos + 1;                      // E comece a próxima pesquisa em
próximo elemento
    } retornar resultados;
}                                            //Retorna array de índices
```

Observe que strings possuem métodos indexOf() e lastIndexOf()

que funcionam como esses métodos de array, exceto que um segundo argumento negativo é tratado como zero.

INCLUI()

O método ES2016 inclui() recebe um único argumento e retorna verdadeiro se o array contiver esse valor ou falso caso contrário. Não informa o índice do valor, apenas se ele existe. O método include() é efetivamente um teste de associação de conjunto para arrays. Observe, entretanto, que arrays não são uma representação eficiente para conjuntos, e se você estiver trabalhando com mais do que alguns elementos, você deve usar um objeto Set real (§11.1.1).

O método include() é um pouco diferente do indexOf()

método de uma maneira importante. indexOf() testa a igualdade usando o mesmo algoritmo que o operador === faz, e esse algoritmo de igualdade considera o valor que não é um número como diferente de todos os outros valores, incluindo ele mesmo. inclui() usa uma versão ligeiramente diferente de igualdade que considera NaN igual a si mesmo. Isso significa que indexOf() não detectará o valor NaN em um array, mas

incluir() irá:

```
seja a = [1,verdadeiro,3,NaN]; a.includes(true) //  
=> verdadeiro a.includes(2) // => falso  
a.includes(NaN) // => verdadeiro a.indexOf(NaN)  
// => -1; indexOf não consegue encontrar NaN
```

ORGANIZAR()

sort() classifica os elementos de um array no lugar e retorna o classificado

variedade. Quando sort() é chamado sem argumentos, ele ordena os elementos do array em ordem alfabética (convertendo-os temporariamente em strings para realizar a comparação, se necessário):

```
deixe a = ["banana", "cereja", "maçã"];
a.sort(); // a == ["maçã", "banana", "cereja"]
```

Se uma matriz contiver elementos indefinidos, eles serão classificados no final da matriz.

Para classificar um array em alguma ordem diferente da alfabética, você deve passar uma função de comparação como argumento para sort(). Esta função decide qual dos seus dois argumentos deve aparecer primeiro no array ordenado. Se o primeiro argumento aparecer antes do segundo, a função de comparação deverá retornar um número menor que zero. Se o primeiro argumento aparecer após o segundo na matriz classificada, a função deverá retornar um número maior que zero. E se os dois valores forem equivalentes (ou seja, se a ordem deles for irrelevante), a função de comparação deverá retornar 0. Assim, por exemplo, para classificar os elementos do array em ordem numérica em vez de alfabética, você pode fazer o seguinte:

```
seja a = [33, 4, 1111, 222];
a.sort();                                //a == [1111, 222, 33, 4];
ordem alfabética
a.sort(function(a,b) {                  //Passa uma função comparadora
    retornar ab;                      // Retorna < 0, 0 ou > 0, dependendo
    sob encomenda
});                                //a == [4, 33, 222, 1111]; numérico
ordenar a.sort((a,b)                  //a == [1111, 222, 33, 4]; reverter
=> ba);                                //a == [4, 33, 222, 1111]; numérico
ordem numérica
```

Como outro exemplo de classificação de itens de array, você pode executar um caso-

classificação alfabética insensível em uma matriz de strings, passando uma função de comparação que converte ambos os seus argumentos em letras minúsculas (com o método `toLowerCase()`) antes de compará-los:

```
deixe a = [&quot;formiga&quot;, &quot;Bug&quot;, &quot;gato&quot;,  
&quot;Cachorro&quot;]; a.sort(); // a ==  
[&quot;Bug&quot;,&quot;Cachorro&quot;,&quot;Formiga&quot;,&quot;Gato&quot;];  
classificação com distinção entre maiúsculas e minúsculas  
a.sort(function(s,t) {  
    deixe a = s.toLowerCase();  
    deixe b = t.toLowerCase();  
    se (a < b) retornar -1;  
    se (a > b) retornar 1;  
    retornar 0;  
});      // a == [&quot;formiga&quot;,&quot;Bug&quot;,&quot;gato&quot;,&quot;Cão&quot;]; sem distinção entre  
organizar          maiúsculas e minúsculas
```

REVERTER()

O método `reverse()` inverte a ordem dos elementos de um array e retorna o array invertido. Ele faz isso no lugar; em outras palavras, ele não cria um novo array com os elementos reorganizados, mas os reorganiza no array já existente:

```
seja a = [1,2,3];  
a.reverse();      // uma == [3,2,1]
```

7.8.7 Conversões de array para string

A classe `Array` define três métodos que podem converter arrays em strings, o que geralmente é algo que você pode fazer ao criar mensagens de log e de erro. (Se você quiser salvar o conteúdo de um array em formato textual para reutilização posterior, serialize o array com

`JSON.stringify()` [§6.8] em vez de usar os métodos descritos aqui.)

O método junção () converte todos os elementos de uma matriz em strings e os concatena, retornando a sequência resultante. Você pode especificar uma sequência opcional que separa os elementos na sequência resultante. Se nenhuma string separadora for especificada, uma vírgula será usada:

```
Seja a = [1, 2, 3]; A.Join () // =>  
"1,2,3"; A.Join ("") // =>  
"1 2 3"; A.Join (" ") // =>  
"123"; Let B = Novo Matriz (10); // Uma  
matriz de comprimento 10 sem elementos  
  
B.Join ("-"); // => "-----": uma sequência de 9  
hífens
```

O método junção () é o inverso do método string.split (), que cria uma matriz quebrando uma string em pedaços.

Matrizes, como todos os objetos JavaScript, possuem um método ToString (). Para uma matriz, esse método funciona como o método junção () sem argumentos:

```
[1,2,3] .ToString ()           // => "1,2,3"  
[A,B,C].ToString ()           // => "A, B, C"  
[[1, [2, C]]].ToString ()     // => "1,2, C"
```

Observe que a saída não inclui colchetes quadrados ou qualquer outro tipo de delimitador em torno do valor da matriz.

toLocalestring () é a versão localizada do toString (). Ele converte cada elemento da matriz em uma string chamando o

toLocalestring () Método do elemento e, em seguida

string separadora definida pela implementação).

7.8.8 Funções de matriz estática

Além dos métodos de array que já documentamos, a classe Array também define três funções estáticas que você pode invocar por meio do construtor Array em vez de em arrays. Array.of() e

Array.from() são métodos de fábrica para criar novos arrays. Eles foram documentados em §7.1.4 e §7.1.5.

A outra função de array estático é Array.isArray(), que é útil para determinar se um valor desconhecido é um array ou não:

```
Array.isArray([])           // => verdadeiro  
Array.isArray({})         // => falso
```

7.9 Objetos semelhantes a array

Como vimos, arrays JavaScript possuem alguns recursos especiais que outros objetos não possuem:

- A propriedade length é atualizada automaticamente à medida que novos elementos são adicionados à lista.
- Definir o comprimento para um valor menor trunca a matriz.
- Arrays herdam métodos úteis de Array.prototype.
- `Array.isArray()` retorna verdadeiro para arrays.

Esses são os recursos que diferenciam os arrays JavaScript dos objetos regulares. Mas não são os recursos essenciais que definem um array. Muitas vezes é perfeitamente razoável tratar qualquer objeto com comprimento numérico

propriedade e propriedades inteiras não negativas correspondentes como uma espécie de matriz.

Esses objetos "parecidos com a matriz" realmente aparecem ocasionalmente na prática e, embora você não possa invocar diretamente os métodos de matriz neles ou esperar um comportamento especial da propriedade Length, você ainda pode iterá-los através deles com o mesmo código que usaria para um verdadeiro varíeade. Acontece que muitos algoritmos de matriz funcionam tão bem com objetos semelhantes a matrizes quanto com matrizes reais. Isto é especialmente verdade se seus algoritmos tratarão a matriz somente leitura ou se eles pelo menos deixarem o comprimento da matriz

inalterado.

O código a seguir pega um objeto regular, adiciona propriedades para torná-lo um objeto semelhante a uma matriz e depois iterá-los através dos "elementos" do pseudo-grays resultante:

```
deixe A = {};// Comece com um objeto vazio regular
// Adicione propriedades para torná-lo "parecido com a matriz"; deixe i = 0;
enquanto (i < 10) {
    a [i] = i * i; i++;
}

A.Length = i;

// agora iterá-los através dele como se fosse uma matriz real, deixe total = 0;
para (vamos j = 0; j < A.Length; j++) {total+= a [j];
}
```

No JavaScript do lado do cliente, vários métodos para trabalhar com

Documentos HTML (como Document.QuerySelectorall (), por exemplo)
Retornar objetos semelhantes a matrizes. Aqui está uma função que você
pode usar para testar objetos que funcionam como matrizes:

```
// Determine se o é um objeto semelhante a uma matriz. // Strings
e funções têm propriedades numéricas de comprimento, mas são

// excluído pelo teste TIP00F. No JavaScript do lado do
cliente, DOM Text
// os nós têm uma propriedade de comprimento numérico e pode
precisar ser excluído
// com um teste O.NodeType adicional! == 3. função isArraylike (o) {

    se (O &amp;&amp;
        indefinido, etc.
        typeof o === "objeto" &amp;&amp;
        Número.isfinite (O.Length) &amp;&amp;
        número finito
        O.Length> = 0 &amp;&amp;
        negativo
        Número.isinteger (O.Length)
        Inteiro &amp;&amp;
        O.Length <4294967295) {
            1
            retornar true;
            como.
        } outro {
            retornar falso;
            não.
    } }
```

Veremos em uma seção posterior que as cordas se comportam
como matrizes. No entanto, testes como este para objetos
semelhantes a matrizes geralmente retornam falsos para strings-eles
geralmente são melhor tratados como cordas, não como matrizes.

A maioria dos métodos de matriz JavaScript é definida propositadamente como genérico, então

que eles funcionam corretamente quando aplicados a objetos semelhantes a matrizes, além de matrizes verdadeiras. Já que objetos semelhantes a matrizes não herdam

`Array.prototype`, você não pode invocar métodos de matriz diretamente neles. Você pode invocá -los indiretamente usando o método `function.call`, no entanto (consulte §8.7.4 para obter detalhes):

```
Seja a = {"0": "a", "1": "b", "2": "c", comprimento: 3}; // um objeto de matriz
Array.prototype.join.call (a, "+") // =>
"A+B+C" Array.prototype.map.call (a, x
=> x.TOUppercase ()) // =>
[A, B, C] Array.prototype.slice.call (a, 0)
// => [a, b, c]: verdadeiro
Array Copy
Array.From (A) // => [a, b, c]
cópia mais fácil da matriz
```

A segunda linha do código deste código chama o método da matriz `Slice ()` em um objeto semelhante a uma matriz para copiar os elementos desse objeto para um objeto de matriz verdadeiro. Este é um truque idiomático que existe em muito código legado, mas agora é muito mais fácil de fazer com

`Array.From ()`.

7.10 Strings como matrizes

As cordas JavaScript se comportam como matrizes somente leitura de caracteres UTF-16 Unicode. Em vez de acessar caracteres individuais com o método `charat ()`, você pode usar colchetes:

```
Seja s =
"teste" // => "t"
s.Charat (0) // => "t"
s [1] // => "e"
```

O operador `typeof` ainda retorna “string” para strings, é claro, e o método `Array.isArray()` retorna `false` se você passar uma string para ele.

O principal benefício das strings indexáveis é simplesmente que podemos substituir chamadas para `charAt()` por colchetes, que são mais concisos e legíveis, e potencialmente mais eficientes. O fato de strings se comportarem como arrays também significa, entretanto, que podemos aplicar métodos genéricos de array a elas. Por exemplo:

```
Array.prototype.join.call("JavaScript", " ") // => "Java  
Roteiro";
```

Tenha em mente que strings são valores imutáveis, portanto, quando tratadas como arrays, são arrays somente leitura. Métodos de array como `push()`, `sort()`, `reverse()` e `splice()` modificam um array no local e não funcionam em strings. Entretanto, a tentativa de modificar uma string usando um método de array não causa um erro: ela simplesmente falha silenciosamente.

7.11 Resumo

Este capítulo abordou arrays JavaScript em profundidade, incluindo detalhes esotéricos sobre arrays esparsos e objetos semelhantes a arrays. Os principais pontos a serem extraídos deste capítulo são:

- Literais de array são escritos como listas de valores separados por vírgulas entre colchetes.
- Os elementos individuais da matriz são acessados especificando o índice da matriz desejado entre colchetes.

- O loop for/of e o operador ... spread introduzidos no ES6 são formas particularmente úteis de iterar arrays.
- A classe Array define um rico conjunto de métodos para manipular arrays, e você deve se familiarizar com a API Array.

Capítulo 8. Funções

Este capítulo abrange as funções JavaScript. As funções são um bloco de construção fundamental para programas JavaScript e um recurso comum em quase todas as linguagens de programação. Você já pode estar familiarizado com o conceito de uma função em um nome como sub -rotina ou procedimento.

Uma função é um bloco de código JavaScript que é definido uma vez, mas pode ser executado ou invocado, várias vezes. As funções JavaScript são parametrizadas: uma definição de função pode incluir uma lista de identificadores, conhecidos como parâmetros, que funcionam como variáveis locais para o corpo da função. As invocações da função fornecem valores ou argumentos para os parâmetros da função. As funções geralmente usam seus valores de argumento para calcular um valor de retorno que se torna o valor da expressão de invocação de função. Além dos argumentos, cada invocação tem outro valor - o contexto de invocação - esse é o valor dessa palavra -chave.

Se uma função for atribuída a uma propriedade de um objeto, ela é conhecida como um método desse objeto. Quando uma função é invocada ou através de um objeto, esse objeto é o contexto de invocação ou esse valor para a função. As funções projetadas para inicializar um objeto recém -criado são chamadas de construtores. Os construtores foram descritos no §6.2 e serão cobertos novamente no capítulo 9.

No JavaScript, as funções são objetos e podem ser manipuladas por programas. O JavaScript pode atribuir funções a variáveis e passá-las para outras funções, por exemplo. Como as funções são objetos, você pode definir propriedades neles e até chamar métodos sobre eles.

As definições de função JavaScript podem ser aninhadas em outras funções e têm acesso a quaisquer variáveis que estejam no escopo onde são definidas. Isso significa que as funções de JavaScript são fechamentos e permite técnicas importantes e poderosas de programação.

8.1 Definindo funções

A maneira mais direta de definir uma função JavaScript é com a palavra-chave da função, que pode ser usada como declaração ou expressão. O ES6 define uma nova maneira importante de definir funções sem a palavra-chave da função: “Funções de seta” têm uma sintaxe particularmente compacta e são úteis ao passar uma função como um argumento para outra função. As subseções a seguir cobrem essas três maneiras de definir funções. Observe que alguns detalhes da sintaxe da definição de função envolvendo parâmetros de função são adiados para §8.3.

Em literais de objetos e definições de classe, há uma sintaxe de abreviação conveniente para definir métodos. Essa sintaxe abreviada foi coberta no §6.10.5 e é equivalente a usar uma expressão de definição de função e atribuí-la a uma propriedade de objeto usando o nome básico: Sintaxe literal do objeto Valor. Em outro caso especial, você pode usar palavras-chave e definir em literais de objeto para definir a propriedade especial Getter e Setter

Métodos. Esta sintaxe de definição de função foi abordada no §6.10.6.

Observe que as funções também podem ser definidas com o construtor `function ()`, que é o assunto do §8.7.7. Além disso, o JavaScript define alguns tipos especializados de funções. Função* define as funções geradoras (consulte o capítulo 12) e a função assíncrona define funções assíncronas (consulte o Capítulo 13).

8.1.1 declarações de função

As declarações de função consistem na palavra -chave da função, seguida por esses componentes:

- Um identificador que nomeia a função. O nome é uma parte necessária das declarações de função: é usado como o nome de uma variável e o objeto de função recém -definido é atribuído à variável.
- Um par de parênteses em torno de uma lista separada por vírgula de zero ou mais identificadores. Esses identificadores são os nomes de parâmetros para a função e se comportam como variáveis locais dentro do corpo da função.
- Um par de aparelhos encaracolados com zero ou mais declarações JavaScript dentro. Essas declarações são o corpo da função: elas são executadas sempre que a função é invocada.

Aqui estão algumas declarações de função de exemplo:

```
// Imprima o nome e o valor de cada propriedade de o.           Retornar
indefinido. Função printProps
(0) {for (Let p in 0) {

    console.log(` ${p}: ${o[p]}\n`);
```

```
}

// Calcula a distância entre os pontos cartesianos (x1,y1) e
// (x2,y2).
função distância(x1, y1, x2, y2) {seja
dx = x2 - x1; seja dy = y2 - y1;

retornar Math.sqrt(dx*dx + dy*dy); }

// Uma função recursiva (que chama a si mesma) que calcula
fatoriais
// Lembre-se disso x! é o produto de x e todos os números
inteiros positivos menores que ele. função fatorial(x) {

    se (x &lt;= 1) retornar 1;
    retornar x * fatorial (x-1);
}
```

Uma das coisas importantes a entender sobre as declarações de funções é que o nome da função se torna uma variável cujo valor é a própria função. As instruções de declaração de função são “içadas” para o topo do script, função ou bloco envolvente para que as funções definidas desta forma possam ser invocadas a partir do código que aparece antes da definição. Outra maneira de dizer isso é que todas as funções declaradas em um bloco de código JavaScript serão definidas ao longo desse bloco e serão definidas antes que o interpretador JavaScript comece a executar qualquer código desse bloco.

As funções `distance()` e `factorial()` que descrevemos são projetadas para calcular um valor e usam `return` para retornar esse valor ao chamador. A instrução `return` faz com que a função pare de ser executada e retorne o valor de sua expressão (se houver) ao chamador. Se a instrução `return` não tiver uma expressão associada, o

o valor de retorno da função é indefinido.

A função `printprops()` é diferente: sua função é gerar os nomes e valores das propriedades de um objeto. Nenhum valor de retorno é necessário e a função não inclui uma instrução de retorno. O valor de uma invocação da função `printprops()` é sempre indefinido. Se uma função não contém uma instrução de retorno, ela simplesmente executa cada instrução no corpo da função até chegar ao fim e retorna o valor indefinido ao chamador.

Antes do ES6, as declarações de funções só eram permitidas no nível superior em um arquivo JavaScript ou em outra função. Embora algumas implementações violassem a regra, não era tecnicamente legal definir funções dentro do corpo de loops, condicionais ou outros blocos. No modo estrito do ES6, entretanto, declarações de funções são permitidas dentro de blocos. Entretanto, uma função definida dentro de um bloco só existe dentro desse bloco e não é visível fora do bloco.

8.1.2 Expressões de Função

As expressões de função se parecem muito com declarações de função, mas aparecem no contexto de uma expressão ou instrução maior e o nome é opcional. Aqui estão alguns exemplos de expressões de função:

```
// Esta expressão de função define uma função que eleva ao
// quadrado seu argumento.
// Observe que o atribuímos a uma variável const square = function(x) {
return x*x; };

// As expressões de função podem incluir nomes, o que é útil
// para recursão.
const f = função fato(x) { if (x <= 1) retornar 1; outro
```

```
retornar x*fato (x-1); };

// As expressões de função também podem ser usadas como
argumentos para outras funções:
[3,2,1] .Sort (função (a, b) {return ab;});

// As expressões de função são às vezes definidas e imediatamente
invocadas:
Seja tensquared = (function (x) {return x*x;} (10));
```

Observe que o nome da função é opcional para funções definidas como expressões e a maioria das expressões de função anterior que o omitimos. Uma declaração de função realmente declara uma variável e atribui um objeto de função a ele. Uma expressão de função, por outro lado, não declara uma variável: cabe a você atribuir o objeto de função recém -definido a uma constante ou variável, se você precisar se referir a ele várias vezes. É uma boa prática usar const com expressões de função para que você não substitua accidentalmente suas funções atribuindo novos valores.

Um nome é permitido para funções, como a função factorial, que precisam se referir a si mesmas. Se uma expressão de função incluir um nome, o escopo da função local para essa função incluirá uma ligação desse nome ao objeto de função. Com efeito, o nome da função se torna uma variável local dentro da função. A maioria das funções definidas como expressões não precisa de nomes, o que torna sua definição mais compacta (embora não seja tão compacta quanto as funções de seta, descritas abaixo).

Há uma diferença importante entre definir uma função f () com uma declaração de função e atribuir uma função à variável f após criá -la como uma expressão. Quando você usa o formulário de declaração, os objetos de função são criados antes do código que os contém começar a

Executar e as definições são içadas para que você possa chamar essas funções do código que aparece acima da instrução de definição. No entanto, isso não é verdade para funções definidas como expressões: essas funções não existem até que a expressão que as define seja realmente avaliada. Além disso, para invocar uma função, você deve ser capaz de se referir a ela e não pode se referir a uma função definida como uma expressão até que seja atribuída a uma variável; portanto, as funções definidas com expressões não podem ser invocadas antes de serem definido.

8.1.3 Funções de seta

No ES6, você pode definir funções usando uma sintaxe particularmente compacta conhecida como "funções de seta". Esta sintaxe é uma reminiscência de notação matemática e usa um "`=>`" para separar os parâmetros da função do corpo da função. A palavra -chave da função não é usada e, como as funções de seta são expressões em vez de declarações, também não há necessidade de um nome de função. A forma geral de uma função de seta é uma lista separada por vírgula de parâmetros entre parênteses, seguida pela seta `=>`, seguida pelo corpo da função em aparelhos encaracolados:

```
const sum = (x, y) => {return x + y;};
```

Mas as funções de seta suportam uma sintaxe ainda mais compacta. Se o corpo da função for uma única declaração de retorno, você poderá omitir a palavra -chave de retorno, o ponto de vírgula que acompanha ela e os aparelhos encaracolados e escrever o corpo da função como expressão cujo valor deve ser devolvido:

```
const sum = (x, y) => x + y;
```

Além disso, se uma função de seta tiver exatamente um parâmetro, você poderá omitir os parênteses ao redor da lista de parâmetros:

```
const polinômio = x => x*x + 2*x + 3;
```

Observe, entretanto, que uma função de seta sem nenhum argumento deve ser escrita com um par de parênteses vazio:

```
const constanteFunc = () => 42;
```

Observe que, ao escrever uma função de seta, você não deve colocar uma nova linha entre os parâmetros da função e a seta =>. Caso contrário, você poderia acabar com uma linha como const polynomial = x, que é uma instrução de atribuição sintaticamente válida por si só.

Além disso, se o corpo da sua função de seta for uma única instrução de retorno, mas a expressão a ser retornada for um objeto literal, você deverá colocar o objeto literal entre parênteses para evitar ambiguidade sintática entre as chaves de um corpo de função e as chaves chaves de um objeto literal:

```
const f = x => {retornar {valor: x}; };           // Bom: f()
retorna um objeto const g = x => ({               // Bom: g()
  valor: x });
retorna um objeto const h = x => {                 // Ruim: h() retorna
  valor: x };
nada const i = x => { v: x, w: x };              // Ruim: Sintaxe
                                                 Errro
```

Na terceira linha deste código, a função h() é verdadeiramente ambígua: o código que você pretendia ser um objeto literal pode ser analisado como um objeto rotulado

instrução, então uma função que retorna indefinido é criada. Na quarta linha, entretanto, o objeto literal mais complicado não é uma instrução válida e esse código ilegal causa um erro de sintaxe.

A sintaxe concisa das funções de seta as torna ideais quando você precisa passar uma função para outra função, o que é comum com métodos de array como `map()`, `filter()` e `reduzir()` (consulte §7.8.1), por exemplo:

```
// Faz uma cópia de um array com elementos nulos removidos. Deixe  
filtrado = [1,null,2,3].filter(x => x !== null); // filtrado ==  
[1,2,3]  
// Eleve ao quadrado alguns números: let  
squares = [1,2,3,4].map(x => x*x); //  
quadrados == [1, 4, 9, 16]
```

As funções de seta diferem das funções definidas de outras maneiras de uma maneira crítica: elas herdam o valor da palavra-chave `this` do ambiente em que são definidas, em vez de definir seu próprio contexto de invocação, como fazem as funções definidas de outras maneiras. Este é um recurso importante e muito útil das funções de seta, e voltaremos a ele mais adiante neste capítulo. As funções de seta também diferem de outras funções porque não possuem uma propriedade de protótipo, o que significa que não podem ser usadas como funções construtoras para novas classes (ver §9.2).

8.1.4 Funções aninhadas

Em JavaScript, as funções podem ser aninhadas em outras funções. Por exemplo:

```
função hipotenusa(a, b) {
```

```
    função quadrado(x) { return x*x; } return  
    Math.sqrt(quadrado(a) + quadrado(b));  
}
```

O interessante sobre funções aninhadas são suas regras de escopo de variáveis: elas podem acessar os parâmetros e variáveis da função (ou funções) nas quais estão aninhadas. No código mostrado aqui, por exemplo, a função interna square() pode ler e escrever os parâmetros a e b definidos pela função externa hypotenuse(). Essas regras de escopo para funções aninhadas são muito importantes e as consideraremos novamente na Seção 8.6.

8.2 Invocando Funções

O código JavaScript que compõe o corpo de uma função não é executado quando a função é definida, mas sim quando ela é invocada. As funções JavaScript podem ser invocadas de cinco maneiras:

- Como funções
- Como métodos
- Como construtores
- Indiretamente por meio de seus métodos call() e apply()
- Implicitamente, por meio de recursos da linguagem JavaScript que não aparecem como invocações normais de funções

8.2.1 Invocação de Função

As funções são invocadas como funções ou como métodos com uma expressão de invocação (§4.5). Uma expressão de invocação consiste em uma função

expressão que é avaliada como um objeto de função seguido por um parêntese de abertura, uma lista separada por vírgula de zero ou mais expressões de argumento e um parêntese de fechamento. Se a expressão da função for uma expressão de acesso à propriedade – se a função for propriedade de um objeto ou elemento de um array – então é uma expressão de invocação de método. Esse caso será explicado no exemplo a seguir. O código a seguir inclui diversas expressões de invocação de função regular:

```
printprops({x: 1}); seja total = distância (0,0,2,1) + distância  
(2,1,3,5); deixe probabilidade = fatorial(5)/fatorial(13);
```

Em uma invocação, cada expressão de argumento (aqueles entre parênteses) é avaliada e os valores resultantes tornam-se os argumentos da função. Esses valores são atribuídos aos parâmetros nomeados na definição da função. No corpo da função, uma referência a um parâmetro é avaliada como o valor do argumento correspondente.

Para invocação regular de função, o valor de retorno da função torna-se o valor da expressão de invocação. Se a função retornar porque o interpretador chega ao fim, o valor de retorno será indefinido. Se a função retornar porque o interpretador executa uma instrução return, então o valor de retorno será o valor da expressão que segue o retorno ou será indefinido se a instrução return não tiver valor.

INVOCAÇÃO CONDICIONAL

No ES2020 você pode inserir ?. após a expressão da função e antes do parêntese aberto em uma invocação de função para invocar a função somente se ela não for nula ou indefinida. Ou seja, o

expressão `f?.(x)` é equivalente (supondo que não haja efeitos colaterais) a:

```
(f !== nulo && f !== indefinido)? f(x): indefinido
```

Detalhes completos sobre esta sintaxe de invocação condicional estão em §4.5.1.

Para invocação de função em modo não estrito, o contexto de invocação (o valor `this`) é o objeto global. No modo estrito, entretanto, o contexto de invocação é indefinido. Observe que as funções definidas usando a sintaxe de seta se comportam de maneira diferente: elas sempre herdam o valor `this` que está em vigor onde são definidas.

Funções escritas para serem invocadas como funções (e não como métodos) normalmente não usam a palavra-chave `this`. A palavra-chave pode ser usada, entretanto, para determinar se o modo estrito está em vigor:

```
// Defina e invoque uma função para determinar se estamos
// no modo estrito.
const strict = (function() { return! this; })();
```

FUNÇÕES RECURSIVAS E A PILHA

Uma função recursiva é aquela, como a função `factorial()` no início deste capítulo, que chama a si mesma. Alguns algoritmos, como aqueles que envolvem estruturas de dados baseadas em árvores, podem ser implementados de maneira particularmente elegante com funções recursivas. Ao escrever uma função recursiva, entretanto, é importante pensar nas restrições de memória. Quando uma função A chama a função B e, em seguida, a função B chama a função C, o interpretador JavaScript precisa acompanhar os contextos de execução de todas as três funções. Quando a função C for concluída, o intérprete precisa saber onde retomar a execução da função B e, quando a função B for concluída, ele precisa saber onde retomar a execução da função A. Você pode imaginar esses contextos de execução como uma pilha. Quando uma função chama outra função, um novo contexto de execução é colocado na pilha. Quando essa função retorna, seu objeto de contexto de execução é retirado da pilha. Se uma função se chamar recursivamente 100 vezes, a pilha terá 100 objetos colocados nela e, em seguida, esses 100 objetos serão removidos. Essa pilha de chamadas ocupa memória. Em hardware moderno, normalmente não há problema em escrever funções recursivas que se autodenominam centenas de vezes. Mas se uma função chama a si mesma dez mil vezes, é provável que falhe com um erro como "Tamanho máximo da pilha de chamadas excedido".

8.2.2 Invocação de Método

Um método nada mais é do que uma função JavaScript armazenada em uma propriedade de um objeto. Se você tiver uma função `f` e um objeto `o`, poderá definir um método chamado `m` de `o` com a seguinte linha:

```
om = f;
```

Tendo definido o método `m()` do objeto `o`, invoque-o assim:

```
om();
```

Ou, se `m()` espera dois argumentos, você pode invocá-lo assim:

```
om(x, y);
```

O código neste exemplo é uma expressão de invocação: inclui uma expressão de função `om` e duas expressões de argumento, `x` e `y`. A expressão da função é em si uma expressão de acesso à propriedade e isso significa que a função é invocada como um método e não como uma função regular.

Os argumentos e o valor de retorno de uma invocação de método são tratados exatamente como descrito para invocação de função regular. Contudo, as invocações de método diferem das invocações de função em um aspecto importante: o contexto de invocação. As expressões de acesso à propriedade consistem em duas partes: um objeto (neste caso `o`) e um nome de propriedade (`m`). Em uma expressão de invocação de método como esta, o objeto `o` se torna o contexto de invocação, e o corpo da função pode referir-se a esse objeto usando a palavra-chave `this`. Aqui está um exemplo concreto:

```

let calculator = { // Um objeto
literal operandol: 1, operando2: 1,

    adicionar() {           // Estamos usando a sintaxe abreviada do método para
    esta função           // Observe o uso da palavra-chave this para se referir ao
    contendo objeto.
        este.resultado = este.operando1 + este.operando2; }

};

calculadora.add(); // Uma invocação de método para calcular 1+1.
calculadora.resultado // => 2

```

A maioria das invocações de métodos usa a notação de ponto para acesso a propriedades, mas expressões de acesso a propriedades que usam colchetes também causam invocação de métodos. A seguir estão as invocações de métodos, por exemplo:

```

o["m"](x,y); // Outra forma de escrever om(x,y).
uma[0](z)      // Também uma invocação de método (assumindo que a[0] é
uma função).

```

As invocações de métodos também podem envolver expressões de acesso a propriedades mais complexas:

```

cliente.sobrenome.toUpperCase(); // Invoca o método em customer.surname

f().m();                      // Invoca o método m() em
valor de retorno de f()

```

Métodos e a palavra-chave this são centrais para o paradigma de programação orientada a objetos. Qualquer função usada como método recebe efetivamente um argumento implícito – o objeto por meio do qual ela é invocada. Normalmente, um método executa algum tipo de operação naquele objeto, e a sintaxe de invocação de método é uma maneira elegante de expressar o fato de que uma função está operando em um objeto. Compare o

Duas linhas a seguir:

```
rect.setSize(largura, altura);
setRectSize(rect, largura, altura);
```

As funções hipotéticas invocadas nessas duas linhas de código podem realizar exatamente a mesma operação no objeto (hipotético) rect, mas a sintaxe de invocação de método na primeira linha indica mais claramente a ideia de que é o objeto rect que é o foco principal da operação.

ENCADEAMENTO DE MÉTODOS

Quando os métodos retornam objetos, você pode usar o valor retornado de uma invocação de método como parte de uma invocação subsequente. Isso resulta em uma série (ou "cadeia") de invocações de método como uma única expressão. Ao trabalhar com operações assíncronas baseadas em promessas (consulte o Capítulo 13), por exemplo, é comum escrever código estruturado assim:

```
Execute três operações assíncronas em sequência, manipulando
errors.doStepOne().then(doStepTwo).then(doStepThree).catch(handleErrors);
```

Quando você escreve um método que não tem um valor retornado próprio, considere fazer com que o método retorne this. Se você fizer isso de forma consistente em toda a sua API, você habilitará um estilo de programação conhecido como encadeamento de métodos no qual um objeto pode ser nomeado uma vez e, em seguida, vários métodos podem ser invocados nele:

```
new Quadrado().x(100).y(100).size(50).outline("vermelho").fill("azul").draw();
```

Observe que esta é uma palavra-chave, não um nome de variável ou propriedade. A sintaxe JavaScript não permite que você atribua um valor a isso.

A palavra-chave this não tem o escopo da mesma forma que as variáveis e, exceto para funções de seta, as funções aninhadas não herdam o valor this da função que contém. Se uma função aninhada for invocada como um método, sua

Esse valor é o objeto em que ele foi invocado. Se uma função aninhada (ou seja, não uma função de seta) for invocada como uma função, então seu valor será o objeto global (modo não estrito) ou indefinido (modo estrito). É um erro comum supor que uma função aninhada definida dentro de um método e invocada como uma função pode usar isso para obter o contexto de invocação do método. O código a seguir demonstra o problema:

```
sejaUm objeto o.m: function() { // Método m do
  objeto.
    seja self = isso; Salve o valor "this" em um
  variável.
    isso === o           => verdadeiro: "this" é o objeto o.
    f();                 Agora chame a função auxiliar
    f().
    função f() {
      isso === o         => false: "this" é global ou
    indefinido          auto === o        => verdadeiro: o eu é o exterior
    "este".
  }};

o.m();

Invoque o método m no
objeto o.
```

Dentro da função aninhada f(), a palavra-chave this não é igual ao objeto o. Isso é amplamente considerado uma falha na linguagem JavaScript, e é importante estar ciente disso. O código acima demonstra uma solução alternativa comum. Dentro do método m, atribuímos o valor this a uma variável self, e dentro da função aninhada f, podemos usar self em vez de this para nos referirmos ao containingobject.

No ES6 e posterior, outra solução alternativa para esse problema é converter a função f em uma função de seta, que herdará corretamente este valor:

```
const f = () => {
    isso === o true, uma vez que as funções de seta herdam isso
};
```

Funções definidas como expressões em vez de instruções não são içadas, portanto, para fazer esse código funcionar, a definição da função para f precisará ser movida dentro do método m para que apareça antes de ser invocada.

Outra solução alternativa é invocar o método bind() da função aninhada para definir uma nova função que é invocada implicitamente em um objeto especificado:

```
const f = (função() {
    isso === o true, uma vez que vinculamos essa função ao
    outer
    this}).bind(this);
```

Falaremos mais sobre bind() em §8.7.5.

8.2.3 Invocação do construtor

Se uma invocação de função ou método for precedida pela palavra-chave new, ela será uma invocação de construtor. (As invocações de construtores foram introduzidas em §4.6 e §6.2.2, e os construtores serão abordados com mais detalhes no Capítulo 9.) As invocações de construtor diferem das invocações de função regular e método em sua manipulação de argumentos, contexto de chamada e valor de retorno.

Se uma invocação de construtor incluir uma lista de argumentos entre parênteses, essas expressões de argumento serão avaliadas e passadas para a função da mesma forma que seriam para invocações de função e método. Não é uma prática comum, mas você pode omitir um par de parênteses vazios em uma invocação do construtor. As duas linhas seguintes, por exemplo, são equivalentes:

```
o = novo Objeto();  
o = novo objeto;
```

Uma invocação de construtor cria um objeto novo e vazio que herda do objeto especificado pela propriedade prototype do construtor. As funções de construtor destinam-se a inicializar objetos, e esse objeto recém-criado é usado como o contexto de invocação, para que a função de construtor possa se referir a ele com a palavra-chave this. Observe que o newobject é usado como o contexto de invocação, mesmo que o constructorinvocation se pareça com uma invocação de método. Ou seja, na expressão new o.m(), o não é usado como o contexto de invocação.

As funções de construtor normalmente não usam a palavra-chave return. Eles normalmente inicializam o novo objeto e retornam implicitamente quando atingem o final do corpo. Nesse caso, o novo objeto é o valor da expressão de invocação do construtor. Se, no entanto, um construtor usar explicitamente a instrução return para retornar um objeto, então thatobject se tornará o valor da expressão de invocação. Se o construtor usar return sem valor, ou se retornar um valor primitivo, esse valor retornado será ignorado e o novo objeto será usado como o valor da invocação.

8.2.4 Invocação indireta

As funções JavaScript são objetos e, como todos os objetos JavaScript, possuem métodos. Dois desses métodos, `call()` e `apply()`, invocam a função indiretamente. Ambos os métodos permitem que você especifique explicitamente esse valor para a invocação, o que significa que você pode invocar qualquer função como um método de qualquer objeto, mesmo que não seja realmente um método desse objeto. Ambos os métodos também permitem que você especifique os arguments para a invocação. O método `call()` usa sua própria lista de argumentos como argumentos para a função, e o método `apply()` espera que uma matriz de valores seja usada como argumentos. Os métodos `call()` e `apply()` são descritos em detalhes em §8.7.4.

8.2.5 Invocação de função implícita

Existem vários recursos da linguagem JavaScript que não se parecem com invocações de função, mas que fazem com que as funções sejam invocadas. Tenha muito cuidado ao escrever funções que podem ser invocadas implicitamente, porque bugs, efeitos colaterais e problemas de desempenho nessas funções são mais difíceis de diagnosticar e corrigir do que em funções regulares pela simples razão de que pode não ser óbvio a partir de uma simples inspeção do seu código quando eles estão sendo chamados.

Os recursos de linguagem que podem causar a invocação implícita de função incluem:

- Se um objeto tiver getters ou setters definidos, consultar ou definir o valor de suas propriedades poderá invocar esses métodos. Consulte §6.10.6 para obter mais informações. Quando um objeto é usado em um contexto de cadeia de caracteres (como quando é

- concatenado com uma string), seu método `toString()` é chamado. Da mesma forma, quando um objeto é usado em um contexto numérico, seu método `valueOf()` é invocado. Consulte §3.9.3 para obter
- detalhes. Quando você faz um loop sobre os elementos de um objeto iterável, há várias chamadas de método que ocorrem. O Capítulo 12 explica como os iteradores funcionam no nível de chamada de função e demonstra como escrever esses métodos para que você possa definir seus próprios tipos iteráveis. Um literal de modelo marcado é uma invocação de função disfarçada. §14.5 demonstra como escrever funções que podem ser usadas em conjunto com cadeias de caracteres literais de modelo. Os objetos proxy (descritos na seção 14.7) têm seu comportamento completamente controlado por funções. Praticamente qualquer operação em um desses objetos fará com que uma função seja invocada.

8.3 Argumentos e parâmetros da função

As definições de função JavaScript não especificam um tipo esperado para os parâmetros de função, e as invocações de função não fazem nenhuma verificação de tipo nos valores de argumento que você passa. Na verdade, as invocações de função JavaScript nem mesmo verificam o número de argumentos que estão sendo passados. As subseções a seguir descrevem o que acontece quando uma função é invocada com menos argumentos do que os parâmetros declarados ou com mais argumentos do que os parâmetros declarados. Eles também demonstram como você pode testar explicitamente o tipo de argumentos de função se precisar garantir que uma função não seja invocada com argumentos inadequados.

8.3.1 Parâmetros opcionais e padrões

Quando uma função é invocada com menos argumentos do que os declarados

parâmetros, os parâmetros adicionais são definidos com seu valor padrão, que normalmente é indefinido. Muitas vezes é útil escrever funções para que alguns argumentos sejam opcionais. Veja a seguir um exemplo:

Anexe os nomes das propriedades enumeráveis do objeto oto o// array a, e retorne a. Se a for omitido, crie e retorne um novo array.

```
function getPropertyNames(o, a) {
```

```
    se (a === indefinido) a = []; Se indefinido, use um novo
    array
    for(let propriedade em o)
        a.push(propriedade); retornar a;}
```

```
getPropertyNames() pode ser invocado com um ou dois
argumentos:let o = {x: 1}, p = {y: 2, z: 3}// Dois objetos
para testelet a = getPropertyNames(o); a == ["x"]; obter
o'sproperties em um novo arraygetPropertyNames(p, a);// a ==
["x", "y", "z"]; adicione p'sproperties a ele
```

Em vez de usar uma instrução if na primeira linha desta função, você pode usar o comando || desta forma idiomática:

```
a = a || [];
```

Lembre-se de §4.10.2 que o || retornará seu primeiro argumento se esse argumento for verdadeiro e, caso contrário, retornará seu segundo argumento. Nesse caso, se algum objeto for passado como o segundo argumento, a função usará esse objeto. Mas se o segundo argumento for omitido (ou null ou outro valor falso for passado), um array vazio recém-criado será usado.

Observe que, ao projetar funções com argumentos opcionais, você deve ter certeza de colocar os opcionais no final da lista de argumentos para que possam ser omitidos. O programador que chama sua função não pode omitir o primeiro argumento e passar o segundo: eles teriam que passar explicitamente undefined como o primeiro argumento.

No ES6 e posterior, você pode definir um valor padrão para cada um dos parâmetros da função diretamente na lista de parâmetros da função. Basta seguir o nome do parâmetro com um sinal de igual e o defaultvalue a ser usado quando nenhum argumento for fornecido para esse parâmetro:

Anexe os nomes das propriedades enumeráveis do objeto oto a// array a, e retorne a. Se a for omitido, crie e retorne um novo array.

```
function getPropertyNames(o, a = []) {
```

```
for(let propriedade em o)
  a.push(propriedade); retornar a;}
```

As expressões padrão de parâmetro são avaliadas quando sua função é chamada, não quando ela é definida, portanto, cada vez que a função thisGetPropertyNames() é invocada com um argumento, uma nova matriz vazia é criada e passada. Provavelmente é mais fácil raciocinar sobre funções se os padrões dos parâmetros forem constantes (ou expressões literais como [] e {}). Mas isso não é necessário: você pode usar variáveis ou invocações de função, por exemplo, para calcular o defaultvalue de um parâmetro. Um caso interessante é que, para funções com vários parâmetros, você pode usar o valor de um parâmetro anterior para definir o valor padrão dos parâmetros que o seguem:

Essa função retorna um objeto que representa o

```
dimensões.// Se apenas a largura for fornecida, torne-a
duas vezes mais alta do que é larga.const retângulo =
(largura, altura = largura * 2) => ({largura, altura});
retângulo (1) // => { largura: 1, altura: 2 }
```

Este código demonstra que os padrões de parâmetro funcionam com arrowfunctions. O mesmo vale para funções abreviadas de método e todas as outras formas de definições de função.

8.3.2 Parâmetros REST e listas de argumentos de comprimento variável

Os padrões de parâmetro nos permitem escrever funções que podem ser invocadas com menos argumentos do que parâmetros. Os parâmetros rest permitem o caso oposto: eles nos permitem escrever funções que podem ser invocadas com arbitrariamente mais argumentos do que parâmetros. Aqui está um exemplo função que espera um ou mais argumentos numéricos e retorna o maior:

```
function max(first=-Infinity, ... resto) {
  let maxValue = primeiro; Comece assumindo o primeiro argis
  maior
  Em seguida, percorra o resto dos argumentos, procurando
  por maiores
    for(let n de descanso) {
      if (n > maxValue) {
        maxValue = n;}}Retorna o
        maxValue biggestreturn;}
```

```
max(1, 10, 100, 2, 3, 1000, 4, 5, 6) => 1000
```

Um parâmetro rest é precedido por três pontos e deve ser o lastparameter em uma declaração de função. Quando você invoca uma função com um parâmetro est, os argumentos que você passa são primeiro atribuídos aos parâmetros não-rest e, em seguida, quaisquer argumentos restantes (ou seja, o "resto" dos argumentos) são armazenados em uma matriz que se torna o valor do parâmetro rest. Este último ponto é importante: dentro do corpo de uma função, o valor de um parâmetro rest sempre será um array. A matriz pode estar vazia, mas um parâmetro rest nunca será indefinido. (Segue-se disso que nunca é útil — e não legal — definir um parâmetro padrão para um parâmetro rest.)

Funções como o exemplo anterior que podem aceitar qualquer número de argumentos são chamadas de funções variádicas, funções de aridade variável ou funções vararg. Este livro usa o termo mais coloquial, varargs, que data dos primórdios da linguagem de programação C.

Não confunda o ... que define um parâmetro REST em uma FunctionDefinition com o ... spread, descrito em §8.3.4, que pode ser usado em invocações de função.

8.3.3 O objeto de argumentos

Os parâmetros rest foram introduzidos no JavaScript no ES6. Antes disso, na linguagem, as funções varargs eram escritas usando o objeto Arguments: dentro do corpo de qualquer função, o identificador arguments refere-se ao objeto Arguments para essa invocação. O objeto Arguments é um objeto semelhante a uma matriz (consulte §7.9) que permite que os valores dos argumentos passados para a função sejam recuperados por número, em vez de por nome. Aqui está a função max() de antes,

reescreto para usar o objeto Arguments em vez de um parâmetro rest:

```
função max(x) {  
let maxValue = -Infinity;// Percorra os  
argumentos, procurando e lembrando os maiores.  
  
    for(let i = 0; i < argumentos.comprimento; i++) {  
if (argumentos[i] > maxValue) maxValue =  
argumentos[i];}Retorna o maxValue biggestreturn;}  
  
  
max(1, 10, 100, 2, 3, 1000, 4, 5, 6) => 1000
```

O objeto Arguments remonta aos primeiros dias do JavaScript e carrega consigo uma estranha bagagem histórica que o torna ineficiente e difícil de otimizar, especialmente fora do modo estrito. Você ainda pode encontrar código que usa o objeto Arguments, mas deve evitar usá-lo em qualquer novo código que escrever. Ao refatorar código antigo, se você encontrar uma função que usa argumentos, muitas vezes poderá substituí-la por um ... args rest parâmetro. Parte do infeliz legado do objeto theArguments é que, no modo estrito, arguments é tratado como uma palavra reservada e você não pode declarar um parâmetro de função ou uma variável local com esse nome.

8.3.4 O operador de propagação para chamadas de função

O operador de spread ... é usado para descompactar ou "espalhar" os elementos de uma matriz (ou qualquer outro objeto iterável, como strings) em um contexto em que valores individuais são esperados. Vimos o spreadoperator usado com literais de matriz em §7.1.2. O operador pode ser usado, da mesma forma, em invocações de função:

```
sejam números = [5, 2, 10, -1, 9, 100,  
1]; Math.min(... números)// => -1
```

Observe que ... não é um operador verdadeiro no sentido de que não pode ser avaliado para produzir um valor. Em vez disso, é uma sintaxe JavaScript especial que pode ser usada em literais de array e invocações de função.

Quando usamos o mesmo ... sintaxe em uma definição de função em vez de uma invocação de função, ela tem o efeito oposto ao operador spread. Como vimos em §8.3.2, usar ... em uma definição de função reúne vários argumentos de função em uma matriz. Os parâmetros rest e o operador spread geralmente são úteis juntos, como na função a seguir, que recebe um argumento de função e retorna uma versão instrumentada da função para teste:

Essa função recebe uma função e retorna uma função wrappedversionfunction timed(f) {

*função de retorno(... args) { Coletar args em um descanso
Matriz de parâmetros
 console.log('Inserindo a função \${f.name}');
 deixe startTime = Date.now(); try {// Passe
 todos os nossos argumentos para o wrapper*

*função
 return f(... args); Espalhe os argumentos de volta
outra vez
 }finalme
 nte {*

*Antes de retornarmos o valor de retorno encapsulado,
 imprima o tempo decorrido.
 console.log('Saindo de \${f.name} após
 \${Date.now()-startTime}ms');*

}};}}

*Calcule a soma dos números entre 1 e n por
bruteforcefunction benchmark(n) {*

```
    seja soma = 0; for(let i = 1; i <= nou;  
    i++) soma += i;return sum;}
```

*Agora invoque a versão cronometrada desse teste
function timed(benchmark)(1000000) // => 500000500000; esta é
a soma dos números*

8.3.5 Desestruturando argumentos de função em Parameters

Quando você invoca uma função com uma lista de valores de argumento, esses valores acabam sendo atribuídos aos parâmetros declarados na definição da função. Essa fase inicial da invocação da função é muito parecida com variableassignment. Portanto, não deve ser surpreendente que possamos usar as técnicas de atribuição de desestruturação (ver §3.10.3) com funções.

Se você definir uma função que tenha nomes de parâmetro entre colchetes, estará dizendo à função para esperar que um valor de matriz seja passado para cada par de colchetes. Como parte do processo de invocação, os argumentos da matriz serão descompactados nos parâmetros individualmente nomeados. Como exemplo, suponha que estamos representando vetores 2D como matrizes de dois números, onde o primeiro elemento é a coordenada X e o segundo elemento é a coordenada Y. Com isso estrutura de dados simples, poderíamos escrever a seguinte função para adicionar dois vetores:

```
função vectorAdd(v1, v2) {  
    return [v1[0] + v2[0], v1[1] + v2[1]];
```

```
 }vectorAdd([1,2],  
 [3,4]) => [4,6]
```

O código seria mais fácil de entender se desestruturarmos os argumentos `twovector` em parâmetros nomeados com mais clareza:

```
function vectorAdd([x1,y1], [x2,y2]) { // Descompacta 2  
argumentos em 4 parâmetros  
return [x1 + x2, y1 +  
y2];}vectorAdd([1,2], [3,4])// =>  
[4,6]
```

Da mesma forma, se você estiver definindo uma função que espera um `objectargument`, poderá desestruturar os parâmetros desse objeto. Vamos usar um exemplo de vetor novamente, exceto que desta vez, vamos supor que representamos vetores como objetos com parâmetros `x` e `y`:

```
Multiplique o vetor {x,y} por uma função de  
valor escalar vectorMultiply({x, y}, escalar) {  
return { x: x*escalar, y: y*escalar  
};}multiplicarvetor({x: 1, y: 2}, 2)// => {x: 2,  
y: 4}
```

Este exemplo de desestruturação de um único argumento de objeto em dois parâmetros é bastante claro porque os nomes de parâmetro que usamos correspondem aos nomes de propriedade do objeto de entrada. A sintaxe é mais detalhada e mais confusa quando você precisa desestruturar propriedades com um nome em parâmetros com nomes diferentes. Aqui está o exemplo de `vectoraddition`, implementado para vetores baseados em objetos:

```
função vectorAdd(  
{x: x1, y: y1}, // Descompacta o 1º objeto em x1 e  
y1params  
{x: x2, y: y2} Descompacte o 2º objeto em x2 e y2
```

```
parâm  
etros  
{  
return { x: x1 + x2, y: y1 + y2  
};}vectorAdd({x: 1, y: 2}, {x: 3, y:  
4}) => {x: 4, y: 6}
```

O complicado de desestruturar a sintaxe como {x:x1, y:y1} é lembrar quais são os nomes das propriedades e quais são os nomes dos parâmetros. A regra a ter em mente para desestruturarchamadas de função de atribuição e desestruturação é que as variáveis ou parâmetros que estão sendo declarados vão para os locais onde você esperaria que os valores fossem em um literal de objeto. Portanto, os nomes das propriedades estão sempre no lado esquerdo dos dois pontos e os nomes dos parâmetros (ou variáveis) estão à direita.

Você pode definir padrões de parâmetro com parâmetros desestruturados. Aqui está a multiplicação de vetores que funciona com vetores 2D ou 3D:

```
Multiplique o vetor {x,y} ou {x,y,z} por uma função de  
valor escalar vectorMultiply({x, y, z=0}, escalar) {  
return { x: x*escalar, y: y*escalar, z: z*escalar  
};}multiplicarvetor({x: 1, y: 2}, 2)// => {x: 2, y: 4,  
z: 0}
```

Algumas linguagens (como Python) permitem que o chamador de uma função invoque uma função com argumentos especificados na forma name=value, o que é conveniente quando há muitos argumentos opcionais ou quando a lista de parâmetros é longa o suficiente para que seja difícil lembrar a ordem correta. O JavaScript não permite isso diretamente, mas você pode aproximá-lo desestruturando um argumento de objeto em seus parâmetros de função. Considere uma função que copia um número especificado de elementos de

uma matriz em outra matriz com deslocamentos iniciais especificados opcionalmente para cada matriz. Como existem cinco parâmetros possíveis, alguns dos quais têm padrões, e seria difícil para um chamador lembrar em qual ordem passar os argumentos, podemos definir e invocar a função `arraycopy()` assim:

```
function arraycopy({de, para=de,
n=de.comprimento,deÍndice=0, paraÍndice=0}) {
let valoresParaCopiar = from.slice(fromIndex, fromIndex +
n); to.splice(toIndex, 0, ... valuesToCopy); voltar
para;}seja a = [1,2,3,4,5], b = [9,8,7,6,5]; arraycopy({de:
a, n: 3, para: b, paraÍndice: 4}) // =>[9,8,7,6,1,2,3,5]
```

Ao desestruturar uma matriz, você pode definir um parâmetro `rest` para valores extras dentro da matriz que está sendo descompactada. Esse parâmetro `rest` entre colchetes é completamente diferente do parâmetro `true rest` para a função:

Essa função espera um argumento de matriz. Os dois primeiros elementos dessa matriz // são descompactados nos parâmetros `x` e `y`. Quaisquer elementos restantes// são armazenados na matriz `coords`. E quaisquer argumentos após o primeiro array// são empacotados no array restante.

```
function f([x, y, ... coords], ... resto) {
```

```
    retornar [x+y, ... descansar... coordenadas]; Nota: spread
operador aqui}f([1, 2, 3, 4], 5, 6)// => [3,
5, 6, 3, 4]
```

No ES2018, você também pode usar um parâmetro `rest` ao desestruturar um objeto. O valor desse parâmetro `rest` será um objeto que tem qualquer

propriedades que não foram desestruturadas. Os parâmetros de descanso do objeto são muitas vezes úteis com o operador de propagação do objeto, que também é um novorecurso do ES2018:

```
Multiplique o vetor {x,y} ou {x,y,z} por um valor escalar, retenha outra função propsvetorMultiply({x, y, z=0, ... props}, escalar) {
    return { x: x*escalar, y: y*escalar, z: z*escalar, ... adereços};}vectorMultiply({x: 1, y: 2, w: -1}, 2)// => {x: 2, y: 4, z:0, w: -1}
```

Por fim, lembre-se de que, além de desestruturar objetos de argumento e matrizes, você também pode desestruturar matrizes de objetos, objetos que possuem propriedades de matriz e objetos que possuem propriedades de objeto, em essencialmente qualquer profundidade. Considere o código gráfico que representa círculos como objetos com propriedades x, y, radius e color, em que a colorpropriedade é uma matriz de componentes de cores vermelho, verde e azul. Você pode definir uma função que espera que um único objeto de círculo seja passado para ela, mas desestrutura esse objeto de círculo em seis parâmetros separados:

```
function drawCircle({x, y, raio, cor: [r, g, b]}) {
    Ainda não implementado}
```

Se a desestruturação do argumento da função for mais complicada do que isso, acho que o código se torna mais difícil de ler, em vez de mais simples. Às vezes, é mais claro ser explícito sobre o acesso à propriedade do objeto e a indexação da matriz.

8.3.6 Tipos de argumento

Os parâmetros do método JavaScript não têm tipos declarados e nenhuma verificação de tipo é executada nos valores que você passa para uma função. Você pode ajudar a tornar seu código auto-documentado escolhendo nomes descritivos para argumentos de função e documentando-os cuidadosamente nos comentários de cada função. (Como alternativa, consulte §17.8 para obter uma extensão de linguagem que permite que você verifique o tipo em camadas sobre o JavaScript regular.)

Conforme descrito no §3.9, o JavaScript executa a conversão de tipo liberal conforme necessário. Portanto, se você escrever uma função que espera um argumento de string e, em seguida, chamar essa função com um valor de algum outro tipo, o valor que você passou será simplesmente convertido em uma string quando a função tentar usá-lo como uma string. Todos os tipos primitivos podem ser convertidos em strings, e todos os objetos têm métodos `toString()` (se não forem necessariamente úteis), portanto, um erro nunca ocorre neste caso.

Isso nem sempre é verdade, no entanto. Considere novamente o método `arraycopy()` mostrado anteriormente. Ele espera um ou dois argumentos de matriz e falhará se esses argumentos forem do tipo errado. A menos que você esteja escrevendo uma função privada que só será chamada de partes próximas do seu código, pode valer a pena adicionar código para verificar os tipos de argumentos como este. É melhor que uma função falhe imediatamente quando valores incorretos passados do que começar a ser executada e falhar mais tarde com uma mensagem de erro que provavelmente não será clara. Aqui está um exemplo de função que executa a verificação de tipo:

*Retorna a soma dos elementos de um objeto iterável
a.// Os elementos de a devem ser todos
números.*

```
function sum(a) {  
    seja total = 0;
```

```
for(let element of a) { // Lança TypeError se a for
notiterável
    if (elemento typeof !== "número") {
        throw new TypeError("sum(): os elementos devem ser
números");
    }total += elemento;}return total;}soma([1,2,3])// =>
6soma(1, 2, 3);// ! TypeError: 1 não é
iterablesum([1,2,"3"]); // ! TypeError: o elemento 2
não é um número
```

8.4 Funções como valores

As características mais importantes das funções são que elas podem ser definidas e invocadas. A definição e a invocação de funções são características sintáticas do JavaScript e da maioria das outras linguagens de programação. Em JavaScript, no entanto, as funções não são apenas sintaxe, mas também valores, o que significa que eles podem ser atribuídos a variáveis, armazenados nas propriedades de objetos ou elementos de matrizes, passados como argumentos para funções e assim por diante.

Para entender como as funções podem ser dados JavaScript, bem como sintaxe JavaScript, considere esta definição de função:

```
function quadrado(x) { return x*x; }
```

Essa definição cria um novo objeto de função e o atribui ao quadrado variável. O nome de uma função é realmente irrelevante; é simplesmente o nome de uma variável que se refere ao objeto de função. A função pode ser atribuída a outra variável e ainda funcionar da mesma maneira:

```
seja s = quadrado; Agora s refere-se à mesma função que  
quadrado  
fazquadrado(4)    => 16  
s(4)              => 16
```

As funções também podem ser atribuídas a propriedades de objeto em vez de variáveis. Como já discutimos, chamamos as funções de "métodos" quando fazemos isso:

```
let o = {square: function(x) { return x*x; }}; Um  
objectliterallet y = o.square(16); // y == 256
```

As funções nem precisam de nomes, como quando são atribuídas a elementos de array:

```
seja a = [x => x*x, 20]; Um array  
literal a[0](a[1]) // => 400
```

A sintaxe deste último exemplo parece estranha, mas ainda é uma expressão de invocação de função legal!

Como exemplo de como é útil tratar funções como valores, considere o método Array.sort(). Esse método classifica os elementos de uma matriz. Como há muitas ordens possíveis para classificar (ordem numérica, ordem alfabética, ordem de data, crescente, decrescente e em breve), o método sort() opcionalmente usa uma função como um argumento para dizer a ele como executar a classificação. Esta função tem um trabalho simples: para qualquer dois valores que é passado, ela retorna um valor que especifica qual elemento viria primeiro em uma matriz classificada. Este argumento de função torna Array.sort() perfeitamente geral e infinitamente flexível; ele pode classificar qualquer tipo de dados em qualquer ordem concebível. Exemplos são mostrados em

§7.8.6.

O exemplo 8-1 demonstra os tipos de coisas que podem ser feitas quando as funções são usadas como valores. Este exemplo pode ser um pouco complicado, mas os comentários explicam o que está acontecendo.

Exemplo 8-1. Usando funções como dados

```
Definimos algumas funções simples
aqui
function add(x,y) { return x + y;
} função subtrair (x, y) { return x - y;
} function multiplicar(x,y) { return x * y;
} function divide(x,y) { return x / y; }
```

Aqui está uma função que usa uma das funções anteriores// como argumento e a invoca em dois operandos
function operate(operator, operand1, operand2) {
operador de retorno(operando1,
operando2);}

Poderíamos invocar esta função assim para calcular o valor($2+3$) + ($4*5$):
let i = operar(adicionar, operar(adicionar, 2, 3), operar(multiplicar, 4, 5));

Para o exemplo, implementamos as funções simples novamente,// desta vez dentro de um objeto literal;
operadores const = {

```
adicionar: (x,y) => x+y,
subtrair: (x,y) => x-y,multiplicar: (x,y) => x*y, dividir:
(x,y) => x/y,pow:Math.pow// Isso funciona para funções
predefinidas também};
```

Essa função usa o nome de um operador, procura thatoperator// no objeto e, em seguida, o invoca nos operandos fornecidos.
Nota

```
A sintaxe usada para invocar o operador function.function  
operate2(operation, operando1, operando2) {
```

```
    if (operadores typeof[operation] === "function") {  
operadores de retorno[operação](operando1,  
operando2);}else throw "operador desconhecido";}
```

```
operar2("adicionar", "olá", operar2("adicionar", " ",  
"mundo")) // =>"olá mundo"operar2("pow", 10, 2)// => 100
```

8.4.1 Definindo suas próprias propriedades de função

As funções não são valores primitivos em JavaScript, mas um tipo de objeto especializado, o que significa que as funções podem ter propriedades.

Quando uma função precisa de uma variável "estática" cujo valor persiste entre invocações, geralmente é conveniente usar uma propriedade da própria função. Por exemplo, suponha que você queira escrever uma função que retorne um inteiro único sempre que for invocada. A função nunca deve retornar o mesmo valor duas vezes. Para gerenciar isso, a função precisa acompanhar os valores que já retornou e essas informações devem persistir entre as invocações de função. Você pode armazenar essas informações em uma variável global, mas isso é desnecessário, porque as informações são usadas apenas pela própria função. É melhor armazenar as informações em uma propriedade do objeto Function. Aqui está um exemplo que retorna um inteiro único sempre que é chamado:

*Initialize a propriedade counter do objeto de função.//
As declarações de função são içadas para que possamos
realmente // fazer essa atribuição antes da declaração de
função.uniqueInteger.counter = 0;*

*Essa função retorna um inteiro diferente cada vez que é
chamada.*

```
Ele usa uma propriedade de si mesmo para lembrar o próximo valor a ser retornado.function uniqueInteger() {
```

```
    retorna uniqueInteger.counter++; Retorno e incremento propriedade do contador}unique  
    Integer()           => 0  
    uniqueInteger()    => 1
```

Como outro exemplo, considere a seguinte função factorial() que usa propriedades de si mesma (tratando-se como uma matriz) para armazenar em cache os resultados previamente calculados:

```
Calcule fatoriais e armazene em cache os resultados como propriedades da própria função.function factorial(n) {
```

```
    if (Number.isInteger(n) && n > 0) {Positivo  
        somente inteiros  
        se (!(n em factorial)) {Se não  
            resultado em cache  
            factorial[n] = n * factorial(n-1);Calcular  
            e armazená-lo em cache  
            }ffatorial de  
            retorno[n];Retornar  
            O resultado armazenado em cache  
        } else {  
            return NaN;// Se a entrada  
            foi ruim  
        }  
    } factorial[1] =  
    1;Initialize o cache para manter essa base  
    caso.fatori  
    al(6)      => 720  
    factorial[5]  => 120;A chamada acima armazena esse valor em cache
```

8.5 Funções como namespaces

As variáveis declaradas dentro de uma função não são visíveis fora do

função. Por esse motivo, às vezes é útil definir uma função simplesmente para atuar como um namespace temporário no qual você pode definir variáveis sem sobrecarregar o namespace global.

Suponha, por exemplo, que você tenha um pedaço de código JavaScript que deseja usar em vários programas JavaScript diferentes (ou, para JavaScript do lado do cliente, em várias páginas da Web diferentes). Suponha que este código, como a maioria dos códigos, defina variáveis para armazenar os resultados intermediários de sua computação. O problema é que, como esse pedaço de código será usado em muitos programas diferentes, você não sabe se as variáveis que ele cria entrarão em conflito com as variáveis criadas pelos programas que o utilizam. A solução é colocar o pedaço de código em uma função e, em seguida, invocar a função. Dessa forma, as variáveis que seriam globais tornam-se locais para a função:

```
function chunkNamespace() {  
    Pedaço de código vai aqui// Todas as variáveis definidas  
    no pedaço são locais para this function  
  
    em vez de bagunçar o namespace  
    global.}chunkNamespace();// Mas não se esqueça de  
    invocar a função!
```

Esse código define apenas uma única variável global: a função namechunkNamespace. Se definir até mesmo uma única propriedade for demais, você poderá definir e invocar uma função anônima em uma única expressão:

```
(função() {    chunkNamespace() reescrita como uma função  
    expressão sem nome.  
    Pedaço de código vai aqui})();// Finalize a função literal e  
    invoque-a agora.
```

Essa técnica de definir e invocar uma função em uma única expressão é usada com frequência suficiente para se tornar idiomática e receber o nome de "expressão de função imediatamente invocada". Observe o uso de parênteses no exemplo de código anterior. O openparenthesis antes da função é necessário porque, sem ele, o interpretador JavaScript tenta analisar a palavra-chave function como uma instrução de declaração de função. Com o parêntese, o interpretador reconhece corretamente isso como uma expressão de definição de função. O parêntese principal também ajuda os leitores humanos a reconhecer quando uma função está sendo definida para ser imediatamente invocada em vez de definida para uso posterior.

Esse uso de funções como namespaces torna-se realmente útil quando definimos uma ou mais funções dentro da função namespace usando variáveis dentro desse namespace, mas depois as passamos de volta como valor de retorno da função namespace. Funções como essa são conhecidas como encerramentos e são o tópico da próxima seção.

8.6 Fechamentos

Como a maioria das linguagens de programação modernas, o JavaScript usa escopo léxico. Isso significa que as funções são executadas usando o variablescope que estava em vigor quando foram definidas, não a variável scopeque está em vigor quando são invocadas. Para implementar o escopo léxico, o estado interno de um objeto de função JavaScript deve incluir não apenas o código da função, mas também uma referência ao escopo no qual a definição da função aparece. Essa combinação de um functionobject e um escopo (um conjunto de associações de variáveis) no qual o objeto

variáveis são resolvidas é chamado de fechamento na literatura de ciência da computação.

Tecnicamente, todas as funções JavaScript são encerramentos, mas como a maioria das funções é invocada do mesmo escopo em que foram definidas, normalmente não importa que haja um encerramento envolvido. Os encerramentos tornam-se interessantes quando são invocados de um escopo diferente daquele em que foram definidos. Isso acontece mais comumente quando um objeto de função aninhado é retornado da função na qual foi definido. Existem várias técnicas de programação poderosas que envolvem esse tipo de fechamento de função aninhada, e seu uso se tornou relativamente comum na programação JavaScript. Os fechamentos podem parecer confusos quando você os encontra pela primeira vez, mas é importante que você os entenda bem o suficiente para usá-los confortavelmente.

A primeira etapa para entender os encerramentos é revisar as regras de escopo lexical para funções aninhadas. Considere o seguinte código:

```
let escopo = "escopo global";           Uma variável global
função checkscope() {
    let escopo = "escopo local";       Uma variável local
    function f() { return scope; }     Retornar o valor em
    escopo aqui
    return
    f();}checkscop
e()                                     => "escopo local"
```

A função checkscope() declara uma variável local e, em seguida, define e invoca uma função que retorna o valor dessa variável. Deve ficar claro para você por que a chamada para checkscope() retorna "localscope". Agora, vamos mudar um pouco o código. Você pode dizer o que isso

o código retornará?

```
let escopo = "escopo global";           Uma variável global
função checkscope() {
    let escopo = "escopo local";       Uma variável local
        function f() { return scope; }   Retornar o valor em
escopo aqui
retornar f; }seja s =
checkscope();
retornar?                                         O que isso significa
```

Neste código, um par de parênteses foi movido de insidecheckscope() para fora dele. Em vez de invocar a função aninhada e retornar seu resultado, checkscope() agora apenas retorna o próprio objeto de função thenested. O que acontece quando invocamos essa função aninhada (com o segundo par de parênteses na última linha de código) fora da função na qual ela foi definida?

Lembre-se da regra fundamental do escopo léxico: JavaScriptsão executadas usando o escopo em que foram definidas. A função thenested f() foi definida em um escopo onde a variável scopewas vinculada ao valor "local scope". Essa ligação ainda está em vigor quando f é executado, não importa de onde seja executado. Portanto, a última linha do exemplo de código anterior retorna "escopo local", não "globalscope". Isso, em poucas palavras, é a natureza surpreendente e poderosa dos encerramentos: eles capturam as ligações de variáveis locais (e parâmetros) da função externa dentro da qual são definidos.

Em §8.4.1, definimos uma função uniqueInteger() que usava uma propriedade da própria função para acompanhar o próximo valor a ser

Retornado. Uma falha dessa abordagem é que o código com bugs ou malicioso pode redefinir o contador ou defini-lo como um não inteiro, fazendo com que a função uniqueInteger() viole a parte "unique" ou "integer" de seu contrato. Os encerramentos capturam as variáveis locais de uma invocação de função única e podem usar essas variáveis como estado privado. Aqui está como poderíamos reescrever o uniqueInteger() usando uma expressão de função imediatamente invocada para definir um namespace e um fechamento que usa esse namespace para manter seu estado privado:

```
let únicoInteiro = (função() {      Definir e chamar
    let contador = 0;                  Estado privado de
    função abaixo
    return function() { return counter++; };}
()); uniqueInteger()// =>
uniqueInteger()// => 1
```

Para entender este código, você deve lê-lo com atenção. À primeira vista, a primeira linha de código parece estar atribuindo uma função à variável uniqueInteger. Na verdade, o código está definindo e invocando (conforme sugerido pelo parêntese aberto na primeira linha) uma função, portanto, é o valor retornado da função que está sendo atribuída a uniqueInteger. Agora, se estudarmos o corpo da função, veremos que seu valor de retorno é outra função. É esse objeto de função aninhado que é atribuído a uniqueInteger. A função aninhada tem acesso às variáveis em seu escopo e pode usar a variável de contador definida na função externa. Uma vez que a função externa retorna, nenhum outro código pode ver a variável do contador: a função interna tem acesso exclusivo a ela.

Variáveis privadas como counter não precisam ser exclusivas de um único encerramento: é perfeitamente possível que duas ou mais funções aninhadas sejam definidas dentro da mesma função externa e compartilhem o mesmo escopo. Considere o seguinte código:

```
função contador() {  
    seja n = 0;  
    retornar {  
        count: function() { return n++; }, reset:  
        function() { n = 0; }};}  
  
seja c = contador(), d = contador();      Criar dois contadores  
c.count()                                => 0  
d.count()                                => 0: eles contam  
independente  
lyc.reset();                            reset() e count()  
métodos compartilham  
statec.count()                           => 0: porque nós reiniciamos  
cd.count  
()                                     => 1: d não foi reposto
```

A função counter() retorna um objeto "counter". Este objeto tem dois métodos: count() retorna o próximo inteiro e reset() redefine o estado interno. A primeira coisa a entender é que os dois métodos compartilham o acesso à variável privada n. A segunda coisa a entender é que cada invocação de counter() cria um novo escopo, independente dos escopos usados por invocações anteriores, e uma nova variável privada dentro desse escopo. Portanto, se você chamar counter() duas vezes, obterá dois objetos counter com variáveis privadas diferentes. Chamar count() ou reset() em um objeto contador não tem efeito sobre o outro.

Vale a pena notar aqui que você pode combinar essa técnica de fechamento com getters e setters de propriedades. A seguinte versão da função counter() é uma variação do código que apareceu em §6.10.6, mas usa encerramentos para estado privado em vez de depender de uma propriedade regular object:

```
função contador(n) {    O argumento da função n é o
    retorno
    variável {
        Retornos e incrementos do método getter de propriedade
        contador privado var.
        get count() { return n++; },// O setter de
        propriedade não permite que o valor de n
        diminuir
        definir contagem (m) {
            if (m > n) n = m;else throw Error("count só
            pode ser definido como um
            valor maior");
    };}
}

seja c = contador(1000); c.count// => 1000c.count// =>
1001c.count = 2000; c.count// => 2000c.count = 2000;///
! Erro: a contagem só pode ser definida como um valor
maior
```

Observe que esta versão da função counter() não declara uma variável local, mas apenas usa seu parâmetro n para manter o estado privado compartilhado pelos métodos do acessador de propriedade. Isso permite que o chamador ofcounter() especifique o valor inicial da variável privada.

O exemplo 8-2 é uma generalização do estado privado compartilhado por meio do

técnicas de fechamento que demonstramos aqui. Este exemplo define uma função addPrivateProperty() que define uma privatevariable e duas funções aninhadas para obter e definir o valor dessa variável. Ele adiciona essas funções aninhadas como métodos do objeto especificado.

Exemplo 8-2. Métodos de acessador de propriedade privada usando closures

Esta função adiciona métodos de acessador de propriedade para uma propriedade com// o nome especificado ao objeto o. Os métodos são nomeados get<name> e set<name>. Se uma função de predíicado é fornecida, o método thesetter// a usa para testar a validade de seu argumento antes de armazená-lo.// Se o predíicado retorna false, o método setter lança uma exceção.//// A coisa incomum sobre esta função é que o propertyvalue// que é manipulado pelos métodos getter e setter não é armazenado no// objeto o. Em vez disso, o valor é armazenado apenas em uma variável local// nesta função. Os métodos getter e setter também são definidos// localmente para esta função e, portanto, têm acesso a thislocal variable.// Isso significa que o valor é privado para os dois métodos acessadores, e ele// não pode ser definido ou modificado, exceto através do método setter.function addPrivateProperty(o, name, predicate) {

```
    deixe      Este é o valor da propriedade
    valor;
    O método getter simplesmente retorna o
    value.o['get${name}'] = function() { return value; };
```

O método setter armazena o valor ou lança uma exceção se

```
    O predíicado rejeita o
    valor.o['set${name}'] = function(v) {
        if (predíicado && !predíicado(v)) {
```

```
        throw new TypeError('set${name}: valor inválido  
${v}');  
    } else {  
valor = v;}};}
```

O código a seguir demonstra o
`addPrivateProperty()` method. `let o = {};// Aqui está um`
`objeto vazio`

`Adicionar métodos de acesso de propriedade getName e setName()//`
`Certifique-se de que apenas valores de string sejam`
`permitidos addPrivateProperty(o, "Name", x => typeof x === "string");`

`o.setName("Frank"); Definir o valor da propriedade`
~~`o.getName();`~~`// ! TypeError: tente definir um valor`
`do tipo errado`

Vimos agora vários exemplos em que dois encerramentos são definidos no mesmo escopo e compartilham o acesso à mesma variável ou variáveis privadas. Esta é uma técnica importante, mas é igualmente importante reconhecer quando os fechamentos inadvertidamente compartilham o acesso a uma variável que não deveriam compartilhar. Considere o seguinte código:

Esta função retorna uma função que sempre retorna v
`function constfunc(v) { return () => v; }`

Crie uma matriz de funções constantes:
`let funcs = [];`
`for(var i = 0; i < 10; i++) funcs[i] =`
`constfunc(i);`

A função no elemento 5 do array retorna o valor 5.
`funcs[5]// => 5`

Ao trabalhar com código como este que cria vários encerramentos usando aloop, é um erro comum tentar mover o loop dentro da função

que define os fechamentos. Pense no seguinte código, por exemplo:

```
Retorna um array de funções que retornam os valores 0-9
function constfuncs() {
  let funcs = []; for(var i =
  0; i < 10; i++) {
  funcs[i] = () =>
  i;}funções de retorno;}

let funcs = constfuncs(); funcs[5]/// => 10; Por
que isso não retorna 5?
```

Esse código cria 10 encerramentos e os armazena em uma matriz. Os closures são todos definidos dentro da mesma invocação da função, portanto, eles compartilham acesso à variável i. Quando constfuncs() retorna, o valor da variável i é 10 e todos os 10 fechamentos compartilham esse valor. Portanto, todas as funções na matriz de funções retornada retornam o mesmo valor, que não é o que queríamos. É importante lembrar que o escopo associado a um fechamento é "ativo". As funções aninhadas não fazem cópias privadas do escopo nem fazem instantâneos estáticos das associações de variáveis. Fundamentalmente, o problema aqui é que as variáveis declaradas com var são definidas em toda a função. Nosso loop for declara a variável de loop com var i, então a variável i é definida em toda a função em vez de ter um escopo mais restrito ao corpo do loop. O código demonstra uma categoria comum de bugs no ES5 e antes, mas a introdução de variáveis de escopo de bloco no ES6 resolve o problema. Se apenas substituirmos o var por um let ou um const, o problema desaparece. Como let e const têm escopo de bloco, cada iteração do loop define um escopo independente dos escopos

para todas as outras iterações, e cada um desses escopos tem sua própria associação independente de i.

Outra coisa a lembrar ao escrever encerramentos é que esta é uma palavra-chave JavaScript, não uma variável. Conforme discutido anteriormente, arrowfunções herdam o valor this da função que as contém, masas funções definidas com a palavra-chave function não. Portanto, se você estiver reescrevendo um fechamento que precisa usar o valor this de sua containingfunction, você deve usar uma função de seta, ou chamar bind(), no closure antes de retorná-lo, ou atribuir o valor this externo a uma variável que seu encerramento herdará:

```
const self = isso; Disponibilize esse valor para  
Funções aninhadas
```

8.7 Propriedades da função, métodos e construtor

Vimos que funções são valores em programas JavaScript. O operador typeof retorna a string "function" quando aplicado a uma função, mas as funções são realmente um tipo especializado de objeto JavaScript. Como as funções são objetos, elas podem ter propriedades e métodos, assim como qualquer outro objeto. Existe até um construtor Function() para criar novos objetos de função. As subseções a seguir documentam as propriedades de comprimento, nome e protótipo; os métodos call(), apply(), bind() e toString(); e o construtor Function().

8.7.1 A propriedade length

A propriedade `length` somente leitura de uma função especifica a aridade da função — o número de parâmetros que ela declara em sua lista de parâmetros, que geralmente é o número de argumentos que a função espera. Se uma função tiver um parâmetro rest, esse parâmetro não será contado para os propósitos dessa propriedade `length`.

8.7.2 O nome Propriedade

A propriedade de nome somente leitura de uma função especifica o nome que foi usado quando a função foi definida, se ela foi definida com um nome ou o nome da variável ou propriedade à qual uma expressão de função sem nome foi atribuída quando foi criada pela primeira vez. Essa propriedade é útil principalmente ao escrever mensagens de depuração ou erro.

8.7.3 A propriedade do protótipo

Todas as funções, exceto as funções de seta, têm uma propriedade `prototype` que se refere a um objeto conhecido como objeto `prototype`. Cada função tem um objeto protótipo diferente. Quando uma função é usada como um construtor, o objeto recém-criado herda propriedades do objeto `prototype`. Os protótipos e a propriedade do protótipo foram discutidos no ponto 6.2.3 e serão abordados novamente no Capítulo 9.

8.7.4 Os métodos `call()` e `apply()`

`call()` e `apply()` permitem que você invoque indiretamente (§8.2.4) uma função como se fosse um método de algum outro objeto. O primeiro argumento `call()` e `apply()` é o objeto no qual a função deve ser invocada; Esse argumento é o contexto de invocação e se torna o valor da palavra-chave `this` dentro do corpo da função. Para invocar

a função f() como um método do objeto o (sem passar argumentos), você pode usar call() ou apply():

```
f.call(o); f.  
aplicar (o);
```

Qualquer uma dessas linhas de código é semelhante à seguinte (que pressupõe que o ainda não tenha uma propriedade chamada m):

```
o.m = f;      Faça de f um método temporário de o.  
o.m();        Invoque-o, sem passar argumentos.  
excluir o.m; Remova o método temporário.
```

Lembre-se de que as funções de seta herdam o valor this do contexto onde são definidas. Isso não pode ser substituído pelos métodos call() e apply(). Se você chamar qualquer um desses métodos em uma função estreita, o primeiro argumento será efetivamente ignorado.

Quaisquer argumentos para call() após o primeiro argumento de contexto de invocações são os valores que são passados para a função que é invocada (e esses argumentos não são ignorados para funções de seta). Por exemplo, para passar dois números para a função f() e invocá-la como se fosse um método do objeto o, você pode usar um código como este:

```
f.call(o, 1, 2);
```

O método apply() é como o método call(), exceto que os argumentos a serem passados para a função são especificados como um array:

```
f.apply(o, [1,2]);
```

Se uma função é definida para aceitar um número arbitrário de argumentos, o método apply() permite que você invoque essa função no conteúdo de uma matriz de comprimento arbitrário. No ES6 e posterior, podemos apenas usar o operador de propagação, mas você pode ver o código ES5 que usa apply() em vez disso. Por exemplo, para encontrar o maior número em uma matriz de números sem usar o operador de propagação, você pode usar o método apply() para passar os elementos da matriz para a função Math.max():

```
let maior = Math.max.apply(Matemática, matrizDeNúmeros);
```

A função trace() definida a seguir é semelhante à função timed() definida em §8.3.4, mas funciona para métodos em vez de funções. Ele usa o método apply() em vez de um operador spread e, ao fazer isso, é capaz de invocar o método encapsulado com os mesmos argumentos e o mesmo valor que o método wrapper:

Substitua o método chamado m do objeto o por uma versão que registra// mensagens antes e depois de invocar o método original.

```
function trace(o, m) {  
    seja original = o[m];          Lembre-se do método original  
    no fechamento.  
    o[m] = função(... args) {      Agora defina o novo  
        método.  
        console.log(new Date(), "Inserindo:", m);      Tora  
        Mensagem.  
        let resultado = original.apply(this, args); //  
        Invocar original.  
        console.log(new Date(), "Saindo:", m);      Tora  
        Mensagem.  
        resultado de retorno;                      //  
        Resultado retornado.  
    };}
```

8.7.5 O método bind()

O objetivo principal de bind() é vincular uma função a um objeto.

Quando você invoca o método bind() em uma função f e passa um objeto o, o método retorna uma nova função. Invocar a nova função (como uma função) invoca a função original f como um método de o.

Quaisquer argumentos que você passar para a nova função são passados para a função original. Por exemplo:

```
function f(y) { return this.x + y; } // Esta função precisa
ser vinculada let o = { x: 1 }; // Um objeto que vamos
bindtolet g = f.bind(o); // Chamando g(x) invokesf() on
og(2); // => 3let p = { x: 10, g }; // Invocando g() como um
método deste objetop.g(2); // => 3: g ainda está ligado a o,
não a p.
```

As funções de seta herdam seu valor this do ambiente no qual são definidas, e esse valor não pode ser substituído por bind(), portanto, se a função f() no código anterior fosse definida como uma função de seta, a vinculação não funcionaria. O caso de uso mais comum para chamar bind() é fazer com que funções que não sejam de seta se comportem como funções de seta, no entanto, essa limitação na vinculação de funções de seta não é um problema na prática.

O método bind() faz mais do que apenas vincular uma função a um objeto, no entanto. Ele também pode executar uma aplicação parcial: quaisquer argumentos que você passar para bind() após o primeiro são vinculados junto com o valor this.

Este recurso de aplicação parcial de bind() funciona com arrowfunctions. A aplicação parcial é uma técnica comum na programação funcional e às vezes é chamada de currying. Aqui estão alguns exemplos do método bind() usado para aplicação parcial:

```
seja soma = (x,y) => x + y;      Retorna a soma de 2 argumentos
let succ = sum.bind(null, 1); // Vincule o primeiro
argumento a 1 // => 3: x está vinculado a 1 e passamos
2 para o argumento y

function f(y,z) { return this.x + y + z; } let g = f.bind({x:
1}, 2); // Vincule this e y g(3) // => 6: this.x está ligado a
1, y está ligado a 2 e z é 3
```

A propriedade name da função retornada por bind() é a propriedade named da função que bind() foi chamada, prefixada com a palavra "bound".

8.7.6 O método `toString()`

Como todos os objetos JavaScript, as funções têm um método `toString()`. A especificação ECMAScript requer que esse método retorne uma cadeia de caracteres que segue a sintaxe da instrução de declaração de função. Na prática, a maioria (mas não todas) implementações deste método `toString()` retornam o código-fonte completo para a função. As funções internas normalmente retornam uma cadeia de caracteres que inclui algo como "[código nativo]" como o corpo da função.

8.7.7 O construtor `Function()`

Como as funções são objetos, há um construtor `Function()`

que podem ser usadas para criar novas funções:

```
const f = new Function("x", "y", "return x*y;");
```

Essa linha de código cria uma nova função que é mais ou menos equivalente a uma função definida com a sintaxe familiar:

```
const f = function(x, y) { return x*y; };
```

O construtor `Function()` espera qualquer número de string arguments. O último argumento é o texto do corpo da função; ele pode conter instruções JavaScript arbitrárias, separadas umas das outras por ponto e vírgula. Todos os outros argumentos para o construtor são cadeias de caracteres que especificam os nomes de parâmetro para a função. Se você estiver definindo uma função que não aceita argumentos, basta passar uma única cadeia de caracteres — o corpo da função — para o construtor.

Observe que o construtor `Function()` não recebe nenhum argumento que especifique um nome para a função que ele cria. Como literais de função, o construtor `Function()` cria funções anônimas.

Existem alguns pontos que são importantes para entender sobre o construtor `theFunction()`:

- O construtor `Function()` permite que as funções JavaScript sejam criadas e compiladas dinamicamente em tempo de execução. O construtor `Function()` analisa o corpo da função e cria um novo objeto de função cada vez que é chamado. Se o callto o construtor aparecer dentro de um loop ou dentro de uma função chamada com frequência, esse processo poderá ser ineficiente. Em contrapartida,

- Funções aninhadas e expressões de função que aparecem em loops não são recompiladas sempre que são encontradas. Um último ponto muito importante sobre o construtor Function() é que as funções que ele cria não usam escopo léxico; em vez disso, eles são sempre compilados como se fossem funções de nível superior, como demonstra o código a seguir:

```
let escopo = "global"; função  
constructFunction() {  
    let escopo = "local"; return new  
    Function("return scope");// Não captura  
    o escopo local!}Esta linha retorna  
    "global" porque a função retornada pelo  
    construtor // Function() não usa o  
    escopo local.constructFunction()()// =>  
    "global"
```

O construtor Function() é melhor pensado como uma versão de escopo global de eval() (consulte §4.12.2) que define novas variáveis e funções em seu próprio escopo privado. Você provavelmente nunca precisará usar esse construtor em seu código.

8.8 Programação Funcional

JavaScript não é uma linguagem de programação funcional como Lisp ou Haskell, mas o fato de que JavaScript pode manipular funções como objetos significa que podemos usar técnicas de programação funcional em JavaScript. Métodos de array como map() e reduce() emprestam

particularmente bem a um estilo de programação funcional. As seções a seguir demonstram técnicas de programação funcional em JavaScript. Eles pretendem ser uma exploração de expansão da mente do poder das funções do JavaScript, não como uma receita para um bom estilo de programação.

8.8.1 Processando matrizes com funções

Suponha que temos uma matriz de números e queremos calcular o temaan e o desvio padrão desses valores. Podemos fazer esse estilo não funcional assim:

```
let dados = [1,1,3,5,5]; Este é o nosso conjunto de números
A média é a soma dos elementos dividida pelo número de
elementos
let total = 0; for(let i = 0; i < data.length;
i++) total += data[i]; let média = total/data.length;//
média == 3; A média dos nossos dados é 3
```

Para calcular o desvio padrão, primeiro somamos os quadrados de// o desvio de cada elemento da média.
total = 0; for(let i = 0; i < data.length; i++)
{

```
let desvio = data[i] - média; total += desvio
* desvio;}let stddev =
Math.sqrt(total/(data.length-1));
stddev ==
```

2

Podemos realizar esses mesmos cálculos em estilo funcional conciso usando os métodos de matriz map() e reduce() desta forma ([consulte §7.8.1](#) para revisar esses métodos):

```
Primeiro, defina duas funções
simplestconst sum = (x,y) => x+y;
const quadrado = x => x * x;
```

```
Em seguida, use essas funções com métodos Array para
calcular dados médios e padrão = [1,1,3,5,5]; let média =
data.reduce(sum)/data.length;// média == 3let desvios =
data.map(x => x-mean); let stddev
=Math.sqrt(desvios.map(quadrado).reduce(soma)/(data.length-1)); stddev// => 2
```

Esta nova versão do código parece bem diferente da primeira, mas ainda está invocando métodos em objetos, portanto, tem algumas convenções orientadas a objetos restantes. Vamos escrever versões funcionais dos métodos themap() e reduce():

```
const map = function(a, ... args) { return a.map(... args);
}; const reduzir = função(a, ... args) { returna.reduce(... args); };
```

Com essas funções map() e reduce() definidas, nosso código para calcular a média e o desvio padrão agora fica assim:

```
soma const = (x,y) => x+y;
const quadrado = x => x * x;

let dados = [1,1,3,5,5]; let mean = reduce(dados,
soma)/data.length; let desvios = map(dados, x => x-
média); let stddev = Math.sqrt(reduce(map(desvios,
quadrado),soma)/(data.length-1)); stddev// => 2
```

8.8.2 Funções de ordem superior

Uma função de ordem superior é uma função que opera em funções, tomando uma ou mais funções como argumentos e retornando uma nova função. Aqui está um exemplo:

Essa função de ordem superior retorna uma nova função que passa seus// argumentos para f e retorna a negação lógica do valor de retorno de f; function not(f) {

 função de retorno(... args) { *Devolver um novo*
 função
 que é a negação lógica do resultado da função f para os argumentos fornecidos.
 };}

const par = x => x % 2 === 0; *Uma função para determinar se*
um número é par const ímpar = não(par);*// Uma nova função*
que faz o oposto[1,1,3,5,5].cada(ímpar)*// => verdadeiro:*
cada elemento da matriz é ímpar

Esta função not() é uma função de ordem superior porque recebe um argumento de função e retorna uma nova função. Como outro exemplo, considere a função mapper() a seguir. Ele pega um argumento de função e retorna uma nova função que mapeia uma matriz para outra usando essa função. Esta função usa a função map() definida anteriormente, e é importante que você entenda como as duas funções são diferentes:

Retorna uma função que espera um argumento de array e
aplica f a// cada elemento, retornando o array de
valores de retorno.// Compare isso com a função map()
de anteriormente.function mapper(f) {

```
retorna um => mapa(a,  
f);}  
  
const incremento = x => x+1; const  
incrementoTodos = mapeador(incremento);  
incrementAll([1,2,3])// => [2,3,4]
```

Aqui está outro exemplo, mais geral, que pega duas funções, f e g, e retorna uma nova função que calcula f(g()):

Retorna uma nova função que calcula f(g(...)). A função retornada h passa todos os seus argumentos para g, depois passa// o valor de retorno de g para f, depois retorna o valor de retorno de f.// Tanto f quanto g são invocados com o mesmo valor this que h foi invocado com.

```
function compose(f, g) {
```

```
    função de retorno(... args) {  
        Usamos call para f porque estamos passando um único  
        valor e  
        aplicar para g porque estamos passando uma matriz de  
        Valores.  
        return f.call(this, g.apply(this, args));};}
```

```
soma const = (x,y) => x+y;  
const quadrado = x => x *  
x; compor (quadrado, soma) => 25; o quadrado da soma  
(2,3)
```

As funções partial() e memoize() definidas nas seções a seguir são duas funções de ordem superior mais importantes.

8.8.3 Aplicação Parcial de Funções

O método bind() de uma função f (consulte §8.7.5) retorna uma newfunction que invoca f em um contexto especificado e com um conjunto especificado

de argumentos. Dizemos que ele liga a função a um objeto e aplica parcialmente os argumentos. O método bind() aplica parcialmente argumentos à esquerda, ou seja, os argumentos que você passa para bind() são substituídos no início da lista de argumentos que é passada para a função original. Mas também é possível aplicar parcialmente argumentos à direita:

Os argumentos para esta função são passados na função leftpartialLeft(f, ... outerArgs) {

```
    função de retorno(... innerArgs) { // Retorna esta função
        let args = [... outerArgs, ... innerArgs]; Construa o
        lista de argumentos
        return f.apply(this, args);           Então
    invocar f com isso
};}
```

Os argumentos para esta função são passados na função direitaDireita(f, ... outerArgs) {

```
    função de retorno(... innerArgs) { Retornar esta função
        let args = [... innerArgs, ... outerArgs]; Construa o
        lista de argumentos
        return f.apply(this, args);           Então
    invocar f com isso
};}
```

Os argumentos para essa função servem como um modelo. Valores indefinidos// na lista de argumentos são preenchidos com valores do conjunto interno. função parcial(f, ... outerArgs) {

```
    função de retorno(... innerArgs) {
        let args = [... outerArgs]; cópia local do exterior
        modelo de argumentos
        deixe innerIndex = 0;      qual argumento interno é o próximo
        Percorra os argumentos, preenchendo valores indefinidos
        de argumentos internos
        for(let i = 0; i < args.length; i++) {
```

```
        if (args[i] === indefinido) args[i] =  
    innerArgs[innerIndex++];  
    // Agora anexe quaisquer argumentos internos  
    restantes args.push(...  
    innerArgs.slice(innerIndex)); return f.apply(this,  
    args);}};
```

Aqui está uma função com três argumentos const $f =$
 $function(x,y,z) \{ return x * (y - z); \};$ // Observe como
essas três aplicações parciais diferem partialLeft($f, 2$)
 $(3,4)// \Rightarrow -2$: Vincula o primeiro argumento: $2 * (3 -$
 $4)$ partialRight($f, 2$) $(3,4)// \Rightarrow 6$: Vincula o último
argumento: $3 * (4 - 2)$ partial($f, undefined, 2$) $(3,4)// \Rightarrow -6$:
Vincula o argumento do meio: $3 * (2 - 4)$

Essas funções parciais do aplicativo nos permitem definir facilmente funções interessantes a partir de funções que já definimos. Aqui estão alguns exemplos:

```
const incremento = partialLeft(soma, 1);  
const raiz cúbica = partialRight(Math.pow,  
1/3); Raiz cúbica(incremento(26))// \Rightarrow 3
```

A aplicação parcial torna-se ainda mais interessante quando a combinamos com outras funções de ordem superior. Aqui, por exemplo, está uma maneira de definir a função not() anterior que acabamos de mostrar usando a composição e aplicação parcial:

```
const not = partialLeft(compor, x => !x);  
const par = x => x % 2 === 0; const ímpar  
= não(par); const isNumber = not(isNaN);  
ímpar(3) && éNúmero(2)// \Rightarrow verdadeiro
```

Também podemos usar a composição e a aplicação parcial para refazer nossos cálculos de desvio padrão de média em estilo funcional extremo:

```
As funções sum() e square() são definidas acima. Aqui
estão mais alguns: const produto = (x, y) => x * y; const
neg = parcial(produto, -1); const sqrt = parcial(Math.pow,
indefinido, .5); const recíproco = parcial(Math.pow,
indefinido, neg(1));
```

```
Agora calcule a média e o desvio padrão.let
dados = [1,1,3,5,5];// Nossa média de dados =
produto(reduzir(dados,
soma),recíproco(dados.comprimento)); let
stddev = sqrt(product(reduce(map(data,
compor(quadrado,
parcial(soma,
neg(média)))),
soma),
recíproco(soma(dados.comprimento,neg(
1))))); [média, stddev]// => [3, 2]
```

Observe que esse código para calcular a média e o desvio padrão é inteiramente invocações de função; não há operadores envolvidos e o número de parênteses cresceu tanto que este JavaScript está começando a se parecer com o código Lisp. Novamente, este não é um estilo que eu defendo para a programação JavaScript, mas é um exercício interessante para ver o quanto profundamente funcional o código JavaScript pode ser.

8.8.4 Memorização

Na seção 8.4.1, definimos uma função factorial que armazenava em cache seus resultados calculados anteriormente. Na programação funcional, esse tipo de cache é chamado de memoização. O código a seguir mostra uma ordem superior

função, memoize(), que aceita uma função como argumento e retorna uma versão memoizada da função:

Retorna uma versão memoizada de f.// Ele só funciona se todos os argumentos para f tiverem stringrepresentations distintos.

```
function memoize(f) {  
  const cache = new Map(); Cache de valores armazenado no encerramento.  
  
  function de retorno(... args) {  
    Crie uma versão de cadeia de caracteres dos argumentos a serem usados como a cache key.  
    let chave = args.length +  
      args.join("+"); if (cache.has(chave)) {  
      return cache.get(chave);}  
    else {  
      let resultado = f.apply(this, args);  
      cache.set(chave, resultado); resultado do  
      retorno;}};  
}
```

A função memoize() cria um novo objeto para usar como cache e atribui esse objeto a uma variável local para que seja privado (no fechamento da função retornada. A função retornada converte sua matriz de argumentos em uma cadeia de caracteres e usa essa cadeia de caracteres como um nome de propriedade para o objeto de cache. Se um valor existir no cache, ele o retornará diretamente. Caso contrário, ele chama a função especificada para calcular o valor desses argumentos, armazena esse valor em cache e o retorna. Aqui está como podemos usarmemoize():

Retorne o máximo divisor comum de dois inteiros usando o algoritmo euclidiano //:

```

http://en.wikipedia.org/wiki/Euclidean\_algorithmfunction
gcd(a,b) { // A verificação de tipo para a e b foi omitida

    if (a < b) {           Certifique-se de que a >= b quando
        começar
            [a, b] = [b, a];   Atribuição de desestruturação para
            variáveis swap
            }while(b !== 0)
            {
                Este é o algoritmo de Euclides para
                GCD
                [a, b] = [b,
                a%b];}retornar a;}

```

```

const gcdmemo = memoize(gcd);
gcmemo(85, 187)// => 17

```

Observe que, quando escrevemos uma função recursiva que estaremos memorizando, // normalmente queremos recusar para a versão memoizada, notthe original.

```

const factorial = memoize(function(n) {

```

```

    retorno (n <= 1) ? 1 : n * factorial(n-1);}); factorial(5)//
=> 120: também armazena em cache os valores de 4, 3, 2 e
1.

```

8.9 Resumo

Alguns pontos-chave a serem lembrados sobre este capítulo são os seguintes:

- Você pode definir funções com a palavra-chave `function` e com a sintaxe de seta ES6 `=>`. Você pode invocar funções,
- que podem ser usadas como métodos e construtores. Alguns recursos do ES6 permitem que você defina valores padrão
- para

- parâmetros de função opcionais, para reunir vários argumentos em uma matriz usando um parâmetro rest e para desestruturar argumentos de objeto e matriz em parâmetros
- de função. Você pode usar o ... spread para passar os elementos de uma matriz ou outro objeto iterável como argumentos em uma invocação de função. Uma função definida dentro e retornada por uma função enclosing, retém o acesso ao seu escopo lexical e, portanto, pode, ler e gravar as variáveis definidas dentro da função externa. As funções usadas dessa maneira são chamadas de fechamentos, e essa é uma técnica que vale a pena entender. Funções são objetos
 - que podem ser manipulados por JavaScript, e isso permite um estilo funcional de programação.

1 O termo foi cunhado por Martin Fowler.

See <http://martinfowler.com/dslCatalog/methodChaining.html>.

2 Se você estiver familiarizado com o Python, observe que isso é diferente do Python, no qual every invocation compartilha o mesmo valor padrão.

3 Isso pode não parecer um ponto particularmente interessante, a menos que você esteja familiarizado com linguagens mais estáticas, nas quais as funções fazem parte de um programa, mas não podem ser manipuladas por o programa.

Capítulo 9. Classes

Os objetos JavaScript foram abordados no Capítulo 6. Esse capítulo tratou cada objeto como um conjunto único de propriedades, diferente de todos os outros objetos. No entanto, geralmente é útil definir uma classe de objetos que compartilham certas propriedades. Os membros, ou instâncias, da classe têm suas próprias propriedades para manter ou definir seu estado, mas também têm métodos que definem seu comportamento. Esses métodos são definidos pela classe e compartilhados por todas as instâncias. Imagine uma classe chamada Complex que representa e executa aritmética em números complexos, por exemplo. Uma instância complexa teria propriedades para conter as partes reais e imaginárias (o estado) do número complexo. E a classe Complex definiria métodos para realizar adição e multiplicação (o comportamento) desses números.

Em JavaScript, as classes usam herança baseada em protótipo: se dois objetos herdam propriedades (geralmente propriedades com valor de função ou métodos) do mesmo protótipo, então dizemos que esses objetos são instâncias da mesma classe. Em poucas palavras, é assim que as classes JavaScript funcionam. Os protótipos e a herança do JavaScript foram abordados nos parágrafos 6.2.3 e 6.3.2, e você precisará estar familiarizado com o material nessas seções para entender este capítulo. Este capítulo aborda os protótipos em §9.1.

Se dois objetos herdarem do mesmo protótipo, isso normalmente (mas não necessariamente) significa que eles foram criados e inicializados pelo mesmo

função construtora ou função de fábrica. Os construtores foram abordados em §4.6, §6.2.2 e §8.2.3, e este capítulo tem mais em §9.2.

O JavaScript sempre permitiu a definição de classes. O ES6 introduziu uma sintaxe totalmente nova (incluindo uma palavra-chave de classe) que facilita ainda mais a criação de classes. Essas novas classes JavaScript funcionam da mesma forma que as classes antigas, e este capítulo começa explicando a maneira antiga de criar classes, porque isso demonstra mais claramente o que está acontecendo nos bastidores para fazer as classes funcionarem. Depois de explicarmos esses fundamentos, mudaremos e começaremos a usar a nova sintaxe simplificada de definição de classe.

Se você estiver familiarizado com linguagens de programação orientadas a objetos fortemente tipadas, como Java ou C++, notará que as classes JavaScript são bem diferentes das classes nessas linguagens. Existem algumas semelhanças sintáticas, e você pode emular muitos recursos de classes "clássicas" em JavaScript, mas é melhor entender de antemão que as classes do JavaScript e o mecanismo de herança baseado em protótipo são substancialmente diferentes das classes e do mecanismo de herança baseado em classe do Java e linguagens semelhantes.

9.1 Classes e Protótipos

Em JavaScript, uma classe é um conjunto de objetos que herdam propriedades do mesmo objeto protótipo. O objeto protótipo, portanto, é a característica central de uma classe. O Capítulo 6 abordou a função `Object.create()` que retorna um objeto recém-criado que herda de um objeto protótipo especificado. Se definirmos um objeto protótipo e depois usar `Object.create()` para criar objetos que herdam dele, temos

definiu uma classe JavaScript. Normalmente, as instâncias de uma classe requerem inicialização adicional e é comum definir uma função que cria e inicializa o novo objeto. O exemplo 9-1 demonstra isso: ele define um objeto protótipo para uma classe que representa um intervalo de valores e também define uma função de fábrica que cria e inicializa uma nova instância da classe.

Exemplo 9-1. Uma classe JavaScript simples

Esta é uma função de fábrica que retorna um novo intervalo object.function range(from, to) {

Use Object.create() para criar um objeto que herda do

prototype definido abaixo. O objeto protótipo é armazenado como uma propriedade dessa função e define os métodos compartilhados (comportamento) para todos os objetos de intervalo.let r = Object.create(intervalo.métodos);

Armazene os pontos inicial e final (estado) desse novo rangeobject.

Essas são propriedades não herdadas que são exclusivas desse objeto.

```
r.from = de;  
r.to = para;
```

*Por fim, retorne o novo
objeto return r;}*

Este objeto protótipo define métodos herdados por todos rangeobjects.range.methods = {

*Retorne true se x estiver no intervalo, false caso contrário//
Este método funciona para intervalos textuais e de datas, bem como numéricos.*

```
includes(x) { return this.from <= x && x <= this.to; },
```

Uma função geradora que torna as instâncias do classiterável.

Observe que ele só funciona para intervalos numéricos.

```

*[Symbol.iterator]() {
    for(let x = Math.ceil(this.from); x <= this.to; x++)
rendimento x;
},

Retorna uma representação de string do rangeToString() {
return "(" + this.from + "... " + this.to +")"; }};

```

Aqui estão exemplos de usos de um objeto de intervalo. let r = intervalo(1,3); // Crie um objeto de intervalo. includes(2) // => true: 2 está no ranger.toString() // => "(1...3)[... r]" // => [1, 2, 3]; Converter em um Iterador ArrayVia

Há algumas coisas que vale a pena observar no código do Exemplo 9-1:

- Este código define uma função de fábrica range() para criar novos objetos Range. Ele usa a propriedade methods desta função range() como um local conveniente para armazenar o objeto protótipo que define a classe. Não há nada de especial ou idiomático em colocar o objeto protótipo aqui. A função range() define as propriedades from e to em cada objeto Range. Essas são as propriedades não compartilhadas e não herdadas que definem o estado exclusivo de cada objeto individualRange. O objeto range.methods usa a sintaxe abreviada ES6 para definir métodos, e é por isso que você não vê a palavra-chave function em nenhum lugar. (Consulte §6.10.5 para revisar a sintaxe do método abreviado objectliteral.) Um dos métodos no protótipo tem o nome computado (§6.10.2) Symbol.iterator, o que significa que é
-

definindo um iterador para objetos Range. O nome deste método é prefixado com `*`, o que indica que é uma função geradora em vez de uma função regular. Iteradores e geradores são abordados em detalhes no Capítulo 12. Por enquanto, o resultado é que as instâncias dessa classe Range podem ser usadas com o loop `for/of` e com o `... operador de propagação`.

- Os métodos compartilhados e herdados definidos em `range.methods` usam todas as propriedades `from` e `to` que foram inicializadas na função de fábrica `range()`. Para se referir a eles, eles usam a palavra-chave `this` para se referir ao objeto por meio do qual foram invocados. Esse uso disso é uma característica fundamental dos métodos de qualquer classe.

9.2 Classes e Construtores

O exemplo 9-1 demonstra uma maneira simples de definir uma classe JavaScript. Não é a maneira idiomática de fazer isso, no entanto, porque não definiu um construtor. Um construtor é uma função projetada para a inicialização de objetos recém-criados. Os construtores são invocados usando a `new` keyword, conforme descrito em §8.2.3. As invocações do construtor usando `new` criam automaticamente o novo objeto, portanto, o próprio construtor só precisa inicializar o estado desse novo objeto. A característica crítica das invocações do construtor é que a propriedade `prototype` do construtor é usada como o protótipo do novo objeto.

§6.2.3 introduziu protótipos e enfatizou que, embora quase todos os objetos tenham um protótipo, apenas alguns objetos têm uma propriedade de protótipo. Finalmente, podemos esclarecer isso: são objetos de função que possuem uma propriedade de protótipo. Isso significa que todos os objetos criados com a mesma função construtora herdam do mesmo objeto e são

portanto, membros da mesma classe. O Exemplo 9-2 mostra como poderíamos alterar a classe Range do Exemplo 9-1 para usar uma função construtora em vez de uma função de fábrica. O exemplo 9-2 demonstra a maneira idiomática de criar uma classe em versões do JavaScript que não suportam a palavra-chave de classe ES6. Mesmo que a classe seja bem suportada agora, ainda há muito código JavaScript mais antigo por aí que define classes como esta, e você deve estar familiarizado com o idioma para que possa ler o código antigo e para entender o que está acontecendo "nos bastidores" ao usar a palavra-chave class.

Exemplo 9-2. Uma classe Range usando um construtor

Esta é uma função construtora que inicializa novos objetos Range.// Observe que ela não cria ou retorna o objeto. Ele apenas inicializa this.function Range(from, to) {

Armazene os pontos inicial e final (estado) desse novo rangeobject.

Essas são propriedades não herdadas que são exclusivas desse objeto.

```
this.from = de;  
this.to = para;}
```

Todos os objetos Range herdam deste objeto.// Observe que o nome da propriedade deve ser "prototype" para que isso funcione. Gama.prototype = {

Retorne true se x estiver no intervalo, false caso contrário// Este método funciona para intervalos textuais e de datas, bem como numéricos.

```
includes: function(x) { return this.from <= x && x  
<=this.to; },
```

Uma função geradora que torna as instâncias do classiterável.

Observe que ele só funciona para intervalos numéricos. [Symbol.iterator]: função() {*

```
        for(let x = Math.ceil(this.from); x <= this.to; x++)
rendimento x;
},
```

Retorna uma representação de string do rangeToString:
`function() { return "(" + this.from + "..." +this.to +
")"; };`

Aqui estão exemplos de usos deste novo classlet Range r = new Range(1,3); // Crie um objeto Range; observe o uso de newr.includes(2) // => true: 2 está no ranger.toString() // => "(1...3)"[... r] // => [1, 2, 3]; Converter em um Iterador ArrayVia

Vale a pena comparar os exemplos 9-1 e 9-2 com bastante cuidado e observar as diferenças entre essas duas técnicas para definir classes. Primeiro, observe que renomeamos a função de fábrica range() para Range() quando a convertemos em um construtor. Esta é uma convenção de codificação muito comum: as funções do construtor definem, em certo sentido, classes e as classes têm nomes que (por convenção) começam com letras maiúsculas. Funções e métodos regulares têm nomes que começam com letras minúsculas.

Em seguida, observe que o construtor Range() é invocado (no final do exemplo) com a palavra-chave new, enquanto a função factoryrange () foi invocada sem ela. O Exemplo 9-1 usa function invocation regular (§8.2.1) para criar o novo objeto, e o Exemplo 9-2 usa a invocação do construtor (§8.2.3). Como o construtor Range() é invocado com new, ele não precisa chamar Object.create() ou executar qualquer ação para criar um novo objeto. O novo objeto é criado automaticamente antes que o construtor seja chamado e pode ser acessado como this

valor. O construtor Range() só precisa inicializar isso. Os construtores nem precisam retornar o objeto recém-criado. A invocação do construtor cria automaticamente um novo objeto, invoca o construtor como um método desse objeto e retorna o novo objeto. O fato de que a invocação do construtor é tão diferente da invocação de função regular é outra razão pela qual damos aos construtores nomes que começam com letras maiúsculas. Os construtores são escritos para serem invocados como construtores, com a palavra-chave new, e geralmente não funcionarão corretamente se forem invocados como funções regulares. Uma convenção de nomenclatura que mantém as funções de construtor distintas das funções regulares ajuda os programadores a saber quando usar new.

CONSTRUTORES E NOVOS. ALVO

Dentro de um corpo de função, você pode dizer se a função foi invocada como um construtor com a expressão especial new.target. Se o valor dessa expressão for definido, você saberá que a função foi invocada como um construtor, com a nova palavra-chave. Quando discutirmos subclasses em §9.5, veremos que new.target nem sempre é uma referência ao construtor em que é usado: também pode se referir à função construtora de uma subclasse.

Se new.target for indefinido, a função que contém foi invocada como uma função, sem a nova palavra-chave. Os vários construtores de erro do JavaScript podem ser invocados sem new, e se você quiser emular esse recurso em seus próprios construtores, você pode escrevê-los assim:

```
função C() {  
  if (!new.target) return new C()//  
  o código de inicialização vai aqui}
```

Essa técnica só funciona para construtores definidos dessa maneira antiquada. As classes criadas com a palavra-chave theclass não permitem que seus construtores sejam invocados sem new.

Outra diferença crítica entre os Exemplos 9-1 e 9-2 é a forma como o objeto protótipo é nomeado. No primeiro exemplo, o protótipo wasrange.methods. Este era um nome conveniente e descritivo, mas

arbitrário. No segundo exemplo, o protótipo `isRange.prototype`, e esse nome é obrigatório. Uma invocação do construtor `Range()` usa automaticamente `Range.prototype` como o protótipo do novo objeto `Range`.

Por fim, observe também as coisas que não mudam entre os Exemplos 9-1 e 9-2: os métodos de intervalo são definidos e invocados da mesma maneira para ambas as classes. Como o Exemplo 9-2 demonstra a maneira idiomática de criar classes em versões do JavaScript anteriores ao ES6, ele não usa a sintaxe do método abreviado ES6 no objeto protótipo e explicita os métodos com a palavra-chave `function`. Mas você pode ver que a implementação dos métodos é a mesma em ambos os exemplos.

É importante ressaltar que nenhum dos dois exemplos de intervalo usa arrowfunctions ao definir construtores ou métodos. Lembre-se de §8.1.3 que as funções definidas dessa maneira não têm uma propriedade de protótipo e, portanto, não podem ser usadas como construtores. Além disso, as funções de seta herdam a palavra-chave `this` do contexto em que são definidas, em vez de defini-la com base no objeto por meio do qual são invocadas, e isso as torna inúteis para métodos porque a característica definidora dos métodos é que eles usam isso para se referir à instância em que foram invocados.

Felizmente, a nova sintaxe da classe ES6 não permite a opção de definindo métodos com funções de seta, então isso não é um erro que você pode cometer accidentalmente ao usar essa sintaxe. Abordaremos a palavra-chave `ES6class` em breve, mas primeiro, há mais detalhes a serem abordados sobre os construtores.

9.2.1 Construtores, Identidade de Classe e instanceof

Como vimos, o objeto protótipo é fundamental para a identidade de uma classe: dois objetos são instâncias da mesma classe se e somente se herdarem do mesmo objeto protótipo. A função construtora que inicializa o estado de um novo objeto não é fundamental: duas funções construtoras podem ter propriedades de protótipo que apontam para o mesmo objeto protótipo. Em seguida, ambos os construtores podem ser usados para criar instâncias da mesma classe.

Mesmo que os construtores não sejam tão fundamentais quanto os protótipos, o construtor serve como a face pública de uma classe. Mais obviamente, o nome da função construtora é geralmente adotado como o nome da classe. Dizemos, por exemplo, que o construtor Range() cria objetos Range. Mais fundamentalmente, no entanto, os construtores são usados como o operando à direita do operador instanceof ao testar objetos para associação em uma classe. Se tivermos um objeto r e quisermos saber se é um objeto Range, podemos escrever:

```
r instanceof Intervalo => true: r herda de  
Gama.prototype
```

O operador instanceof foi descrito em §4.9.4. O lefthand operand deve ser o objeto que está sendo testado e o righthand operand deve ser uma função construtora que nomeia uma classe. A expressão o instânciade C é avaliada como verdadeira se o herda de C.prototype. A herança não precisa ser direta: se o herda de um objeto que herda de um objeto que herda de C.prototype, a expressão ainda será avaliada como verdadeira.

Tecnicamente falando, no exemplo de código anterior, o operador `instanceof` não está verificando se `r` foi realmente inicializado pelo construtor `Range`. Em vez disso, ele está verificando se `r` herda de `Range.prototype`. Se definirmos uma `functionStrange()` e definirmos seu protótipo como o mesmo que `Range.prototype`, então os objetos criados com `new Strange()` contarão como objetos `Range` no que diz respeito a `instanceof` (eles não funcionarão como objetos `Range`, no entanto, porque suas propriedades `from` e `to` não foram inicializadas):

```
function Estranho() {}Estranho.prototype =  
Interval.prototype; new Strange()  
instanceof Range// => true
```

Mesmo que `instanceof` não possa realmente verificar o uso de um construtor, ele ainda usa uma função de construtor como seu lado direito porque os construtores são a identidade pública de uma classe.

Se você quiser testar a cadeia de protótipos de um objeto para um protótipo específico e não quiser usar a função de construtor como intermediário, poderá usar o método `isPrototypeOf()`. No Exemplo 9-1, por exemplo, definimos uma classe sem uma função construtora, portanto, não há como usar `instanceof` com essa classe. Em vez disso, no entanto, poderíamos testar se um objeto `r` era um membro dessa classe sem construtor com este código:

```
range.methods.isPrototypeOf(r); range.methods é o  
objeto protótipo.
```

9.2.2 A propriedade do construtor

No Exemplo 9-2, definimos Range.prototype como um novo objeto que continha os métodos para nossa classe. Embora fosse conveniente expressar esses métodos como propriedades de um único objeto literal, não era realmente necessário criar um novo objeto. Qualquer JavaScriptfunction regular (excluindo funções de seta, funções geradoras e asyncfunctions) pode ser usada como um construtor e invocações de construtor precisam de uma propriedade prototype. Portanto¹, cada JavaScriptfunction regular tem automaticamente uma propriedade prototype. O valor de thisproperty é um objeto que tem uma única propriedade de construtor não enumerável. O valor da propriedade do construtor é o functionobject:

```
let F = função() {}; Este é um objeto de função.  
let p = F.prototype; // Este é o objeto prototype associado a  
F.  
let c = p.constructor; Esta é a função associada ao  
protótipo.  
c === F // => true: F.prototype.constructor === F  
para qualquer F
```

A existência desse objeto protótipo predefinido com sua propriedade constructor significa que os objetos normalmente herdam uma propriedade constructor que se refere ao seu construtor. Como os construtores servem como a identidade pública de uma classe, essa propriedade do construtor fornece a classe de um objeto:

```
let o = new F(); Crie um objeto o da classe F  
o.construtor === F => true: a propriedade do construtor  
especifica a classe
```

A Figura 9-1 ilustra essa relação entre a função construtora, seu objeto protótipo, a referência anterior do protótipo para o

construtor e as instâncias criadas com o construtor.

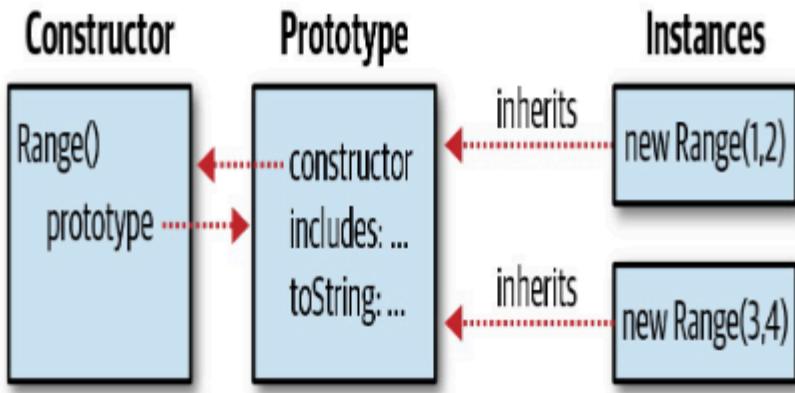


Figura 9-1. Uma função de construtor, seu protótipo e instâncias

Observe que a Figura 9-1 usa nosso construtor Range() como exemplo. Na verdade, no entanto, a classe Range definida no Exemplo 9-2 substitui o objeto Range.prototype predefinido por um objeto próprio. E o novo objeto protótipo que ele define não tem uma propriedade de construtor. Portanto, as instâncias da classe Range, conforme definidas, não têm uma propriedade de construtor. Podemos remediar esse problema adicionando explicitamente um construtor ao protótipo:

```
Intervalo.protótipo =  
{construtor: Intervalo, Definir explicitamente o construtor  
referência anterior  
  
/* definições de método vão aqui  
*/};
```

Outra técnica comum que você provavelmente verá no código JavaScript mais antigo é usar o objeto prototype predefinido com sua propriedade constructor e adicionar métodos a ele um de cada vez com o código

Assim:

```
Estenda o objeto Range.prototype predefinido para que não
sobrescrevamos// o
Range.prototype.constructorproperty.Range.prototype.includ
es = function(x) {

    return this.from <= x && x <= this.to;};
Range.prototype.toString = function() {

    return "(" + this.from + "..." + this.to + ")";};
```

9.3 Classes com a palavra-chave class

As classes fazem parte do JavaScript desde a primeira versão da linguagem, mas no ES6, elas finalmente obtiveram sua própria sintaxe com a introdução da palavra-chave `class`. O exemplo 9-3 mostra a aparência da classe `ourRange` quando escrita com essa nova sintaxe.

Exemplo 9-3. A classe Range reescrita usando a classe

```
class Intervalo {
    construtor(de, para) {
        Armazene os pontos inicial e final (estado) deste novo
        range do Range.
        Essas são propriedades não herdadas que são exclusivas de
        este objeto.
        this.from = de;
        this.to = para;}
```

*Retorne true se x estiver no intervalo, false caso contrário//
Este método funciona para intervalos textuais e de datas, bem
como numéricos.*

```
    includes(x) { return this.from <= x && x <= this.to; }
```

*Uma função geradora que torna as instâncias do
classiterável.*

```

Observe que ele só funciona para intervalos
numéricos.*[Symbol.iterator]() {
    for(let x = Math.ceil(this.from); x <= this.to; x++)
rendimento x;
}

Retorna uma representação de string do rangeToString()
{ return '${this.from}...${this.to}'; }

```

Aqui estão exemplos de usos desta nova classe Range r = new Range(1,3); // Crie um objeto Range.r.includes(2)// => true: 2 está no ranger.toString()// => "(1...3)[... r]// => [1, 2, 3]; Converter em um Iterador ArrayVia

É importante entender que as classes definidas nos Exemplos 9-2 e 9-3 funcionam exatamente da mesma maneira. A introdução da palavra-chave class na linguagem não altera a natureza fundamental das classes baseadas em protótipos do JavaScript. E embora o Exemplo 9-3 use a palavra-chave class, o objeto Range resultante é uma função construtora, assim como a versão definida no Exemplo 9-2. A nova sintaxe de classe é limpa e conveniente, mas é melhor pensada como "açúcar sintático" para o mecanismo de definição de classe mais fundamental mostrado no Exemplo 9-2.

Observe as seguintes coisas sobre a sintaxe da classe no Exemplo 9-3:

- A classe é declarada com a palavra-chave class, que é seguida pelo nome da classe e um corpo de classe entre chaves. O corpo da classe inclui definições de método que usam a abreviação do método objectliteral (que também usamos no Exemplo 9-1), em que a palavra-chave function é omitida. Ao contrário de objectliterals, no entanto, nenhuma vírgula é usada para separar os métodos

um do outro. (Embora os corpos de classe sejam superficialmente semelhantes aos literais de objeto, eles não são a mesma coisa. Em particular, eles não suportam a definição de propriedades com pares nome/valor.) O construtor de palavra-chave é usado para definir a função construtor para a classe. A função definida não é realmente chamada de "construtor", no entanto. A declaração de classe define uma nova variável Range e atribui o valor dessa função de construtor especial a essa variável. Se sua classe não precisar fazer nenhuma inicialização, você fará o canon da palavra-chave constructor e seu corpo, e uma função emptyconstructor será criada implicitamente para você. Se você quiser definir uma classe que subclasse ou herda de outra classe, você pode usar a palavra-chave extends com a palavra-chave class:

Um Span é como um Range, mas em vez de inicializá-lo com // um início e um fim, nós o inicializamos com um início e um length

```
class Span extends Range {
```

```
    construtor(início, comprimento) {
        if (comprimento >= 0) {
            super(início, início +
                duração);} else {
            super(início + duração, início);}}}
```

A criação de subclasses é um tópico por si só. Voltaremos a ele e explicaremos as extensões e super palavras-chave mostradas aqui, em §9.5.

Assim como as declarações de função, as declarações de classe têm formas de instrução e expressão. Assim como podemos escrever:

```
let quadrado = função(x) { return x * x;
}; quadrado(3)// => 9
```

Também podemos escrever:

```
let Quadrado = class { construtor(x) { this.area = x * x; }
}; new Quadrado(3).area// => 9
```

Assim como acontece com as expressões de definição de função, as expressões de definição de classe podem incluir um nome de classe opcional. Se você fornecer esse nome, thatname só será definido dentro do próprio corpo da classe.

Embora as expressões de função sejam bastante comuns (particularmente com a abreviação da função de seta), na programação JavaScript, as expressões de definição de classe não são algo que você provavelmente usará muito, a menos que você se encontre escrevendo uma função que usa uma classe como argumento e retorna uma subclasse.

Concluiremos esta introdução à palavra-chave `class` mencionando algumas coisas importantes que você deve saber que não são aparentes na sintaxe da classe:

- Todo o código dentro do corpo de uma declaração de classe está implicitamente no modo estrito (§5.6.3), mesmo que nenhuma diretiva "use strict" apareça. Isso significa, por exemplo, que você não pode usar literais octalinteiros ou a instrução `with` dentro de corpos de classe e que é mais provável que você obtenha erros de sintaxe se esquecer de declarar uma variável antes de usá-la.

- Ao contrário das declarações de função, as declarações de classe não são "içadas". Lembre-se do §8.1.1 que as definições de função se comportam como se tivessem sido movidas para o topo do arquivo delimitador ou da função delimitadora, o que significa que você pode invocar uma codificação de função que vem antes da definição real da função. Embora as declarações de classe sejam como declarações de função em alguns aspectos, elas não compartilham esse comportamento de içamento: você não pode instanciar uma classe antes de declará-la.

9.3.1 Métodos estáticos

Você pode definir um método estático dentro de um corpo de classe prefixando a declaração methodmethod com a palavra-chave static. Os métodos estáticos são definidos como propriedades da função construtora e não como propriedades do objeto protótipo.

Por exemplo, suponha que adicionamos o seguinte código ao Exemplo 9-3:

```
análise estática (s) {
  let corresponde = s.match(/^((\d+)\.\.\.(\d+))$/);
  if (!corresponde) {
    throw new TypeError('Não é possível analisar Range from "${s}".');
  }
  return new Range(parseInt(matches[1]), parseInt(matches[2]));
}
```

O método definido por este código é Range.parse(), notRange.prototype.parse(), e você deve invocá-lo por meio do construtor, não por meio de uma instância:

```
let r = Range.parse('(1...10)');
Retorna um novo objeto Range
```

```
r.parse('(1...10)');           TypeError: r.parse não é  
uma função
```

Às vezes, você verá métodos estáticos chamados de métodos de classe porque eles são invocados usando o nome da classe/construtor. Quando esse termo é usado, é para contrastar os métodos de classe com os métodos de instância regulares que são invocados em instâncias da classe. Como os métodos estáticos são invocados no construtor em vez de em qualquer instância específica, quase nunca faz sentido usar a palavra-chave `this` em um método estático.

Veremos exemplos de métodos estáticos no Exemplo 9-4.

9.3.2 Getters, Setters e outros Formulários de Método

Dentro de um corpo de classe, você pode definir métodos `getter` e `setter` (§6.10.6) da mesma forma que em literais de objeto. A única diferença é que em corpos de classe, você não coloca uma vírgula após o `getter` ou `setter`. O exemplo 9-4 inclui um exemplo prático de um método `getter` em uma classe.

Em geral, todas as sintaxes de definição de método abreviado permitidas em literais de objeto também são permitidas em corpos de classe. Isso inclui métodos geradores (marcados com `*`) e métodos cujos nomes são o valor de uma expressão entre colchetes. Na verdade, você já viu (no Exemplo 9-3) um método gerador com um nome calculado que torna a classe `Range` iterável:

```
*[Symbol.iterator]() {  
  for(let x = Math.ceil(this.from); x <= this.to;  
    x++) yield x;}
```

9.3.3 Campos Público, Privado e Estático

Na discussão aqui de classes definidas com a palavra-chave `class`, descrevemos apenas a definição de métodos dentro do corpo da classe. O padrão ES6 permite apenas a criação de métodos (incluindo getters, setters e geradores) e métodos estáticos; não inclui syntaxe para definir campos. Se você quiser definir um campo (que é apenas um sinônimo orientado a objetos para "propriedade") em uma instância de classe, você deve fazer isso na função construtora ou em um dos métodos. E se você quiser definir um campo estático para uma classe, você deve fazer isso fora do corpo da classe, depois que a classe tiver sido definida. O exemplo 9-4 inclui exemplos de ambos os tipos de campos.

A padronização está em andamento, no entanto, para a syntaxe de classe estendida que permite a definição de campos de instância e estáticos, tanto em formas públicas quanto privadas. O código mostrado no restante desta seção ainda não é JavaScript padrão no início de 2020, mas já é suportado no Chrome e parcialmente suportado (somente campos de instância pública) no Firefox. A syntaxe para campos de instância pública é de uso comum por programadores JavaScript que usam a estrutura React e o transpilador Babel.

Suponha que você esteja escrevendo uma classe como esta, com um construtor que inicializa três campos:

```
classe Buffer {  
    construtor() {  
        this.size = 0; this.capacity = 4096; this.buffer =  
        new Uint8Array(this.capacity);}}
```

Com a nova sintaxe de campo de instância que provavelmente será padronizada, você pode escrever:

```
classe Buffer {  
    tamanho = 0; capacidade = 4096; buffer =  
    new Uint8Array(this.capacity);}
```

O código de inicialização do campo saiu do construtor e agora aparece diretamente no corpo da classe. (Esse código ainda é executado como parte do construtor, é claro. Se você não definir um construtor, os campos serão inicializados como parte do construtor criado implicitamente.) Os prefixos `this.que` que aparecem no lado esquerdo das atribuições desapareceram, mas observe que você ainda deve usar `this.` para se referir a esses campos, mesmo no lado direito das atribuições do inicializador. A vantagem de inicializar seus campos de instância dessa maneira é que essa sintaxe permite (mas não exige) que você coloque os inicializadores no topo da definição da classe, deixando claro para os leitores exatamente quais campos manterão o estado de cada instância. Você pode declarar campos sem um inicializador apenas escrevendo o nome do campo seguido por um ponto e vírgula. Se você fizer isso, o valor inicial do campo será indefinido. É melhor estilosempre tornar o valor inicial explícito para todos os campos da sua classe.

Antes da adição dessa sintaxe de campo, os corpos de classe se pareciam muito com literais de objeto usando a sintaxe do método de atalho, exceto que as vírgulas haviam sido removidas. Essa sintaxe de campo — com sinais de igual e ponto e vírgula em vez de dois-pontos e vírgulas — deixa claro que os corpos de classe não são iguais aos literais de objeto.

A mesma proposta que busca padronizar esses campos de instância também define campos de instância privada. Se você usar a sintaxe de inicialização de campo de instância mostrada no exemplo anterior para definir um campo cujo nome começa com # (que normalmente não é um caractere legal em identificadores JavaScript), esse campo será utilizável (com o prefixo #) dentro do corpo da classe, mas será invisível e inacessível (e, portanto, imutável) para qualquer código fora do corpo da classe. Se, para a classe Buffer hipotética anterior, você quiser garantir que os usuários da classe não possam modificar inadvertidamente o campo de tamanho de uma instância, você poderá usar um campo de #size privado e, em seguida, definir uma função getter para fornecer acesso somente leitura ao valor:

```
classe Buffer {  
  #size = 0; get size() { return  
    this.#size; }}
```

Observe que os campos privados devem ser declarados usando essa nova sintaxe de campo antes de serem usados. Você não pode simplesmente escrever isso.`#size = 0;` no construtor de uma classe, a menos que você inclua uma "declaração" do campo diretamente no corpo da classe.

Finalmente, uma proposta relacionada busca padronizar o uso da palavra-chave estática para campos. Se você adicionar static antes de uma declaração de campo pública ou privada, esses campos serão criados como propriedades da função construtor em vez de propriedades de instâncias. Considere o método `staticRange.parse()` que definimos. Ele incluía uma expressão regular bastante complexa que pode ser boa para fatorar em seu próprio campo estático. Com a nova sintaxe de campo estático proposta, poderíamos fazer isso como

este:

```
integerRangePattern estático = /^((\d+)\.\.\.(\d+))$/; análise estática (s) {
    let corresponde = s.match(Range.integerRangePattern);
    if (!corresponde) {
        throw new TypeError('Não é possível analisar Range from
"${s}"')
    }return new Range(parseInt(matches[1]), matches[2]);}
```

Se quiséssemos que esse campo estático fosse acessível apenas dentro da classe, poderíamos torná-lo privado usando um nome como #pattern.

9.3.4 Exemplo: uma classe de número complexo

O exemplo 9-4 define uma classe para representar números complexos. A classe é relativamente simples, mas inclui métodos de instância (incluindo getters), métodos estáticos, campos de instância e campos estáticos. Ele inclui algum código comentado demonstrando como podemos usar a sintaxe ainda não padrão para definir campos de instância e campos estáticos dentro do corpo da classe.

Exemplo 9-4. Complex.js: uma classe de números complexos

```
/**
 * Instâncias desta classe complexa representam números
complexos.* Lembre-se de que um número complexo é a soma de um
número real e
* número imaginário e que o número imaginário i é a raiz
quadrada de -1.
*/class
Complexo {
Depois que as declarações de campo de classe são padronizadas,
poderíamos declarar
    campos privados para conter as partes reais e imaginárias de um
```

```
número complexo  
aqui, com código como  
this:// // #r = 0; // #i = 0;
```

Esta função construtora define os campos de instância rand i em cada instância que ele cria. Esses campos contêm as partes reais e imaginárias de o número complexo: eles são o estado do objeto.

```
construtor(real, imaginário) {  
    this.r = real;      Este campo contém a parte real  
    do número.  
    this.i = imaginário; Este campo contém o imaginário  
parte.  
}
```

Aqui estão dois métodos de instância para adição e multiplicação de números complexos. Se c e d são instâncias desta classe, nós pode escrever c.plus(d) ou d.times(c)plus(that) {

```
return new Complex(this.r + that.r, this.i +  
that.i); }times(that) {  
  
    return new Complex(this.r * that.r - this.i * that.i,  
                      this.r * that.i + this.i * that.r);  
}
```

E aqui estão as variantes estáticas dos métodos aritméticos complexos.

```
Poderíamos escrever Complex.sum(c,d) e  
Complex.product(c,d) static sum(c, d) { return c.plus(d);  
}produto estático(c, d) { return c.times(d); }
```

Estes são alguns métodos de instância que são definidos como getters então eles são usados como campos. Os getters reais e imaginários ser útil se estivéssemos usando campos privados this.#r e this.#i

```
get real() { return this.r; }
```

```
get imaginary() { return this.i; }get magnitude() {  
    return Math.hypot(this.r, this.i); }
```

*As classes quase sempre devem ter um `toString()`
`methodToString()` { return "{\$this.r},{\$this.i}"; }*

Muitas vezes é útil definir um método para testar se

*Duas instâncias de sua classe representam o mesmo
`valueequals(that)` {
 retornar essa instância de Complex &&
 this.r === that.r
&&this.i === that.i;
}*

*Uma vez que os campos estáticos são suportados dentro de
corpos de classe, podemos
definir uma constante Complex.ZERO útil como `this://`
`ZERO estático = new Complex(0,0);}`*

*Aqui estão alguns campos de classe que contêm números
complexos predefinidos úteis. Complexo.ZERO = novo
Complexo(0,0); Complexo.ONE = novo Complexo(1,0);
Complexo.I = novo Complexo(0,1);*

Com a classe Complex do Exemplo 9-4 definida, podemos usar o construtor, campos de instância, métodos de instância, campos de classe e classmethods com código como este:

```
let c = novo Complexo(2, 3); Crie um novo objeto com  
o construtor d = novo Complexo(c.i, c.r); Use os campos de  
instância de cc.plus(d).toString()// => "{5,5}"; use  
instancemethodsc.magnitude// => Math.hypot(2,3); use  
agetter functionComplex.product(c, d)// => new Complex(0,  
13); método estáticoComplexo.ZERO.toString()// => "{0,0}";  
uma propriedade estática
```

9.4 Adicionando métodos a classes existentes

O mecanismo de herança baseado em protótipo do JavaScript é dinâmico: um objeto herda propriedades de seu protótipo, mesmo que as propriedades do protótipo mudem após a criação do objeto. Isso significa que aumentamos as classes JavaScript simplesmente adicionando novos métodos aos seus objetos prototype.

Aqui, por exemplo, está o código que adiciona um método para calcular o conjugado complexo à classe Complex do Exemplo 9-4:

Retorne um número complexo que é o conjugado complexo deste. Complexo.protótipo.conj = function() { return novoComplexo(this.r, -this.i); };

O objeto protótipo das classes JavaScript integradas também é aberto assim, o que significa que podemos adicionar métodos a números, strings, matrizes,funções e assim por diante. Isso é útil para implementar novos recursos de linguagem em versões mais antigas da linguagem:

*Se o novo método String.startsWith() ainda não estiver definido... se (! String.prototype.startsWith) {
// ... em seguida, defina-o assim usando o método indexOf() mais antigo.
String.prototype.startsWith = função(ões) {
return this.indexOf(s) === 0;};}*

Aqui está outro exemplo:

Invoque a função f isso várias vezes, passando o número de iteração// Por exemplo, para imprimir "olá" 3 times://let n = 3;//n.times(i => { console.log('olá \${i}'); }); Número.protótipo.vezes = função(f, contexto) {

```
let n = this.valueOf(); for(let i = 0; i < n; i++) f.call(contexto, i);
```

Adicionar métodos aos protótipos de tipos internos como esse é geralmente considerado uma má ideia porque causará confusão e problemas de compatibilidade no futuro se uma nova versão do JavaScript definir um método com o mesmo nome. É até possível adicionar métodos a Object.prototype, disponibilizando-os para todos os objetos. Mas isso nunca é uma boa coisa a se fazer porque as propriedades adicionadas a Object.prototype são visíveis para loops for/in (embora você possa evitar isso usando Object.defineProperty() [§14.1] para tornar a nova propriedade não enumerável).

9.5 Subclasses

Na programação orientada a objetos, uma classe B pode estender ou subclasse outra classe A. Dizemos que A é a superclasse e B é a subclasse. As instâncias de B herdam os métodos de A. A classe B pode definir seus próprios métodos, alguns dos quais podem substituir métodos do mesmo nome definidos pela classe A. Se um método de B substitui um método de A, o método de substituição em B geralmente precisa invocar o método substituído em A. Da mesma forma, o construtor de subclasse B() normalmente deve invocar o construtor de superclasse A() para garantir que as instâncias sejam completamente inicializadas.

Esta seção começa mostrando como definir subclasses da maneira antiga, pré-ES6 e, em seguida, passa rapidamente para demonstrar a subclass usando a classe e estende as palavras-chave e a invocação do método construtor de superclasse com a palavra-chave super. A seguir está uma subseção sobre como evitar subclasses e confiar na composição de objetos em vez de herança. A seção termina com um exemplo estendido que define uma hierarquia de classes Set e demonstra como classes abstratas podem ser usadas para separar a interface da implementação.

9.5.1 Subclasses e protótipos

Suponha que quiséssemos definir uma subclass Span da classe Range from Example 9-2. Essa subclass funcionará como um Range, mas em vez de inicializá-la com um início e um fim, especificaremos um início e uma distância ou intervalo. Uma instância dessa classe Span também é uma instância da superclasse Range. Uma instância de span herda um método customizedToString() de Span.prototype, mas para ser uma subclass de Range, ela também deve herdar métodos (como includes()) de Range.prototype.

Exemplo 9-5. Span.js: uma subclass simples de Range

Esta é a função construtora para nossa

subclasse função Span(start, span) {

```
if (intervalo >= 0) {  
    this.from = início;  
    this.to = início +  
        intervalo;} else {  
this.to = início; this.from =  
    início + intervalo;}}
```

```
Certifique-se de que o protótipo Span herde do Range.prototype
Range.prototypeSpan.prototype =
Object.create(Range.prototype);
```

```
Não queremos herdar Range.prototype.constructor, então nós // definimos nossa própria propriedade de construtor.
Span.prototype.constructor = Span;
```

```
Ao definir seu próprio método toString(), o Span substitui o método // toString() que, de outra forma, herdaria fromRange.Span.prototype.toString = function() {

return '(${this.from}... +${this.to - this.from})';};
```

Para tornar Span uma subclasse de Range, precisamos organizar forSpan.prototype para herdar de Range.prototype. A linha de código no exemplo anterior é esta e, se fizer sentido para você, você entenderá como as subclasses funcionam em JavaScript:

```
Span.prototype = Object.create(Range.prototype);
```

Os objetos criados com o construtor Span() herdarão do objeto theSpan.prototype. Mas criamos esse objeto para herdar fromRange.prototype, então os objetos Span herdarão de bothSpan.prototype e Range.prototype.

Você pode notar que nosso construtor Span() define as mesmas propriedades from e to que o construtor Range() faz e, portanto, não precisa invocar o construtor Range() para inicializar o novo objeto. Da mesma forma, o método toString() do Span reimplementa completamente a conversão de string sem a necessidade de chamar a versão de Range de toString(). Isso torna o Span um caso especial, e só podemos realmente nos safar com esse tipo de subclasse porque conhecemos o

detalhes de implementação da superclasse. Um mecanismo de subclasse robusto precisa permitir que as classes invoquem os métodos e o construtor de sua superclasse, mas antes do ES6, o JavaScript não tinha uma maneira simples de fazer essas coisas.

Felizmente, o ES6 resolve esses problemas com a palavra-chave `super` como parte da sintaxe da classe. A próxima seção demonstra como funciona.

9.5.2 Subclasses com extensões e `super`

No ES6 e posterior, você pode criar uma superclasse simplesmente adicionando uma cláusula `extends` a uma declaração de classe, e você pode fazer isso mesmo para classes integradas:

Uma subclasse Array trivial que adiciona getters para o primeiro e o último elements.

```
class EZArray extends Array {  
  get first() { return this[0]; }  
  get last() {  
    return this[this.length-1]; }  
}
```

```
let a = new EZArray();  
a instanceof EZArray // => true: a é  
// subclass instance  
a instanceof Array // => true: a também é  
// uma superclasse  
a.push(1,2,3,4); // a.length == 4;  
Podemos usar inheritedMethods a.pop() // => 4: outro  
Method a.first // => 1: primeiro getter definido  
porSubclasse.a.last // => 3: último getter definido  
porSubclasse[1] // => 2: sintaxe regular de acesso a  
array ainda funciona. Array.isArray(a) // => true: instância  
da subclasse realmente é um array  
EZArray.isArray(a) // => true: a subclasse herda estática
```

métodos também!

Esta subclasse EZArray define dois métodos getter simples. Instâncias de EZArray se comportam como arrays comuns, e podemos usar inheritedmétodos e propriedades como push(), pop() e length. Mas também podemos usar o primeiro e o último getters definidos na subclasse. Não apenas métodos de instância como pop() são herdados, mas métodos estáticos como Array.isArray também são herdados. Este é um novo recurso habilitado pela sintaxe da classe ES6: EZArray() é uma função, mas herda de Array():

O EZArray herda os métodos de instância porque EZArray.prototype // herda de Array.prototype. Array.prototype.Array.prototype.isPrototypeOf(EZArray.prototype) // => true

E o EZArray herda métodos e propriedades estáticas porque// O EZArray herda do Array. Esta é uma característica especial da palavra-chave extends// e não é possível antes de ES6. Array.isPrototypeOf(EZArray) // => true

Nossa subclasse EZArray é muito simples para ser muito instrutiva. O exemplo 9-6 é um exemplo mais detalhado. Ele define uma subclasse TypedMap da classe Map interna que adiciona verificação de tipo para garantir que as chaves e os valores do mapa sejam dos tipos especificados (de acordo com typeof). É importante ressaltar que este exemplo demonstra o uso da palavra-chave thesuper para invocar o construtor e os métodos da superclasse.

Exemplo 9-6. TypedMap.js: uma subclasse de Map que verifica os tipos de chave e valor

```
class TypedMap extends Map {
```

```
constructor(keyType, valueType, entradas) {  
    Se as entradas forem especificadas, verifique  
    seus tiposif (entradas) {  
        for(let [k, v] de entradas) {  
            if (typeof k !== keyType || typeof v !==  
                tipo_valor) {  
                throw new TypeError('Tipo errado para entrada  
[${}k], ${v}]);  
            }  
        }  
    }  
}
```

*Incialize a superclasse com o comando (tipo marcado)
entradas
iniciaissuper(entradas);*

*Em seguida, inicialize essa subclasse armazenando o
Tipos*

```
this.keyType = keyType;  
this.valueType = valueType;}
```

*Agora redefina o método set() para adicionar verificação de
tipo para qualquer*

*Novas entradas adicionadas ao
map.set(key, value) {
 Gerar um erro se a chave ou o valor forem do erro
 tipo
 if (this.keyType && typeof key !== this.keyType) {
 throw new TypeError('\${key} não é do tipo
 \${this.keyType}');
 }if (this.valueType && typeof value !==
 this.valueType)
 {
 throw new TypeError('\${value} não é do tipo
 \${this.valueType}');
 }
}*

*Se os tipos estiverem corretos, invocamos o
versão do
set(), para realmente adicionar a entrada ao método
mapa. E nós*

```
return o que o método superclasse retorna.return  
super.set(chave, valor);}}
```

Os dois primeiros argumentos para o construtor TypedMap() são os tipos de chave e valor desejados. Devem ser cadeias de caracteres, como "number" e "boolean", que o operador typeof retorna. Você também pode especificar um terceiro argumento: uma matriz (ou qualquer objeto iterável) de matrizes [key,value] que especificam as entradas iniciais no mapa. Se você especificar qualquer entradas iniciais, a primeira coisa que o construtor faz é verificar se seus tipos estão corretos. Em seguida, o construtor invoca o construtor superclass, usando a palavra-chave super como se fosse um nome de função. O construtor Map() recebe um argumento opcional: um objeto iterável de matrizes [chave,valor]. Portanto, o terceiro argumento opcional do construtor TypedMap() é o primeiro argumento opcional para o construtor Map() e o passamos para esse construtor de superclass com super(entries).

Depois de invocar o construtor de superclass para inicializar o estado da superclass, o construtor TypedMap() inicializa seu próprio estado de subclass definindo this.keyType e this.valueType como os tipos especificados. Ele precisa definir essas propriedades para que possa usá-las novamente no método set().

Existem algumas regras importantes que você precisará saber sobre o uso de super() em construtores:

- Se você definir uma classe com a palavra-chave extends, o construtor da sua classe deverá usar super() para invocar o

construtor de superclasse. Se você não definir um construtor em sua subclasse, um serserá definido automaticamente para você. Esse construtor definido implicitamente simplesmente pega quaisquer valores que são passados para ele e passa esses valores para super(). Você não pode usar a palavra-chave this em seu construtor até que você tenha invocado o construtor de superclasse withsuper(). Isso impõe uma regra de que as superclasses podem se inicializar antes das subclasses. A expressão especial new.target é indefinida emfunções que são invocadas sem a nova palavra-chave. No entanto, new.target é uma referência ao construtor que foi invocado. Quando um construtor de subclasse é invocado e usa super() para invocar o construtor de superclasse, esse construtor de superclasse verá o construtor de subclasse como o valor de new.target. Uma superclasse bem projetada não deve precisar saber se foi subclassificada, mas pode ser útil poder usenew.target.name no registro de mensagens, por exemplo. Após o construtor, a próxima parte do Exemplo 9-6 é um método chamado setset(). A superclasse Map define um método chamado set() para adicionar uma nova entrada ao mapa. Dizemos que este método set() em TypedMapsubstitui o método set() de sua superclasse. Esta subclasse TypedMap simples não sabe nada sobre como adicionar novas entradas ao mapa, mas sabesabe como verificar os tipos, então é isso que faz primeiro, verificandose a chave e o valor a serem adicionados ao mapa têm os tipos corretos e lançando um erro se não tiverem. Este método set() não temnenhuma maneira de adicionar a chave e o valor ao mapa em si, mas é para isso que serve o método set() da superclasse. Então, usamos a palavra-chave super novamente

para invocar a versão da superclasse do método. Nesse contexto, super funciona de maneira muito parecida com a palavra-chave this: refere-se ao objeto atual, mas permite acesso a métodos substituídos definidos na superclasse.

Nos construtores, você deve invocar o construtor de superclasse antes de acessá-lo e inicializar o novo objeto por conta própria. Não existem tais regras quando você substitui um método. Um método que substitui um método de superclasse não é necessário para invocar o método superclass. Se ele usar super para invocar o método substituído (ou qualquer método) na superclasse, ele pode fazer isso no início ou no meio ou no final do método de substituição.

Finalmente, antes de deixarmos o exemplo TypedMap para trás, vale a pena notar que essa classe é uma candidata ideal para o uso de campos privados. Como a classe está escrita agora, um usuário pode alterar as propriedades keyType ou valueType para subverter a verificação de tipo. Uma vez que privatefields são suportados, podemos alterar essas propriedades para #keyTypeand #valueType para que não possam ser alteradas de fora.

9.5.3 Delegação em vez de herança

A palavra-chave extends facilita a criação de subclasses. Mas isso não significa que você deva criar muitas subclasses. Se você quiser escrever uma classe que compartilhe o comportamento de alguma outra classe, você pode tentar herdar esse comportamento criando uma subclasse. Mas muitas vezes é mais fácil e flexível obter o comportamento desejado em sua classe, fazendo com que sua classe crie uma instância da outra classe e simplesmente delegando a essa instância conforme necessário. Você cria uma nova classe não por subclasse, mas

em vez disso, envolvendo ou "compondo" outras classes. Essa abordagem de delegação é freqüentemente chamada de "composição" e é uma máxima frequentemente citada da programação orientada a objetos que se deve "favorecer a composição em vez da herança".

Suponha, por exemplo, que quiséssemos uma classe Histogram que se comportasse como a classe Set do JavaScript, exceto que, em vez de apenas acompanhar se um valor foi adicionado a set ou não, ele mantém uma contagem do número de vezes que o valor foi adicionado. Como a API para essa classe Histogram é semelhante a Set, podemos considerar a subclasse Set e adicionar um método count(). Por outro lado, quando começarmos a pensar em como podemos implementar esse método count(), podemos perceber que a classe Histogram é mais parecida com um Map do que com um Set porque precisa manter um mapeamento entre os valores e o número de vezes que eles foram adicionados. Então, em vez de subclasse Set, podemos criar uma classe que define uma API semelhante a Set, mas implementa esses métodos delegando a um objeto Map interno. O exemplo 9-7 mostra como poderíamos fazer isso.

Exemplo 9-7. Histogram.js: uma classe semelhante a Set implementada com delegação

```
/**  
 * Uma classe semelhante a um conjunto que controla quantas  
 * vezes um valor tem  
 * sido adicionado. Chame add() e remove() como faria para  
 * aSet e* chame count() para descobrir quantas vezes um  
 * determinado valor foi adicionado.  
  
 * O iterador padrão produz os valores que foram adicionados  
 * pelo menos  
 * uma vez. Use entries() se quiser iterar [value,  
 * count]pares.  
 */  
class Histograma {
```

Para inicializar, basta criar um objeto Map para delegar

```
construtor() { this.map = new Map(); }
```

Para qualquer chave, a contagem é o valor no mapa, ou zero

```
se a chave não aparecer no Map.count(key) {  
    return this.map.get(key) || 0; }
```

O método Set-like has() retorna true se a contagem for diferente de zero

```
has(chave) { return this.count(chave) > 0; }
```

O tamanho do histograma é apenas o número de entradas no mapa.

```
get size() { return this.map.size; }
```

```
Para adicionar uma chave, basta incrementar sua contagem no  
Map.add(key) { this.map.set(key, this.count(key) + 1); }
```

Excluir uma chave é um pouco mais complicado porque temos que excluir a chave do Mapa se a contagem voltar a zero.

```
excluir(chave) {  
    let contagem = this.count(chave);  
    if (contagem === 1) {  
        this.map.delete(chave);}  
    senão se (contagem > 1) {  
        this.map.set(chave, contagem - 1);}}
```

A iteração de um histograma apenas retorna as chaves armazenadas nele[Symbol.iterator](): { return this.map.keys(); }

Esses outros métodos iteradores apenas delegam ao objeto Map

```
chaves() { return this.map.keys();}  
}valores() { return this.map.values();}  
}entradas() { return this.map.entries(); }}
```

Tudo o que o construtor `Histogram()` faz no [Exemplo 9-7](#) é criar um objeto `Map`. E a maioria dos métodos são one-liners que apenas delegam a um método do mapa, tornando a implementação bastante simples.

Como usamos delegação em vez de herança, um objeto `Histogram` não é uma instância de `Set` ou `Map`. Mas o `Histogram` implementa vários métodos `Set` comumente usados e, em uma linguagem não tipada como JavaScript, isso geralmente é bom o suficiente: uma relação formal de herança às vezes é boa, mas geralmente opcional.

9.5.4 Hierarquias de classe e classes abstratas

O exemplo 9-6 demonstrou como podemos subclasse `Map`. O exemplo 9-7 demonstrou como podemos delegar a um objeto `Map` sem realmente subclassificar nada. Usar classes JavaScript para encapsular dados e modularizar seu código geralmente é uma ótima técnica, e você pode usar a palavra-chave `class` com frequência. Mas você pode descobrir que prefere a composição à herança e que raramente precisa usar extensões (exceto quando estiver usando uma biblioteca ou estrutura que exija que você estenda suas classes base).

No entanto, há algumas circunstâncias em que vários níveis de subclasse são apropriados, e terminaremos este capítulo com um exemplo estendido que demonstra uma hierarquia de classes que representam diferentes tipos de conjuntos. (As classes set definidas no [Exemplo 9-8](#) são semelhantes, mas não completamente compatíveis com a classe `Set` integrada do JavaScript.)

O exemplo 9-8 define muitas subclasses, mas também demonstra como você pode definir classes abstratas — classes que não incluem uma implementação completa — para servir como uma superclasse comum para um grupo de

subclasses relacionadas. Uma superclasse abstrata pode definir uma implementação parcial que todas as subclasses herdam e compartilham. As subclasses, então, só precisam definir seu próprio comportamento único implementando os métodos abstratos definidos - mas não implementados - pela superclasse. Observe que o JavaScript não tem nenhuma definição formal de métodos abstratos ou classes abstratas; Estou simplesmente usando esse nome aqui para métodos não implementados e classes implementadas de forma incompleta.

O exemplo 9-8 é bem comentado e se sustenta por si só. Eu encorajo você a lê-lo como um exemplo fundamental para este capítulo sobre classes. A classe final no Exemplo 9-8 faz muita manipulação de bits com os operadores &, |, e ~, que você pode revisar em §4.8.3.

Exemplo 9-8. Sets.js: uma hierarquia de classes de conjuntos abstratos e concretos

```
/**  
 * A classe AbstractSet define um único método  
 * abstrato, has().  
 */  
class AbstractSet {  
    Lance um erro aqui para que as subclasses sejam  
    forçadas// a definir sua própria versão de trabalho deste  
    método. has(x) { throw new Error("Método abstrato"); }}  
  
    /**  
     * NotSet é uma subclasse concreta de AbstractSet.* Os membros  
     * deste conjunto são todos valores que não são membros de algum  
     * outro conjunto. Por ser definido em termos de outro  
     * conjunto, não é  
     * gravável, e porque tem membros infinitos, não é  
     * enumerável.  
     * Tudo o que podemos fazer com ele é testar a adesão e  
     * convertê-lo em um  
     * string usando notação  
     * matemática.*/  
    class NotSet extends AbstractSet {
```

```
    construtor(conjunto) {
        super(); this.set
        = conjunto;
    }
```

*Nossa implementação do método abstrato que herdamos `has(x)` {
return !this.set.has(x); } E também substituímos este objeto
`methodToString() { return '{ x| x > ${this.set.toString()}' };` }*

```
/*
 * Range set é uma subclasse concreta de AbstractSet. Seus
 * membros são
 * todos os valores que estão entre os limites de e
 * até, inclusive.
 * Como seus membros podem ser números de ponto flutuante,
 * ele não é* enumerável e não tem um tamanho
 * significativo.*/class RangeSet extends AbstractSet {
```

```
    construtor(de, para) {
        super(); this.from =
        de; this.to = para;}
```

`has(x) { return x >= this.from && x <= this.to; }toString() {
return '{ x| ${this.from} ≤ x ≤ ${this.to} }';}}`

```
/*
 * AbstractEnumerableSet é uma subclasse abstrata
 * ofAbstractSet. It define
 * um getter abstrato que retorna o tamanho do conjunto e
 * também define um
 * iterador abstrato. E então implementa
 * concreteIsEmpty(), toString(),
 * e equals() em cima deles. Subclasses que implementam
 * o
 * iterador, o getter de tamanho e o método has() obtêm
 * theseconcrete
 * * Métodos
 * gratuitos.*/
```

```

class AbstractEnumerableSet estende AbstractSet {
    get size() { throw new Error("Método abstrato"); }
    [Símbolo.iterator]() { throw new Error("Método abstrato"); }

    isEmpty() { return this.size === 0; }toString() { return
    '${Array.from(this).join(", ")}'; }igual a (conjunto)
    {
        Se o outro conjunto também não for enumerável, ele não será
        igual a este
        se (!( set instanceof AbstractEnumerableSet)) return
        falso;

        Se eles não tiverem o mesmo tamanho, eles não serão
        iguaisif (this.size !== set.size) return false;

        Percorra os elementos deste elemento
        setfor(let this) {
            Se um elemento não estiver no outro conjunto, eles
            não são iguais
            if (!set.has(element)) return false;}
    }
}

```

*Os elementos corresponderam, então os conjuntos são
iguaisreturn true;}}*

```

/*
* SingletonSet é uma subclasse concreta
de AbstractEnumerableSet.
* Um conjunto singleton é um conjunto somente leitura com
um único membro.*/class SingletonSet extends
AbstractEnumerableSet {
    construtor(membro) {
        super(); this.member =
        membro;}
}

```

*Implementamos esses três métodos e herdamos isEmpty, equals()
e toString() implementações baseadas nestes
métodos.has(x) { return x === this.member; }get size() {
return 1; }*

```
*[Symbol.iterator]() { yield this.member; }

/*
* AbstractWritableSet é uma subclasse abstrata
de AbstractEnumerableSet.
* Ele define os métodos abstratos insert() e remove()
thatinsert e
* remova elementos individuais do conjunto e, em seguida,
implemente concreto
* add(), subtract() e intersect() em cima deles. Observe que
* nossa API diverge aqui do JavaScript Setclass padrão.

*/class AbstractWritableSet se
estende                                AbstractEnumerableSet {
    insert(x) { throw new Error("Método abstrato");
    }remove (x) { throw new Error ("Método abstrato"); }

    adicionar (conjunto) {
        for(let elemento de conjunto) {
        this.insert(elemento);}

    subtrair (definir) {
        for(let elemento de conjunto) {
        this.remove(elemento);}

    intersect(set) {
        for(let elemento disso) {
            if (!set.has(elemento)) {
this.remove(elemento);}}}

/***
* Um BitSet é uma subclasse concreta de AbstractWritableSet com
```

um

** Implementação de conjunto de tamanho fixo muito eficiente para conjuntos cujos elementos * são inteiros não negativos menores que algum tamanho máximo.*

```
*/class BitSet extends
AbstractWritableSet {
    Construtor(max) {
        super();
        this.max = máx; O número inteiro máximo que podemos armazenar.
        this.n = 0; Quantos números inteiros estão no conjunto
        this.numBytes = Math.floor(max / 8) + 1; Quantos bytes que precisamos
        this.data = new Uint8Array(this.numBytes); Os bytes}
```

Método interno para verificar se um valor é um membro legal deste conjunto

```
_valid(x) { return Number.isInteger(x) && x >= 0 && x
<=this.max; }
```

Testa se o bit especificado do byte especificado de nosso

*matriz de dados está definida ou não. Retorna true ou false._has(byte, bit) { return (this.data[byte]
&BitSet.bits[bit]) !== 0; }*

```
O valor x neste BitSet?has(x) {

    if (this._valid(x)) {
        let byte = Math.floor(x / 8);
        seja bit = x % 8; return
        this._has(byte, bit);} else {

    return false;}}
```

Insira o valor x no BitSetinsert(x)

```
{
```

válido

```
    if (this._valid(x)) { Se o valor for
        let byte = Math.floor(x / 8); converter em byte
```

```

e mordeu
    seja bit = x % 8; if
        (!this._has(byte, bit)) {      Se esse bit for
Ainda não definido
            this.data[byte] |= BitSet.bits[bit]; então
Defina
            this.n++;                      e
incrementar o tamanho do conjunto
        } } else
        {
            throw new TypeError("Elemento de conjunto inválido: " + x
        ); } }

```

```

remover (x) {
    if (this._valid(x)) {           Se o valor for
válido
        let byte = Math.floor(x / 8); Calcule o byte
e mordeu
        seja bit = x % 8; if
            (this._has(byte, bit)) {      Se esse bit for
já definido
                this.data[byte] &= BitSet.masks[bit]; então
desmarque isso
                isso.n--;                  e
diminuir o tamanho
            } } else
            {
                throw new TypeError("Elemento de conjunto inválido: " + x
            ); } }

```

*Um getter para retornar o tamanho do
setget size() { return this.n; }*

*Itere o conjunto apenas verificando cada bit por vez.//
(Poderíamos ser muito mais espertos e otimizar isso
substancialmente)*

```

*[Symbol.iterator]() {
    for(let i = 0; i <= this.max; i++) {
        if (this.has(i)) {
            rendimento i;

```

```
}}})
```

Alguns valores pré-computados usados pelos métodos has(), insert() e remove() BitSet.bits = new Uint8Array([1, 2, 4, 8, 16, 32, 64, 128]); BitSet.masks = new Uint8Array([~1, ~2, ~4, ~8, ~16, ~32, ~64, ~128]);

9.6 Resumo

Este capítulo explicou os principais recursos das classes JavaScript:

- Objetos que são membros da mesma classe herdam propriedades do mesmo objeto de protótipo. O objeto protótipo é o principal recurso das classes JavaScript e é possível definir classes com nada mais do que o método `Object.create()`. Antes do ES6, as classes eram mais tipicamente definidas definindo primeiro uma função construtora. As funções criadas com a palavra-chave `function` têm uma propriedade `prototype` e o valor dessa propriedade é um objeto que é usado como o protótipo de todos os objetos criados quando a função é invocada com `new` como um construtor. Ao inicializar esse objeto de protótipo, você pode definir os métodos compartilhados de sua classe. Embora o objeto `prototype` seja o principal recurso da classe, a função construtora é a identidade pública da classe. O ES6 introduz
- uma palavra-chave de classe que facilita a definição de classes, mas sob o capô, o construtor e o mecanismo de protótipo permanecem os mesmos. As subclasses são definidas usando a palavra-chave `extends` em uma classe

- declaração. As subclasses podem invocar o construtor de sua superclasse ou métodos substituídos de sua superclasse com a superpalavra-chave.

1 Exceto funções retornadas pelo método ES5 Function.bind(). As funções vinculadas não têm nenhuma propriedade de protótipo própria, mas usam o protótipo do subjacente se eles forem invocados como construtores.

2 Veja Design Patterns (Addison-Wesley Professional) de Erich Gamma et al. ou EffectiveJava (Addison-Wesley Professional) de Joshua Bloch, por exemplo.

Capítulo 10. Módulos

O objetivo da programação modular é permitir que grandes programas sejam montados usando módulos de código de autores e fontes diferentes e que todo esse código seja executado corretamente, mesmo na presença de código que os vários autores do módulo não previram. Na prática, a modularidade consiste principalmente em encapsular ou ocultar detalhes de implementação privada e manter o namespace global organizado para que os módulos não possam modificar acidentalmente as variáveis, funções e classes definidas por outros módulos.

Até recentemente, o JavaScript não tinha suporte embutido para módulos, e os programadores que trabalhavam em grandes bases de código faziam o possível para usar a modularidade fraca disponível por meio de classes, objetos e fechamentos. A modularidade baseada em encerramento, com suporte de ferramentas de empacotamento de código, levou a uma forma prática de modularidade baseada em uma função `require()`, que foi adotada pelo Node. Os módulos baseados em `require()` são uma parte fundamental do ambiente de programação do Node, mas nunca foram adotados como parte oficial da linguagem JavaScript. Em vez disso, o ES6 define módulos usando palavras-chave de importação e exportação. Embora a importação e a exportação façam parte da linguagem há anos, elas só foram implementadas por navegadores da web e pelo Node há relativamente pouco tempo. E, na prática, a modularidade do JavaScript ainda depende de ferramentas de empacotamento de código.

As seções a seguir cobrem:

- Módulos do tipo "faça você mesmo" com classes, objetos e encerramentos
- Módulos de nó usando require()
- ES6 usando export, import e import()

10.1 Módulos com classes, objetos e encerramentos

Embora possa ser óbvio, vale a pena salientar que uma das características importantes das classes é que elas funcionam como módulos para os seus métodos. Pense no Exemplo 9-8. Esse exemplo definiu um número de classes diferentes, todas com um método chamado has(). Mas você não teria nenhum problema em escrever um programa que usasse várias classes set() desse exemplo: não há perigo de que a implementação de has() de SingletonSet substitua o método has() de BitSet, por exemplo.

A razão pela qual os métodos de uma classe são independentes dos métodos de outras classes não relacionadas é que os métodos de cada classe são definidos como propriedades de objetos protótipos independentes. A razão pela qual as classes são modulares é que os objetos são modulares: definir uma propriedade em um objeto JavaScript é muito parecido com declarar uma variável, mas adicionar propriedades a objetos não afeta o namespace global de um programa, nem afeta as propriedades de outros objetos. O JavaScript define algumas funções matemáticas e constantes, mas em vez de definirlas globalmente, elas são agrupadas como propriedades de um único objeto Math global. Essa mesma técnica poderia ter sido usada no Exemplo 9-8. Em vez de definir classes globais com nomes como SingletonSet e BitSet, esse exemplo poderia ter sido escrito para definir apenas um único Sets global

, com propriedades que fazem referência às várias classes. Os usuários da biblioteca thisSets podem então se referir às classes com nomes comoSets.Singleton e Sets.Bit.

Usar classes e objetos para modularidade é uma técnica comum e útil na programação JavaScript, mas não vai longe o suficiente. Em particular, ele não nos oferece nenhuma maneira de ocultar detalhes de implementação interna dentro do módulo. Considere o Exemplo 9-8 novamente. Se estivéssemos escrevendo esse exemplo como um módulo, talvez quiséssemos manter as várias classes abstratas internas ao módulo, disponibilizando apenas as subclasses concretas para os usuários do módulo. Da mesma forma, na classe BitSet, os métodos _valid() e _has() são internalutilities que não devem ser expostos aos usuários da classe. AndBitSet.bits e BitSet.masks são detalhes de implementação que ficariam melhor escondidos.

Como vimos no §8.6, variáveis locais e funções aninhadas declaradas dentro de uma função são privadas para essa função. Isso significa que podemos usar expressões de função invocadas imediatamente para alcançar um tipo demodularidade, deixando os detalhes de implementação e funções utilitárias ocultos dentro da função delimitadora, mas tornando a API pública domódulo o valor de retorno da função. No caso da classe BitSet, podemos estruturar o módulo assim:

```
const BitSet = (function() { // Define BitSet para o
  valor de retorno desta função
  Detalhes da implementação privada aqui
  function isValid(set, n) { ... }
  function has(set, byte, bit) {
    ...
  }
  const BITS = new Uint8Array([1, 2, 4, 8, 16, 32,
    64, 128]);
}
```

```

const MASKS = new Uint8Array([~1, ~2, ~4, ~8, ~16,
~32, ~64, ~128]);

A API pública do módulo é apenas a classe BitSet, que
definimos
e volte aqui. A classe pode usar as privatefunctions
e as constantes
definido acima, mas eles serão ocultados dos usuários da
classe
    return class BitSet extends AbstractWritableSet {
// ... implementação omitida ...};}());

```

Essa abordagem da modularidade se torna um pouco mais interessante quando o módulo contém mais de um item. O código a seguir, por exemplo, define um mini módulo de estatísticas que exporta as funções mean() e stddev(), deixando os detalhes da implementação ocultos:

```

É assim que poderíamos definir um stats
moduleconst stats = (function() {
    Funções utilitárias privadas para o
    móduloconst soma = (x, y) => x + y; const
    quadrado = x => x * x;

    Uma função pública que será
    exportadafunction mean(data) {
        return data.reduce(sum)/data.length;}


```

```

Uma função pública que
exportaremosfunction stddev(data) {
    let m = média(dados);
    return Math.sqrt(
        data.map(x => x -
m).map(quadrado).reduce(soma)/(data.length-1)
    );}

```

Exportamos a função pública como propriedades de um

```
objeto  
return { significa, stddev  
};}());
```

*E aqui está como podemos usar o
modulestats.mean([1, 3, 5, 7, 9])// =>
5stats.stddev([1, 3, 5, 7, 9]) // => Math.sqrt(10)*

10.1.1 Automatizando a modularidade baseada em fechamento

Observe que é um processo bastante mecânico transformar um arquivo de código JavaScript nesse tipo de módulo, inserindo algum texto no início e no final do arquivo. Tudo o que é necessário é alguma convenção para que o arquivo de código JavaScript indique quais valores devem ser exportados e quais não são.

Imagine uma ferramenta que pega um conjunto de arquivos, envolve o conteúdo de cada um desses arquivos em uma expressão de função imediatamente invocada, controla o valor de retorno de cada função e concatena tudo em um grande arquivo. O resultado pode ser algo assim:

```
const módulos = {}; function require(moduleName) { return  
módulos[moduleName]; }  
  
módulos["sets.js"] = (function() {  
    const exportações = {};  
  
    O conteúdo do arquivo sets.js vai  
aqui:exportações. BitSet = class BitSet { ... };  
  
    exportações de  
    retorno;})();  
  
módulos["stats.js"] = (função() {  
    const exportações = {};  
  
O conteúdo do arquivo stats.js está aqui:
```

```
soma const = (x, y) => x + y; const
quadrado = x => x * x; exportações.média
= função(dados) { ... }; exports.stddev
= function(data) { ... };

exportações de
retorno;}());

```

Com módulos agrupados em um único arquivo como o mostrado no exemplo anterior, você pode imaginar escrever um código como o seguinte para fazer uso desses módulos:

Obtenha referências aos módulos (ou ao conteúdo do módulo) de que precisa
const stats = require("stats.js"); const
BitSet = require("sets.js"). BitSet;

Agora escreva o código usando esses módulos
let s
= new BitSet(100); s.insert(10); s.insert(20);
s.insert(30); let média = stats.mean([... s]); a
média é 20

Este código é um esboço de como funcionam as ferramentas de agrupamento de código (como webpack e Parcel) para navegadores da web, e também é uma introdução simples à função require() como a usada em programas Node.

10.2 Módulos no nó

Na programação Node, é normal dividir os programas em quantos arquivos parecer natural. Esses arquivos de código JavaScript são assumidos como todos liveon um sistema de arquivos rápido. Ao contrário dos navegadores da web, que precisam ler arquivos de

JavaScript em uma conexão de rede relativamente lenta, não há necessidade ou benefício em agrupar um programa Node em um único arquivo JavaScript.

No Node, cada arquivo é um módulo independente com um namespace privado. Constantes, variáveis, funções e classes definidas em um arquivo são privadas para esse arquivo, a menos que o arquivo as exporte. E os valores exportados por um módulo só são visíveis em outro módulo se esse módulo os importar explicitamente.

Os módulos de nó importam outros módulos com a função require() e exportam sua API pública definindo propriedades do objeto Exports substituindo totalmente o objeto module.exports.

10.2.1 Exportações de nós

Node define um objeto de exportações globais que é sempre definido. Se você estiver escrevendo um módulo Node que exporta vários valores, você pode simplesmente atribuí-los às propriedades deste objeto:

```
soma const = (x, y) => x + y;
const quadrado = x => x * x;

exportações.média = dados => dados.reduce(soma)/dados.comprimento;
exports.stddev = function(d) {
  let m = exportações.média(d); return
  Math.sqrt(d.map(x => x -
    m).map(square).reduce(sum)/(d.length-
    1));};
```

Muitas vezes, no entanto, você deseja definir um módulo que exporta apenas uma única função ou classe, em vez de um objeto cheio de funções ou classes. Para isso, basta atribuir o valor único para o qual deseja exportar

module.exports:

```
module.exports = class BitSet extends AbstractWritableSet {  
  implementação omitida};
```

O valor padrão de module.exports é o mesmo objeto ao qual exports se refere. No módulo de estatísticas anterior, poderíamos ter atribuído a função média a module.exports.mean em vez de exports.mean.

Outra abordagem com módulos como o statsmodule é exportar um único objeto no final do módulo, em vez de exportar funções uma a uma à medida que avança:

```
Defina todas as funções, public e privateconst  
sum = (x, y) => x + y; const quadrado = x => x *  
x;const média = dados =>  
data.reduce(sum)/data.length; const stddev = d =>  
{  
  seja m = média(d); return  
  Math.sqrt(d.map(x => x -  
  m).map(square).reduce(sum)/(d.length-  
  1));};
```

```
Agora exporte apenas o  
público module.exports = { mean, stddev };
```

10.2.2 Importações de nós

Um módulo Node importa outro módulo chamando a função require(). O argumento para essa função é o nome do módulo a ser importado e o valor retornado é qualquer valor (normalmente uma função, classe ou objeto) que o módulo exporta.

Se você deseja importar um módulo do sistema integrado ao Node ou a um módulo

que você instalou em seu sistema por meio de um gerenciador de pacotes, então você simplesmente usa o nome não qualificado do módulo, sem nenhum caractere "/" que o transformaria em um caminho do sistema de arquivos:

```
Esse módulos são integrados ao Node
const fs = require("fs"); // O módulo do sistema de arquivos
const http = require("http"); // O módulo HTTP integrado
```

```
A estrutura do servidor HTTP Express é um módulo de terceiros.
// Não faz parte do Node, mas foi instalado localmente
const express = require("express");
```

Quando você deseja importar um módulo de seu próprio código, o nome do módulo deve ser o caminho para o arquivo que contém esse código, em relação ao arquivo do módulo atual. É legal usar caminhos absolutos que começam com um caractere /, mas normalmente, ao importar módulos que fazem parte de seu próprio programa, os nomes dos módulos começarão com ./ ou às vezes .. / para indicar que eles são relativos ao diretório atual ou ao diretório pai. Por exemplo:

```
const stats = require('./stats.js'); const
BitSet = require('./utils/bitset.js');
```

(Você também pode omitir o sufixo .js nos arquivos que está importando e o Node ainda encontrará os arquivos, mas é comum ver essas extensões de arquivo explicitamente incluídas.)

Quando um módulo exporta apenas uma única função ou classe, tudo o que você precisa fazer é exigi-lo. Quando um módulo exporta um objeto com várias propriedades, você tem uma escolha: você pode importar o objeto inteiro ou apenas importar as propriedades específicas (usando a atribuição de desestruturação) do

que você planeja usar. Compare estas duas abordagens:

Importe todo o objeto stats, com todas as suas funções
const stats = require('./stats.js');

Temos mais funções do que precisamos, mas elas estão organizadas em um namespace conveniente de "estatísticas".
let média = estatísticas.média(dados);

Alternativamente, podemos usar a atribuição de desestruturação idiomática para importar// exatamente as funções que queremos diretamente para o localnamespace:
const { stddev } = require('./stats.js');

Isso é bom e sucinto, embora percais um pouco de contexto // sem o prefixo 'stats' como um namespace para a função stddev().
let sd = stddev(data);

10.2.3 Módulos de estilo de nó na Web

Módulos com um objeto Exports e uma função require() são integrados ao Node. Mas se você estiver disposto a processar seu código com uma ferramenta de empacotamento como o webpack, também é possível usar esse estilo de módulos para código que se destina a ser executado em navegadores da web. Até recentemente, isso era uma coisa muito comum de se fazer, e você pode ver muitos códigos baseados na web que ainda fazem isso.

Agora que o JavaScript tem sua própria sintaxe de módulo padrão, no entanto, os desenvolvedores que usam empacotadores são mais propensos a usar os módulos officialJavaScript com instruções de importação e exportação.

10.3 Módulos no ES6

O ES6 adiciona palavras-chave de importação e exportação para JavaScript e, finalmente, suporta modularidade real como um recurso central da linguagem. A modularidade do ES6 é conceitualmente a mesma que a modularidade do Node: cada arquivo é seu próprio módulo e constantes, variáveis, funções e classes definidas em um arquivo são privadas para esse módulo, a menos que sejam explicitamente exportadas. Os valores exportados de um módulo estão disponíveis para uso em módulos que os importam explicitamente. Os módulos ES6 diferem dos módulos do Node na sintaxe usada para exportar e importar e também na maneira como os módulos são definidos nos navegadores da web. As seções a seguir explicam essas coisas em detalhes.

Primeiro, porém, observe que os módulos ES6 também são diferentes dos "scripts" JavaScript regulares em alguns aspectos importantes. A diferença mais óbvia é a própria modularidade: em scripts regulares, declarações de nível superior de variáveis, funções e classes entram em um único contexto global compartilhado por todos os scripts. Com módulos, cada arquivo tem seu próprio contexto privado e pode usar as instruções de importação e exportação, que é o ponto principal, afinal. Mas também existem outras diferenças entre módulos e scripts. O código dentro de um módulo ES6 (como o código dentro de qualquer definição de classe ES6) está automaticamente no modo estrito (consulte §5.6.3). Isso significa que, quando você começar a usar os módulos ES6, nunca mais precisará escrever "usar estrito". E isso significa que o código em módulos não pode usar a instrução `with` ou o objeto `arguments` ou variáveis não declaradas. Os módulos ES6 são até um pouco mais rígidos do que o modo estrito: no modo estrito, em funções invocadas como funções, isso é indefinido. Nos módulos, isso é indefinido mesmo no código de nível superior. (Por outro lado, os scripts em navegadores da Web e no Node definem isso como o objeto `global`.)

ES6 MÓDULOS NA WEB E NO NÓ

Os módulos ES6 estão em uso na web há anos com a ajuda de empacotadores de código, como o webpack, que combinam módulos independentes de código JavaScript em pacotes grandes e não modulares adequados para inclusão em páginas da web. No momento da redação deste artigo, no entanto, os módulos ES6 são finalmente suportados nativamente por todos os navegadores da web, exceto o Internet Explorer. Quando usados nativamente, os módulos ES6 são adicionados a páginas HTML com uma tag especial <script type="module">, descrita mais adiante neste capítulo.

E enquanto isso, tendo sido pioneiro na modularidade do JavaScript, o Node se encontra na posição embaraçosa de ter que suportar dois sistemas de módulos não totalmente compatíveis. O Node 13 suporta módulos ES6, mas, por enquanto, a grande maioria dos programas Node, ainda usa módulos Node.

10.3.1 Exportações ES6

Para exportar uma constante, variável, função ou classe de um módulo ES6, basta adicionar a palavra-chave `export` antes da declaração:

```
export const PI = Math.PI;

função de exportação degreesToRadians(d) { return d * PI / 180; }

exportar classe Circle {
construtor(r) { this.r = r; }area() {
return PI * this.r * this.r; }}
```

Como alternativa à dispersão de palavras-chave de exportação em todo o seu módulo, você pode definir suas constantes, variáveis, funções e classes como faria normalmente, sem nenhuma instrução de exportação e, em seguida, (normalmente no final do módulo) escrever uma única declaração de exportação que declara exatamente o que é exportado em um único lugar. Então, em vez de

escrevendo três exportações individuais no código anterior, poderíamos ter escrito uma única linha no final:

```
export { Círculo, grausParaRadianos, PI };
```

Essa sintaxe se parece com a palavra-chave `export` seguida por um objectliteral (usando notação abreviada). Mas, nesse caso, as chaves não definem realmente um literal de objeto. Essa sintaxe de exportação simplesmente requer uma lista separada por vírgulas de identificadores entre chaves.

É comum escrever módulos que exportam apenas um valor (normalmente uma função ou classe) e, neste caso, geralmente usamos `export default` em vez de `export`:

```
exportar classe padrão BitSet {  
    implementação omitida}
```

As exportações padrão são um pouco mais fáceis de importar do que as exportações não padrão, portanto, quando há apenas um valor exportado, usar `export default` facilita as coisas para os módulos que usam o valor exportado.

As exportações regulares com exportação só podem ser feitas em declarações que tenham um nome. As exportações padrão com `export default` podem exportar qualquer expressão, incluindo expressões de função anônimas e expressões anonymousclass. Isso significa que, se você usar o padrão de exportação, poderá exportar literais de objeto. Portanto, ao contrário da sintaxe de exportação, se você colocar chaves após o padrão de exportação, é realmente um literal de objeto que está sendo exportado.

É legal, mas um tanto incomum, que os módulos tenham um conjunto de exportações regulares e também uma exportação padrão. Se um módulo tiver um defaultexport, ele só poderá ter um.

Por fim, observe que a palavra-chave export só pode aparecer no nível superior do seu código JavaScript. Você não pode exportar um valor de dentro de uma classe, função, loop ou condicional. (Este é um recurso importante do sistema de módulos ES6 e permite a análise estática: uma exportação de módulos será a mesma em todas as execuções e os símbolos exportados podem ser determinados antes que o módulo seja realmente executado.)

10.3.2 Importações ES6

Você importa valores que foram exportados por outros módulos com a palavra-chave import. A forma mais simples de importação é usada para módulos que definem uma exportação padrão:

```
importar BitSet de './bitset.js';
```

Esta é a palavra-chave import, seguida por um identificador, seguida pela palavra-chave from, seguida por um literal de string que nomeia o módulo cuja exportação padrão estamos importando. O valor de exportação padrão do módulo especificado torna-se o valor do identificador especificado no módulo atual.

O identificador ao qual o valor importado é atribuído é uma constante, como se tivesse sido declarado com a palavra-chave const. Assim como as exportações, as importações aparecem apenas no nível superior de um módulo e não são permitidas dentro de classes, funções, loops ou condicionais. Por convenção quase universal, as importações necessárias para um módulo são colocadas no início de

o módulo. Curiosamente, no entanto, isso não é necessário: como as declarações de função, as importações são "içadas" para o topo e todos os valores importados estão disponíveis para qualquer uma das execuções de código do módulo.

O módulo do qual um valor é importado é especificado como um literal de cadeia constante entre aspas simples ou duplas. (Você não pode usar uma variável ou outra expressão cujo valor seja uma cadeia de caracteres e não pode usar uma cadeia de caracteres entre acentos graves porque os literais de modelo podem interpolar variáveis e nem sempre têm valores constantes.) Em navegadores da web, essa string é interpretada como uma URL relativa à localização do módulo que está fazendo a importação. (No Node, ou ao usar uma ferramenta de agrupamento, a string é interpretada como um nome de arquivo em relação ao módulo atual, mas isso faz pouca diferença na prática.) Uma string de especificador de módulo deve ser um caminho absoluto começando com "/", ou um caminho relativo começando com "./" ou "../", ou um URL completo com protocolo e nome de host. A especificação ES6 não permite strings especificadoras de módulo não qualificadas como "util.js" porque é ambíguo se isso se destina a nomear um módulo no mesmo diretório que o atual ou algum tipo de módulo de sistema instalado em algum local especial. (Essa restrição contra "especificadores de módulos simples" não é respeitada por ferramentas de agrupamento de código como o webpack, que pode ser facilmente configurado para encontrar módulos simples em um diretório de biblioteca que você especificar.) Uma versão futura da linguagem pode permitir "especificadores de módulo simples", mas, por enquanto, eles não são permitidos. Se você deseja importar um módulo do mesmo diretório que o atual, basta colocar "./" antes do nome do módulo e importar de "./util.js" em vez de "util.js".

Até agora, consideramos apenas o caso de importar um único valor de um módulo que usa export default. Para importar valores de um

módulo que exporta vários valores, usamos uma sintaxe ligeiramente diferente:

```
import { mean, stddev } from './stats.js';
```

Lembre-se de que as exportações padrão não precisam ter um nome no módulo que as define. Em vez disso, fornecemos um nome local quando importamos esses valores. Mas as exportações não padrão de um módulo têm nomes no módulo de exportação e, quando importamos esses valores, nos referimos a eles por esses nomes. O módulo de exportação pode exportar qualquer número de valor nomeado. Uma instrução de importação que faz referência a esse módulo pode importar qualquer subconjunto desses valores simplesmente listando seus nomes entre chaves. As chaves fazem com que esse tipo de instrução de importação pareça algo como uma atribuição de desestruturação, e a atribuição de desestruturação é, na verdade, uma boa analogia para o que esse estilo de importação está fazendo. Os identificadores entre chaves são todos içados para a parte superior do módulo de importação e se comportam como constantes.

Às vezes, os guias de estilo recomendam que você importe explicitamente todos os símbolos que seu módulo usará. Ao importar de um módulo que define muitas exportações, no entanto, você pode importar tudo facilmente com uma instrução de importação como esta:

```
import * as estatísticas from './stats.js';
```

Uma instrução de importação como essa cria um objeto e o atribui a uma estatística nomeada constante. Cada uma das exportações não padrão do módulo que está sendo importado torna-se uma propriedade desse objeto stats. As exportações não padrão sempre têm nomes, e eles são usados como nomes de propriedade dentro do objeto. Essas propriedades são efetivamente constantes: elas

não pode ser substituído ou excluído. Com a importação curinga mostrada no exemplo anterior, o módulo de importação usaria as funções importedmean() e stddev() por meio do objeto stats, invocando-as como stats.mean() e stats.stddev().

Os módulos normalmente definem uma exportação padrão ou várias exportações nomeadas. É legal, mas um tanto incomum, que um módulo use export e export default. Mas quando um módulo faz isso, você pode importar o valor padrão e os valores nomeados com uma instrução de importação como esta:

```
import Histogram, { mean, stddev } from  
  "./histogram-stats.js";
```

Até agora, vimos como importar de módulos com uma exportação padrão e de módulos com exportações não padrão ou nomeadas. Mas há uma outra forma de instrução de importação que é usada com módulos que não têm nenhuma exportação. Para incluir um módulo sem exportação em seu programa, basta usar a palavra-chave import com o especificador de módulo:

```
importar "./analytics.js";
```

Um módulo como este é executado na primeira vez que é importado. (E as importações subsequentes não fazem nada.) Um módulo que apenas define funções só é útil se exportar pelo menos uma dessas funções. Mas se um módulo executa algum código, pode ser útil importar mesmo sem símbolos. O módulo Ananalytics para um aplicativo Web pode executar código para registrar vários manipuladores de eventos e, em seguida, usar esses manipuladores de eventos para enviar dados de telemetria de volta ao servidor em momentos apropriados. O módulo é independente e não precisa exportar nada, mas ainda precisamos

importá-lo para que ele realmente seja executado como parte do nosso programa.

Observe que você pode usar essa sintaxe de importação sem importação mesmo com módulos que têm exportações. Se um módulo definir um comportamento útil independentemente dos valores que exporta e se o programa não precisar de nenhum desses valores exportados, você ainda poderá importar o módulo . apenas para esse comportamento padrão.

10.3.3 Importações e exportações com renomeação

Se dois módulos exportarem dois valores diferentes usando o mesmo nome e você quiser importar esses dois valores, será necessário renomear um ou ambos os valores ao importá-lo. Da mesma forma, se você quiser importar um valor cujo nome já está em uso em seu módulo, você precisará renomear o valor importado. Você pode usar a palavra-chave as `withnamed imports` para renomeá-las à medida que as importa:

```
import { render as renderImage } from "./imageutils.js";
import { render as renderUI } from "./ui.js";
```

Essas linhas importam duas funções para o módulo atual. As funções são nomeadas `render()` nos módulos que as definem, mas são importadas com os nomes mais descritivos e desambiguantes `renderImage()` e `renderUI()`.

Lembre-se de que as exportações padrão não têm um nome. O módulo de importação sempre escolhe o nome ao importar uma exportação padrão. Portanto, não há necessidade de uma sintaxe especial para renomear nesse caso.

Dito isto, no entanto, a possibilidade de renomear na importação

Fornece outra maneira de importar de módulos que definem uma exportação padrão e exportações nomeadas. Lembre-se do módulo "./histogram-stats.js" da seção anterior. Aqui está outra maneira de importar as exportações padrão e nomeadas desse módulo:

```
import { default as Histogram, mean, stddev }  
from "./histogram-stats.js";
```

Nesse caso, a palavra-chave JavaScript `default` serve como um espaço reservado e nos permite indicar que queremos importar e fornecer um nome para a exportação padrão do módulo.

Também é possível renomear valores à medida que você os exporta, mas somente ao usar a variante de chaves da instrução `export`. Não é comum precisar fazer isso, mas se você escolheu nomes curtos e sucintos para uso dentro de seu módulo, talvez prefira exportar seus valores com nomes mais descritivos que são menos propensos a entrar em conflito com outros módulos. Assim como acontece com as importações, você usa a palavra-chave `as` para fazer isso:

```
exportar {  
  layout como  
  calculateLayout, render como  
  renderLayout};
```

Lembre-se de que, embora as chaves pareçam algo como object literals, elas não são, e a palavra-chave `export` espera um single identifier antes do `as`, não uma expressão. Isso significa, infelizmente, que você não pode usar a renomeação de exportação assim:

```
export { Math.sin disso, Math.cos de cos }; Erro de sintaxe
```

10.3.4 Reexportações

Ao longo deste capítulo, discutimos um módulo hipotético `"./stats.js"` que exporta as funções `mean()` e `stddev()`. Se estivéssemos escrevendo tal módulo e pensássemos que muitos usuários do módulo desejariam apenas uma função ou outra, então poderíamos querer definir `mean()` em um módulo `"./stats/mean.js"` e definir `stddev()` em `"./stats/stddev.js"`. Dessa forma, os programas só precisam importar exatamente as funções de que precisam e não são inchados pela importação de código de que não precisam.

Mesmo se tivéssemos definido essas funções estatísticas em módulos individuais, no entanto, poderíamos esperar que houvesse muitos programas que desejam ambas as funções e apreciariam um módulo conveniente `"./stats.js"` do qual eles poderiam importar ambos em uma linha.

Dado que as implementações agora estão em arquivos separados, definir este módulo `"./stat.js"` é simples:

```
import { mean } from "./stats/mean.js";
import { stddev } from "./stats/stddev.js";
export { significa, stdev };
```

Os módulos ES6 antecipam esse caso de uso e fornecem uma sintaxe especial para isso. Em vez de importar um símbolo simplesmente para exportá-lo novamente, você pode combinar as etapas de importação e exportação em uma única instrução "reexportar" que usa a palavra-chave `export` e a palavra-chave `from`:

```
export { mean } de "./stats/mean.js";
export { stddev } from "./stats/stddev.js";
```

Observe que os nomes significam e stddev não são realmente usados neste código. Se não estivermos sendo seletivos com uma reexportação e simplesmente quisermos exportar todos os valores nomeados de outro módulo, podemos usar um curinga:

```
export * from "./stats/mean.js";
export * from "./stats/stddev.js";
```

A sintaxe de reexportação permite renomear com as instruções de importação e exportação regulares. Suponha que quiséssemos reexportar a função mean(), mas também definir average() como outro nome para a função. Poderíamos fazer isso assim:

```
export { mean, mean as average } from "./stats/mean.js";
export { stddev } from "./stats/stddev.js";
```

Todas as reexportações neste exemplo pressupõem que os módulos "./stats/mean.js" e "./stats/stddev.js" exportam suas funções usando exportarem vez de export default. Na verdade, porém, uma vez que se trata de módulos com apenas uma única exportação, teria feito sentido defini-los com padrão de exportação. Se tivéssemos feito isso, a sintaxe de reexportação seria um pouco mais complicada porque precisa definir um nome para as exportações padrão sem nome. Podemos fazer isso assim:

```
export { default as mean } from "./stats/mean.js";
export { default as stddev } from "./stats/stddev.js";
```

Se você deseja reexportar um símbolo nomeado de outro módulo como a exportação padrão do seu módulo, você pode fazer uma importação seguida por um padrão de exportação ou pode combinar as duas instruções como

este:

```
Importe a função mean() de ./stats.js e torne-a a//  
exportação padrão deste moduleexport { mean as default }  
from "./stats.js"
```

E, finalmente, para reexportar a exportação padrão de outro módulo como a exportação padrão do seu módulo (embora não esteja claro por que você gostaria de fazer isso, já que os usuários podem simplesmente importar o outro módulo diretamente), você pode escrever:

```
O módulo average.js simplesmente reexporta o  
stats/mean.jsdefault exportexport { default } de  
"./stats/mean.js"
```

10.3.5 Módulos JavaScript na Web

As seções anteriores descreveram os módulos ES6 e suas declarações de importação e exportação de uma maneira um tanto abstrata. Nesta seção e na próxima, discutiremos como eles realmente funcionam em navegadores da web e, se você ainda não for um desenvolvedor web experiente, poderá achar o restante deste capítulo mais fácil de entender depois de ler o Capítulo 15.

No início de 2020, o código de produção usando módulos ES6 ainda era geralmente empacotado com uma ferramenta como o webpack. Existem compensações em fazer isso,¹ mas, no geral, o pacote de código tende a oferecer melhor desempenho. Isso pode mudar no futuro, à medida que as velocidades da rede aumentam e os fornecedores de navegadores continuam a otimizar suas implementações de módulos ES6.

Embora as ferramentas de empacotamento ainda possam ser desejáveis na produção, elas não são mais necessárias no desenvolvimento, pois todos os navegadores atuais

fornecer suporte nativo para módulos JavaScript. Lembre-se de que os módulos usammodo estrito por padrão, isso não se refere a um objeto global e as declarações de nível superior não são compartilhadas globalmente por padrão. Como os módulos devem ser executados de forma diferente do código legado que não é módulo, sua introdução requer alterações no HTML e no JavaScript. Se você quiser usar diretivas de importação nativamente em um navegador da web, deverá informar ao navegador da web que seu código é um módulo usando uma tag <scripttype="module">.

Um dos recursos interessantes dos módulos ES6 é que cada módulo possui um conjunto estático de importações. Portanto, dado um único módulo inicial, um navegador da Web pode carregar todos os seus módulos importados e, em seguida, carregar todos os módulos importados por esse primeiro lote de módulos e assim por diante, até que um programa completo seja carregado. Vimos que o especificador de módulo em uma instrução import pode ser tratado como uma URL relativa. Uma tag <scripttype="module"> marca o ponto de partida de um programa modular. No entanto, espera-se que nenhum dos módulos importados esteja em<script> tags: em vez disso, eles são carregados sob demanda como arquivos JavaScript regulares e são executados em modo estrito como módulos ES6 regulares. Usar uma tag <script type="module"> para definir o ponto de entrada principal para um programa JavaScript modular pode ser tão simples quanto isso:

```
<tipo de script="módulo">import "./main.js";</script>
```

O código dentro de uma tag inline <script type="module"> é um ES6module e, como tal, pode usar a instrução de exportação. Não há nenhum ponto em fazê-lo, no entanto, porque a <script> sintaxe da tag HTML

não fornece nenhuma maneira de definir um nome para módulos embutidos, portanto, mesmo que tal módulo exporte um valor, não há como outro módulo importá-lo.

Scripts com o atributo `type="module"` são carregados e executados como scripts com o atributo `defer`. O carregamento do código começa assim que o analisador HTML encontra a `<script>` tag (no caso de módulos, essa etapa de carregamento de código pode ser um processo recursivo que carrega vários arquivos JavaScript). Mas a execução do código não começa até que a análise HTML seja concluída. E uma vez concluída a análise de HTML, os scripts (modulares e não) são executados na ordem em que aparecem no documento HTML.

Você pode modificar o tempo de execução dos módulos com o atributo `async`, que funciona da mesma forma para os módulos que funciona para scripts regulares. Um módulo assíncrono será executado assim que o código for carregado, mesmo que a análise de HTML não esteja concluída e mesmo que isso altere a ordem relativa dos scripts.

Os navegadores da Web que suportam `<script type="module">` também devem suportar `<script nomodule>`. Os navegadores que reconhecem o módulo ignoram qualquer script com o atributo `nomodule` e não o executarão. Os navegadores que não suportam módulos não reconhecerão o atributo `nomodule`, portanto, eles o ignorarão e executarão o script. Isso fornece uma técnica poderosa para lidar com problemas de compatibilidade do navegador. Os navegadores que suportam módulos ES6 também suportam outros recursos modernos do JavaScript, como classes, funções de seta e o loop `for/of`. Se você escrever JavaScript moderno e carregá-lo com `<script`

`type="module">`, você sabe que ele só será carregado por navegadores que podem suportá-lo. E como um fallback para o IE11 (que, em 2020, é efetivamente o único navegador restante que não suporta ES6), você pode usar ferramentas como Babel e webpack para transformar seu código em código ES5 não modular e, em seguida, carregar esse código transformado menos eficiente por meio `<script nomodule>`.

Outra diferença importante entre scripts regulares e modulescripts tem a ver com o carregamento de origem cruzada. Uma `<script>` tag regular carregará um arquivo de código JavaScript de qualquer servidor na Internet, e a infraestrutura de publicidade, análise e código de rastreamento da Internet depende desse fato. Mas `<script type="module">` oferece uma oportunidade de restringir isso, e os módulos só podem ser carregados da mesma origem que o documento HTML que o contém ou quando os cabeçalhos CORS adequados estão em vigor para permitir cargas de origem cruzada com segurança. Um efeito colateral infeliz dessa nova restrição de segurança é que ela dificulta o teste de módulos ES6 no modo de desenvolvimento usando file:URLs. Ao usar módulos ES6, você provavelmente precisará configurar um servidor staticweb para teste.

Alguns programadores gostam de usar a extensão de nome de arquivo `.mjs` para distinguir seus arquivos JavaScript modulares de seus arquivos JavaScript regulares e não modulares com a extensão `.js` tradicional. Para fins de navegadores da web e `<script>` tags, a extensão do arquivo é realmente irrelevante. (O tipo MIME é relevante, no entanto, portanto, se você usar arquivos `.mjs`, talvez seja necessário configurar seu servidor Web para servi-los com o mesmo tipo MIME que `.js` arquivos.) O suporte do Node para ES6 usa a extensão do nome do arquivo como uma dica para distinguir qual módulo

sistema é usado por cada arquivo que carrega. Portanto, se você estiver escrevendo módulos ES6 e quiser que eles sejam utilizáveis com o Node, pode ser útil adotar a convenção de nomenclatura .mjs.

10.3.6 Importações dinâmicas com import()

Vimos que as diretivas de importação e exportação do ES6 são completamente estáticas e permitem que interpretadores de JavaScript e outras ferramentas JavaScript determinem as relações entre os módulos com análise de texto simples enquanto os módulos estão sendo carregados sem ter que realmente executar nenhum código nos módulos. Com módulos importados estaticamente, você tem a garantia de que os valores importados para um módulo estarão prontos para uso antes que qualquer código em seu módulo comece a ser executado.

Na web, o código deve ser transferido por uma rede em vez de ser lido do sistema de arquivos. E uma vez transferido, esse código é frequentemente executado em dispositivos móveis com CPUs relativamente lentas. Este não é o tipo de ambiente em que as importações de módulos estáticos - que exigem que um programa inteiro seja carregado antes de qualquer um deles ser executado - fazem muito sentido.

É comum que os aplicativos da Web carreguem inicialmente apenas o suficiente de seu código para renderizar a primeira página exibida ao usuário. Então, uma vez que o usuário tenha algum conteúdo preliminar para interagir, ele pode começar a carregar a quantidade geralmente muito maior de código necessária para o restante do aplicativo da web. Os navegadores da Web facilitam o carregamento dinâmico do código usando a API DOM para injetar uma nova <script> tag no documento HTML atual, e os aplicativos da Web fazem isso há muitos anos.

Embora o carregamento dinâmico seja possível há muito tempo, ele não faz parte da linguagem em si. Isso muda com a introdução de import() no ES2020 (a partir do início de 2020, a importação dinâmica é suportada por todos os navegadores que suportam módulos ES6). Você passa um especificador de módulo para import() e ele retorna um objeto Promise que representa o processo assíncrono de carregamento e execução do módulo especificado. Quando a importação dinâmica é concluída, a Promise é "cumprida" (consulte o Capítulo 13 para obter detalhes completos sobre programação assíncrona e Promises) e produz um objeto como o que você obteria com a importação * como forma da instrução de importação estática.

Então, em vez de importar o módulo "./stats.js" estaticamente, assim:

```
import * como estatísticas de "./stats.js";
```

Podemos importá-lo e usá-lo dinamicamente, assim:

```
import("./stats.js").then(stats => {
  let média = stats.mean(dados);})
```

Ou, em uma função assíncrona (novamente, talvez seja necessário ler o Capítulo 13 antes de entender esse código), podemos simplificar o código withawait:

```
analisaDados(dados) assíncronos {
  let stats = await
    import("./stats.js"); retornar {
  média: stats.mean(dados), stddev:
  stats.stddev(dados)};}
```

O argumento para import() deve ser um especificador de módulo, exatamente como aquele que você usaria com uma diretiva de importação estática. Mas com import(), você não está restrito a usar um literal de string constante: qualquer expressão que seja avaliada como uma string na forma adequada servirá.

Dynamic import() parece uma invocação de função, mas na verdade não é. Em vez disso, import() é um operador e os parênteses são uma parte obrigatória da sintaxe do operador. A razão para essa sintaxe incomum é que import() precisa ser capaz de resolver especificadores de módulo como URLs em relação ao módulo em execução no momento, e isso requer um pouco de mágica de implementação que não seria legal colocar em uma função JavaScript. A distinção entre função e operador raramente faz diferença na prática, mas você notará isso se tentar escrever código como console.log(import); ou seja require =import;.

Por fim, observe que o dynamic import() não é apenas para navegadores da web. Ferramentas de empacotamento de código, como o webpack, também podem fazer bom uso dele. A maneira mais direta de usar um empacotador de código é informar a ele o ponto de entrada principal do seu programa e deixá-lo encontrar todas as diretivas de importação estáticas e montar tudo em um arquivo grande. No entanto, usando estrategicamente chamadas dinâmicas import(), você pode dividir esse pacote monolítico em um conjunto de pacotes menores que podem ser carregados sob demanda.

10.3.7 import.meta.url

Há um recurso final do sistema de módulos ES6 a ser discutido. Dentro de um módulo ES6 (mas não dentro <script> de um módulo regular ou de nó

carregado com require()), a sintaxe especial import.meta refere-se a um objeto que contém metadados sobre o módulo em execução no momento. A propriedade url desse objeto é a URL da qual o módulo foi carregado. (No Node, essa será uma URL file://.)

O principal caso de uso de import.meta.url é ser capaz de se referir a imagens, arquivos de dados ou outros recursos armazenados no mesmo diretório que (ou em relação a) o módulo. O construtor URL() facilita a resolução de um URL relativo em relação a um URL absoluto como import.meta.url. Suponha, por exemplo, que você tenha escrito um módulo que inclui strings que precisam ser localizadas e que os arquivos de localização são armazenados em um diretório l10n/, que está no mesmo diretório que o próprio módulo. Seu módulo pode carregar suas strings usando uma URL criada com uma função, como esta:

```
function localStringsURL(locale) {  
    return new URL('l10n/${locale}.json', import.meta.url);}
```

10.4 Resumo

O objetivo da modularidade é permitir que os programadores ocultem os detalhes de implementação de seu código para que pedaços de código de várias fontes possam ser montados em grandes programas sem se preocupar que um pedaço substitua funções ou variáveis de outro. Este capítulo explicou três sistemas de módulos JavaScript diferentes:

- Nos primórdios do JavaScript, a modularidade só podia ser alcançada através do uso inteligente de

- expressões de função. O Node adicionou seu próprio sistema de módulos sobre a linguagem JavaScript. Os módulos de nó são importados com require() e definem suas exportações definindo propriedades do objeto Exports ou definindo a propriedade module.exports. No ES6, o JavaScript finalmente ganhou seu próprio sistema de módulos compalavras-chave de importação e exportação, e o ES2020 está adicionando suporte para importações dinâmicas com import().

1 Por exemplo: aplicativos da web que têm atualizações incrementais frequentes e usuários que fazem visitas de retorno frequentes podem descobrir que o uso de módulos pequenos em vez de pacotes grandes pode resultar em melhores tempos médios de carregamento devido à melhor utilização do browsercache do usuário.

Capítulo 11. A Biblioteca JavaScriptStandard

Alguns tipos de dados, como números e strings (Capítulo 3), objetos (Capítulo 6) e matrizes (Capítulo 7) são tão fundamentais para o JavaScript que podemos considerá-los parte da própria linguagem. Este capítulo cobre outras APIs importantes, mas menos fundamentais, que podem ser consideradas como definindo a "biblioteca padrão" para JavaScript: essas são classes e funções úteis que são incorporadas ao JavaScript e disponíveis para todos os programas JavaScript em navegadores da web e no Node. [1](#)

As seções deste capítulo são independentes umas das outras e você pode lê-las em qualquer ordem. Eles cobrem:

- As classes Set e Map para representar conjuntos de valores e mapeamentos de um conjunto de valores para outro conjunto
- de valores. Objetos semelhantes a arrays conhecidos como TypedArrays que representam arrays de dados binários, juntamente com uma classe relacionada para extrair valores de dados binários que não são de array. Expressões regulares e a classe RegExp, que definem padrões textuais e são úteis para processamento de texto. Esta seção também aborda a sintaxe de expressão regular em detalhes. A classe Date para representar e manipular datas e horas. A classe Error e suas várias subclasses, cujas instâncias são lançadas quando ocorrem erros em programas JavaScript.

O objeto JSON, cujos métodos suportam serialização e desserialização de estruturas de dados JavaScript compostas por objetos, matrizes, strings, números e booleanos. O objeto Intl e as classes que ele define podem ajudá-lo a localizar seus programas JavaScript. O objeto Console, cujos métodos geram cadeias de caracteres de maneiras que são particularmente úteis para depurar programas e registrar o comportamento desses programas. A classe URL, que simplifica a tarefa de analisar e manipular URLs. Esta seção também aborda funções globais para codificação e decodificação de URLs e suas partes componentes.setTimeout() e funções relacionadas para especificar o código a ser executado após um intervalo de tempo especificado. Algumas das seções deste capítulo - notavelmente, as seções sobre typedarrays e expressões regulares - são bastante longas porque há informações básicas significativas que você precisa entender antes de poder usar esses tipos de forma eficaz. Muitas das outras seções, no entanto, são curtas: limitam-se a introduzir uma nova API e a mostrar alguns exemplos da sua utilização.

11.1 Conjuntos e Mapas

O tipo de objeto do JavaScript é uma estrutura de dados versátil que pode ser usada para mapear strings (os nomes de propriedade do objeto) para valores arbitrários. E quando o valor que está sendo mapeado é algo fixo como true, então o objeto é efetivamente um conjunto de strings.

Os objetos são realmente usados como mapas e conjuntos rotineiramente em JavaScript

programação, mas isso é limitado pela restrição a strings e complicado pelo fato de que os objetos normalmente herdam propriedades com nomes como "toString", que normalmente não se destinam a fazer parte domapa ou conjunto.

Por esse motivo, o ES6 apresenta as verdadeiras classes Set e Map, que abordaremos nas subseções a seguir.

11.1.1 A classe do conjunto

Um conjunto é uma coleção de valores, como uma matriz. Ao contrário das matrizes, no entanto, os conjuntos não são ordenados ou indexados e não permitem duplicatas: avalue é um membro de um conjunto ou não é um membro; Não é possível perguntar quantas vezes um valor aparece em um conjunto.

Crie um objeto Set com o construtor Set():

```
let s = novo Set();           Um novo conjunto vazio
let t = new Set([1, s]); Um novo conjunto com dois membros
```

O argumento para o construtor Set() não precisa ser um array: qualquer objeto iterável (incluindo outros objetos Set) é permitido:

```
let t = novo(s) conjunto(s);           Um novo conjunto que copia
os elementos de s.let unique = new Set("Mississippi"); 4
elementos: "M", "i", "s" e "p"
```

A propriedade size de um conjunto é como a propriedade length de um array: ela informa quantos valores o conjunto contém:

```
único.tamanho      => 4
```

Os conjuntos não precisam ser inicializados quando você os cria. Você pode adicionar e remover elementos a qualquer momento com add(), delete() e clear(). Lembre-se de que os conjuntos não podem conter duplicatas, portanto, adicionar um valor a um conjunto quando ele já contém esse valor não tem efeito:

```
let s = novo Set(); Começar do zero
s.size           => 0
s.add(1);       Adicionar um número
s.size           => 1; Agora o conjunto tem um membro
s.add(1);       Adicione o mesmo número novamente
s.size           => 1; o tamanho não muda
s.add(verdadeiro); Adicione outro valor; Observe que é
multa para misturar
tysize.size     => 2
s.add([1,2,3]); Adicionar um valor de matriz
s.size           => 3; a matriz foi adicionada, não sua
elementos
.s.delete(1)     => true: elemento excluído com sucesso
1s.
tamanho         => 2: o tamanho volta a ser 2
s.delete("teste") => false: "teste" não era um membro,
falha na
exclusão.delete(true) => true: exclusão bem-sucedida
s.delete([1,2,3]) => false: o array no conjunto é
diferen
ts.size          => 1: ainda há aquele array em
0
set.clear();      Remova tudo do conjunto
s.size           => 0
```

Há alguns pontos importantes a serem observados sobre este código:

- O método add() recebe um único argumento; Se você passar AnArray, ele adicionará o próprio array ao conjunto, não os arrayElements individuais. add() sempre retorna o conjunto em que é invocado, no entanto, se você quiser adicionar vários valores a um conjunto, você pode usar chamadas de método encadeadas como

- s.add('a').add('b').add('c');. O método delete() também exclui apenas um único elemento definido por vez. Ao contrário de add(), no entanto, delete() retorna um valor booleano. Se o valor especificado for realmente um membro do conjunto, delete() o removerá e retornará true. Caso contrário, ele não faz nada e retorna false. Finalmente, é muito importante entender que a associação de conjunto é baseada em verificações de igualdade estritas, como o operador ===. Um conjunto pode conter o número 1 e a cadeia de caracteres "1", porque os considera valores distintos. Quando os valores são objetos (ou matrizes ou funções), eles também são comparados como se fossem com ===. É por isso que não conseguimos excluir o elemento array do conjunto neste código. Adicionamos um array ao conjunto e, em seguida, tentamos remover esse array passando um array diferente (embora com os mesmos elementos) para o método delete(). Para que isso funcionasse, teríamos que passar uma referência exatamente para o mesmo array.

NOTA

Programadores Python, tomem nota: esta é uma diferença significativa entre JavaScript e conjuntos Python. Os conjuntos do Python comparam membros por igualdade, não por identidade, mas a desvantagem é que os conjuntos do Python permitem apenas membros imutáveis, como tuplas, e não permitem que listas e dicts sejam adicionados aos conjuntos.

Na prática, a coisa mais importante que fazemos com conjuntos não é adicionar e remover elementos deles, mas verificar se um valor especificado é um membro do conjunto. Fazemos isso com o método has():

```
let oneDigitPrimes = new Set([2,3,5,7]);
oneDigitPrimes.has(2)// => true: 2 é um primo de um dígito
```

```
número ou um DigitPrimes.  
tem(3)                      => verdadeiro: assim é 3  
oneDigitPrimes.has(4)          => falso: 4 não é um primo  
oneDigitPrimes.has("5")        => falso: "5" não é nem mesmo um  
número
```

A coisa mais importante a entender sobre conjuntos é que eles são otimizados para testes de associação, e não importa quantos membros o conjunto tenha, o método has() será muito rápido. O includes() método de um array também executa testes de associação, mas o tempo que leva é proporcional ao tamanho do array, e usar um array como um conjunto pode ser muito, muito mais lento do que usar um objeto Set real.

A classe Set é iterável, o que significa que você pode usar um loop for/of para enumerar todos os elementos de um conjunto:

```
seja soma = 0; for(let p of oneDigitPrimes) { // Percorre  
os primos de um dígito
```

```
soma += p;                      e somá-los  
} so  
ma                                => 17: 2 + 3 + 5 + 7
```

Como os objetos Set são iteráveis, você pode convertê-los em arrays e listas de argumentos com o ... Operador de propagação:

```
[... oneDigitPrimes]      => [2,3,5,7]: o conjunto  
convertido em um ArrayMath.max (... oneDigitPrimes) // =>  
7: definir elementos passados como argumentos de função
```

Os conjuntos são frequentemente descritos como "coleções não ordenadas". No entanto, isso não é exatamente verdadeiro para a classe JavaScript Set. Um conjunto JavaScript não é indexado: você não pode solicitar o primeiro ou o terceiro elemento de um conjunto da maneira

você pode com uma matriz. Mas a classe JavaScript Set sempre se lembra da ordem em que os elementos foram inseridos e sempre usa essa ordem quando você itera um conjunto: o primeiro elemento inserido será o primeiro iterado (supondo que você não o tenha excluído primeiro) e o elemento inserido mais recentemente será o último iterado.

Além de ser iterável, a classe Set também implementa o método `forEach()` que é semelhante ao método array do mesmo nome:

```
deixe produto = 1; oneDigitPrimes.forEach(n =>
{ produto *= n; });
produto// => 210: 2 * 3 *
5 * 7
```

O `forEach()` de um array passa índices de array como o `secondargument` para a função que você especificar. Os conjuntos não têm índices, então a versão da classe Set desse método simplesmente passa o valor do elemento como o primeiro e o segundo argumento.

11.1.2 A classe do mapa

Um objeto Map representa um conjunto de valores conhecidos como chaves, em que cada chave tem outro valor associado a (ou "mapeado para"). Em certo sentido, um mapa é como uma matriz, mas em vez de usar um conjunto de inteiros sequenciais como chaves, os mapas nos permitem usar valores arbitrários como "índices". Como arrays, os mapas são rápidos: procurar o valor associado a uma chave será rápido (embora não tão rápido quanto indexar um array), não importa o tamanho do mapis.

Crie um novo mapa com o construtor `Map()`:

```
let m = novo Mapa(); Criar um novo mapa vazio
let n = novo Mapa([ Um novo mapa inicializado com chaves de string
mapeado para números
["um", 1],
["dois", 2]]);
```

O argumento opcional para o construtor Map() deve ser um objeto iterável que produz duas matrizes de elementos [chave, valor]. Na prática, isso significa que, se você quiser inicializar um mapa ao criá-lo, normalmente escreverá as chaves desejadas e os valores associados como uma matriz de matrizes. Mas você também pode usar o construtor Map() para copiar outros mapas ou copiar os nomes e valores das propriedades de um objeto existente:

```
let copy = new Map(n); Um novo mapa com as mesmas chaves e
valores que map nlet o = { x: 1, y: 2}; Um objeto com duas
propriedadeslet p = new Map(Object.entries(o)); O mesmo que
new map([[x",1], ["y", 2]])
```

Depois de criar um objeto Map, você pode consultar o valor associado a uma determinada chave com get() e adicionar um novo par chave/valor com set(). Lembre-se, porém, de que um mapa é um conjunto de chaves, cada uma com um valor associado. Isso não é exatamente o mesmo que um conjunto de pares chave/valor. Se você chamar set() com uma chave que já existe no themap, você alterará o valor associado a essa chave, não adicionará um mapeamento de newkey/value. Além de get() e set(), a classe Map também define métodos que são como métodos Set: use has() para verificar se um mapa inclui a chave especificada; use delete() para remover akey (e seu valor associado) do mapa; use clear() para remover

todos os pares de chave/valor do mapa; e use a propriedade size para descobrir quantas chaves um mapa contém.

```
let m = novo Mapa(); Comece com um mapa vazio
m.size                  => 0: mapas vazios não têm chaves
m.set("um", 1);          Mapeie a chave "um" para o valor 1
m.set("dois", 2);        E a chave "dois" para o valor 2.
m.size                  => 2: o mapa agora tem duas chaves
m.get("dois")           => 2: retorna o valor associado
com a chave
"dois".m.get("três")    => indefinido: esta chave não está no
set
m.set("um",             Alterar o valor associado a um
verdadeiro);
keym.size               => 2: o tamanho não muda
existente              => true: o mapa tem uma chave "um"
m.has("um")             => false: o mapa não tem uma chave
m.has(verdadeiro)       => true: a chave existia e exclusão
sucede
dm.size                => 1
m.delete("três")        => false: falha ao excluir um
keym.clear()            Remova todas as chaves e valores do
mapa
```

Como o método add() de Set, o método set() de Map pode ser encadeado, o que permite que os mapas sejam inicializados sem o uso de arrays de arrays:

```
let m = new Map().set("um", 1).set("dois",
2).set("três",3); m.size// => 3m.get("dois")// => 2
```

Assim como em Set, qualquer valor JavaScript pode ser usado como uma chave ou um valor no aMap. Isso inclui nulo, indefinido e NaN, bem como referência

tipos como objetos e matrizes. E como acontece com a classe Set, Map compara keys por identidade, não por igualdade, portanto, se você usar um objeto ou array como akey, ele será considerado diferente de todos os outros objetos e arrays, mesmo aqueles com exatamente as mesmas propriedades ou elementos:

```
let m = novo Mapa(); Comece com um mapa vazio.  
m.set({}, 1); Mapeie um objeto vazio para o número 1.  
m.set({}, 2); Mapeie um objeto vazio diferente para o  
número  
2.m.size => 2: existem duas chaves neste mapa  
m.get({}) => indefinido: mas este objeto vazio  
não é um keym.set(m, indefinido); Mapeie o próprio mapa  
para o valueundefined.m.has(m)// => true: m é uma chave em  
sim.get(m)// => undefined: mesmo valor que obteríamos if m  
não era uma chave
```

Os objetos de mapa são iteráveis e cada valor iterado é uma matriz de dois elementos onde o primeiro elemento é uma chave e o segundo elemento é o valor associado a essa chave. Se você usar o operador spread com um Map object, obterá uma matriz de arrays como os que passamos para o construtor theMap(). E ao iterar um mapa com um loop for/of, é idiomático usar a atribuição de desestruturação para atribuir a chave e o valor a variáveis separadas:

```
let m = new Map([["x", 1], ["y", 2]]);  
[... m]// => [[{"x": 1}, {"y": 2}]]  
  
for(let [chave, valor] de m) {  
  Na primeira iteração, a chave será "x" e o valor será 1  
  
  Na segunda iteração, a chave será "y" e o valor será 2}
```

Assim como a classe Set, a classe Map itera na ordem de inserção. O par firstkey/value iterado será o menos adicionado recentemente ao mapa, e o último par iterado será o adicionado mais recentemente.

Se você quiser iterar apenas as chaves ou apenas os valores associados de um map, use os métodos keys() e values(): eles retornam iterableobjects que iteram chaves e valores, em ordem de inserção. (O método entries() retorna um objeto iterável que itera pares chave/valor, mas isso é exatamente o mesmo que iterar o mapa diretamente.)

```
[... m.keys()] => ["x", "y"]; apenas as teclas  
[... m.values()] => [1, 2]; apenas os valores  
[... m.entries()] => [{"x": 1}, {"y": 2}]; o mesmo que [... m]
```

Os objetos de mapa também podem ser iterados usando o método forEach() que foi implementado pela primeira vez pela classe Array.

```
m.forEach((value, key) => {// valor da nota, chave NÃO  
chave, valor// Na primeira invocação, o valor será 1 e a  
chave será "x"
```

Na segunda invocação, o valor será 2 e a chave será "y"});

Pode parecer estranho que o parâmetro value venha antes do keyparameter no código acima, já que com a iteração for/of, o keyvem primeiro. Conforme observado no início desta seção, você pode pensar em um mapa uma matriz generalizada na qual os índices de matriz inteira são substituídos por valores de chave arbitrários. O método forEach() de arrays passa o elemento array primeiro e o índice do array em segundo, então, por analogia, o método forEach() de um mapa passa o valor do mapa primeiro e o mapa

chave em segundo.

11.1.3 WeakMap e WeakSet

A classe WeakMap é uma variante (mas não uma subclasse real) da classe Map que não impede que seus valores de chave sejam coletados como lixo. A coleta de lixo é o processo pelo qual o interpretador JavaScript recupera a memória de objetos que não são mais "acessíveis" e não podem ser usados pelo programa. Um mapa regular contém referências "fortes" aos seus valores-chave, e eles permanecem acessíveis por meio deles, mesmo que todas as outras referências a eles tenham desaparecido. O WeakMap, por outro lado, mantém referências "fracas" a seus valores-chave para que eles não sejam acessíveis por meio do WeakMap, e sua presença no mapa não impede que sua memória seja recuperada.

O construtor WeakMap() é exatamente como o construtor Map(), mas há algumas diferenças significativas entre WeakMap e Map:

As chaves WeakMap devem ser objetos ou matrizes; Os valores primitivos não estão sujeitos à coleta de lixo e não podem ser usados como chaves. O WeakMap implementa apenas os métodos get(), set(), has() e delete(). Em particular, WeakMap não é iterável e não define keys(), values() ou forEach(). Se WeakMap fosse iterável, suas chaves seriam acessíveis e não seriam fracas. Da mesma forma, WeakMap não implementa a propriedade size porque o tamanho de um WeakMap pode mudar a qualquer momento à medida que os objetos são coletados como lixo. O uso pretendido de WeakMap é permitir que você associe valores a

objetos sem causar vazamentos de memória. Suponha, por exemplo, que você esteja escrevendo uma função que recebe um argumento de objeto e precisa executar algum cálculo demorado nesse objeto. Para eficiência, você gostaria de armazenar em cache o valor calculado para reutilização posterior. Se você usar um objeto Map para implementar o cache, impedirá que qualquer um dos objetos seja recuperado, mas usando um WeakMap, você evitará esse problema. (Muitas vezes, você pode obter um resultado semelhante usando uma propriedade Symbol privada para armazenar em cache o valor calculado diretamente no objeto. Consulte §6.10.3.)

WeakSet implementa um conjunto de objetos que não impede que esses objetos sejam coletados como lixo. O construtor WeakSet() funciona como o construtor Set(), mas os objetos WeakSet diferem dos objetos Setobjects da mesma forma que os objetos WeakMap diferem dos objetos Map:

WeakSet não permite valores primitivos como membros. WeakSet implementa apenas os métodos add(), has() e delete() e não é iterável. WeakSet não tem uma propriedade size. WeakSet não é usado com frequê~~n~~cia: seus casos de uso são como os de WeakMap. Se você quiser marcar (ou "marcar") um objeto como tendo alguma propriedade ou tipo especial, por exemplo, você pode adicioná-lo a um WeakSet. Então, em outro lugar, quando quiser verificar essa propriedade ou tipo, você pode testar a associação a esse WeakSet. Fazer isso com um conjunto regular impediria que todos os objetos marcados fossem coletados como lixo, mas isso não é uma preocupação ao usar o WeakSet.

11.2 Matrizes tipadas e dados binários

Arrays JavaScript regulares podem ter elementos de qualquer tipo e podem aumentar ou diminuir dinamicamente. As implementações de JavaScript executam muitas otimizações para que os usos típicos de arrays JavaScript sejam muito rápidos. No entanto, eles ainda são bem diferentes dos tipos de array de linguagens de nível de flor, como C e Java. As matrizes tipadas, que são novas no ES6, estão muito mais próximas das matrizes de baixo nível dessas linguagens. Arrays tipados não são tecnicamente arrays (`Array.isArray()` retorna `false` para eles), mas implementam todos os métodos de array descritos em §7.8 mais alguns próprios. Eles diferem dos arrays regulares de algumas maneiras muito importantes, no entanto:

- Os elementos de uma matriz tipada são todos números. Ao contrário dos números JavaScript regulares, no entanto, as matrizes tipadas permitem que você especifique o tipo (inteiros com e sem sinal e ponto flutuante IEEE-754) e o tamanho (8 bits a 64 bits) dos números a serem armazenados
- na matriz. Você deve especificar o comprimento de uma matriz tipada ao criá-la, e esse comprimento nunca pode ser alterado. Os elementos de uma matriz tipada são sempre inicializados como 0 quando a matriz é criada.

11.2.1 Tipos de matriz tipada

O JavaScript não define uma classe `TypedArray`. Em vez disso, existem 11 tipos de arrays tipados, cada um com um tipo de elemento diferente e construtor:

Construtor	Tipo numérico
------------	---------------

<code>Int8Array()</code>	bytes assinados
<code>Uint8Array()</code>	bytes não assinados
<code>Uint8ClampedArray()</code>	bytes não assinados sem rollover
<code>Int16Array()</code>	inteiros curtos de 16 bits assinados
<code>Uint16Array()</code>	inteiros curtos de 16 bits sem sinal
<code>Int32Array()</code>	inteiros de 32 bits assinados
<code>Uint32Array()</code>	inteiros de 32 bits sem sinal
<code>BigInt64Array()</code>	valores BigInt de 64 bits assinados (ES2020)
<code>BigUint64Array()</code>	valores BigInt de 64 bits não assinados (ES2020)
<code>Float32Array()</code>	Valor de ponto flutuante de 32 bits
<code>Float64Array()</code>	Valor de ponto flutuante de 64 bits: um número JavaScript regular

Os tipos cujos nomes começam com Int contêm inteiros assinados, de 1, 2 ou 4 bytes (8, 16 ou 32 bits). Os tipos cujos nomes começam com inteiros sem sinal Uinhold desses mesmos comprimentos. Os tipos "BigInt" e "BigUint" contêm inteiros de 64 bits, representados em JavaScript como valores BigInt (consulte §3.2.5). Os tipos que começam com Float holdnúmeros de ponto flutuante. Os elementos de um `Float64Array` são do mesmo tipo que os números JavaScript regulares. Os elementos de `aFloat32Array` têm menor precisão e um alcance menor, mas requerem apenas metade da memória. (Esse tipo é chamado de float em C e Java.)

`Uint8ClampedArray` é uma variante de caso especial em `Uint8Array`. Ambos os tipos contêm bytes sem sinal e podem representar números

entre 0 e 255. Com Uint8Array, se você armazenar um valor maior que 255 ou menor que zero em um elemento de matriz, ele "se enrola" e você obtém algum outro valor. É assim que a memória do computador funciona em um nível baixo, então isso é muito rápido. Uint8ClampedArray faz alguma verificação de tipo extra para que, se você armazenar um valor maior que 255 ou menor que 0, ele "fixe" em 255 ou 0 e não envolva. (Esse comportamento de fixação é exigido pela <canvas> API de nível lento do elemento HTML para manipular cores de pixel.)

Cada um dos construtores de matriz tipada tem um BYTES_PER_ELEMENT property com o valor 1, 2, 4 ou 8, dependendo do tipo.

11.2.2 Criando matrizes tipadas

A maneira mais simples de criar uma matriz tipada é chamar o construtor apropriado com um argumento numérico que especifica o número de elementos que você deseja na matriz:

```
let bytes = new Uint8Array(1024);      1024 bytes
let matrix = new Float64Array(9);       Uma matriz 3x3
let ponto = new Int16Array(3);         Um ponto no espaço 3D
let rgba = new Uint8ClampedArray(4);   Um sudoku
pixelvaluelet RGBA de 4 bytes = new Int8Array(81); // Uma
placa de sudoku 9x9
```

Quando você cria matrizes tipadas dessa maneira, todos os elementos da matriz são garantidos para serem inicializados como 0, 0n ou 0,0. Mas se você souber os valores que deseja em sua matriz digitada, também poderá especificar esses valores ao criar a matriz. Cada um dos construtores de array tipado tem métodos de fábrica static from() e of() que funcionam como Array.from() e Array.of():

```
deixe branco = Uint8ClampedArray.of(255, 255, 255, 0);      RGBA  
branco opaco
```

Lembre-se de que o método de fábrica Array.from() espera um objeto iterável semelhante a um array como seu primeiro argumento. O mesmo vale para as variantes de array tipado, exceto que o objeto iterável ou semelhante a um array também deve ter elementos numéricos. As strings são iteráveis, por exemplo, mas não faria sentido passá-las para o método de fábrica from() de um array tipado.

Se você estiver usando apenas a versão de um argumento de from(), você pode descartar o .from e passar seu objeto iterável ou semelhante a um array diretamente para a função construtora, que se comporta exatamente da mesma forma. Observe quetanto o construtor quanto o método de fábrica from() permitem que você copie arrays tipados existentes, enquanto possivelmente altera o tipo:

```
let ints = Uint32Array.from(branco); Os mesmos 4 números,  
mas como ints
```

Quando você cria uma nova matriz tipada a partir de uma matriz existente, iterável ou objeto semelhante a uma matriz, os valores podem ser truncados para se ajustarem às restrições de tipo da matriz. Não há avisos ou erros quando isso acontece:

```
Floats truncados para ints, ints mais longos truncados para  
8 bitsUint8Array.of(1.23, 2.99, 45000) // => new  
Uint8Array([1, 2.200])
```

Finalmente, há mais uma maneira de criar matrizes tipadas que envolve o Tipo ArrayBuffer. Um ArrayBuffer é uma referência opaca a um pedaço de memória. Você pode criar um com o construtor; basta passar o

Número de bytes de memória que você gostaria de alocar:

```
let buffer = new ArrayBuffer(1024*1024);
buffer.byteLength// => 1024*1024; um megabyte de memória
```

A classe ArrayBuffer não permite que você leia ou grave nenhum dos bytes alocados. Mas você pode criar matrizes tipadas que usam a memória do buffer e que permitem que você leia e escreva essa memória. Para fazer isso, chame o construtor de matriz tipada com anArrayBuffer como o primeiro argumento, um deslocamento de bytes dentro do buffer de matriz como o segundo argumento e o comprimento da matriz (em elementos, não em bytes) como o terceiro argumento. O segundo e o terceiro argumentos são opcionais. Se você omitir ambos, o array usará toda a memória no buffer do array. Se você omitir apenas o argumento length, sua matriz usará toda a memória disponível entre a posição inicial e o final da matriz. Mais uma coisa a ter em mente sobre essa forma do construtor typedarray: as matrizes devem ser alinhadas à memória, portanto, se você especificar um deslocamento de byte, o valor deverá ser um múltiplo do tamanho do seu tipo. O construtor Int32Array() requer um múltiplo de quatro, por exemplo, e o Float64Array() requer um múltiplo de oito.

Dado o ArrayBuffer criado anteriormente, você pode criar matrizes tipadas como estas:

```
let asbytes = new Uint8Array(buffer);           Visto como
byteslet asints = new
Int32Array(buffer);                          Visto como
Intslet assinado de 32 bits lastK = new
Uint8Array(buffer, 1023*1024); Lastkilobyte as
byteslet ints2 = new Int32Array(buffer, 1024, 256);
2ndkilobyte como 256 inteiros
```

Essas quatro matrizes tipadas oferecem quatro exibições diferentes na memória representada pelo ArrayBuffer. É importante entender que as matrizes alltyped têm um ArrayBuffer subjacente, mesmo que você não especifique explicitamente um. Se você chamar um construtor de matriz tipada sem ultrapassar um objeto buffer, um buffer do tamanho apropriado será criado automaticamente. Conforme descrito posteriormente, a propriedade buffer de anytyped array refere-se ao seu objeto ArrayBuffer subjacente. A razão para trabalhar diretamente com objetos ArrayBuffer é que às vezes você pode querer ter várias exibições de matriz tipadas de um único buffer.

11.2.3 Usando matrizes tipadas

Depois de criar um array tipado, você pode ler e escrever seus elementos com notação regular de colchetes, assim como faria com qualquer outro objeto semelhante a array:

Retorne o maior primo menor que n, usando a peneira de Eratóstenes função peneira(n) {

```
let a = new Uint8Array(n+1);           a[x] será 1 se
x é composto
let max = Math.floor(Math.sqrt(n));  Não faça fatores
maior do que isso
seja p = 2;                         2 é o primeiro
principal
while(p <= max) {                  Para primos menos
que o máximo
    for(let i = 2*p; i <= n; i += p) // Marca múltiplos de
p como composto
        a[i] = 1; while(a[++p])
        /* vazio */;                 O próximo não marcado
0 índice é primo
    }while(a[n]) n--;
Volte para trás para encontrar o último primo
retorno n;                           E devolvê-lo
```

```
}
```

A função aqui calcula o maior número primo menor que o número especificado. O código é exatamente o mesmo que seria com um array JavaScript regular, mas usar Uint8Array() em vez de Array() faz com que o código seja executado mais de quatro vezes mais rápido e use oito vezes menos memória em meus testes.

Arrays tipados não são arrays verdadeiros, mas eles reimplementam a maioria dos arraymethods, então você pode usá-los praticamente como usaria arrays regulares:

```
let ints = new Int16Array(10);      10 inteiros curtos
ints.fill(3).map(x=>x*x).join("") => "9999999999"
```

Lembre-se de que as matrizes tipadas têm comprimentos fixos, portanto, a propriedade lengthé somente leitura e os métodos que alteram o comprimento da matriz (como push(), pop(), unshift(), shift() e splice()) não são implementados para matrizes tipadas. Métodos que alteram o conteúdo de um array sem alterar o comprimento (como sort(), reverse() e fill()) são implementados. Métodos como map() e slice() que retornam novos arrays retornam um array tipado do mesmo tipo daquele em que são chamados.

11.2.4 Métodos e propriedades de matriz tipada

Além dos métodos de matriz padrão, as matrizes tipadas também implementam alguns métodos próprios. O método set() define vários elementos de um array tipado de uma só vez, copiando os elementos de um array regular ou tipado em um array tipado:

```
let bytes = new Uint8Array(1024);           Um buffer de 1K
let padrão = new Uint8Array([0,1,2,3]); Um array de 4bytes
bytes.set(pattern); // Copie-os para o início de anotherbyte
arraybytes.set(pattern, 4); // Copie-os novamente em um differentoffset
bytes.set([0,1,2,3], 8); Ou apenas copie os valores diretamente de um array
regularbytes.slice(0, 12) // =>
newUint8Array([0,1,2,3,0,1,2,3,0,1,2,3])
```

O método `set()` recebe um array ou array tipado como seu primeiro argumento e um deslocamento de elemento como seu segundo argumento opcional, que é padronizado para 0 se não for especificado. Se você estiver copiando valores de uma matriz digitada para outra, a operação provavelmente será extremamente rápida.

As matrizes tipadas também têm um método de submatriz que retorna uma parte da matriz na qual é chamado:

```
let ints = new Int16Array([0,1,2,3,4,5,6,7,8,9]);           // 10
inteiros curtos let último3 =
ints.subarray(ints.length-3, ints.length);                  // Últimos 3
delesúltimas3[ => 7: é o mesmo que ints[7]
0]
```

`subarray()` usa os mesmos argumentos que o método `slice()` e parece funcionar da mesma maneira. Mas há uma diferença importante. `slice()` retorna os elementos especificados em um novo array tipado independente que não compartilha memória com o array original. `subarray()` não copia nenhuma memória; ele apenas retorna uma nova visão dos mesmos valores subjacentes:

```
ints[9] = -1; Altere um valor na matriz original e...
última3[2]      => -1: também muda no subarray
```

O fato de o método subarray() retornar uma nova visualização de um array existente nos leva de volta ao tópico de ArrayBuffer. Cada typedarray tem três propriedades relacionadas ao buffer subjacente:

```
último3.buffer          O objeto ArrayBuffer para um  
arraylast3.buffer digitado === ints.buffer // => true: ambas  
são visualizações do mesmo bufferlast3.byteOffset// => 14:  
esta visualização começa no byte 14 do  
bufferlast3.byteLength// => 6: esta visualização tem 6 bytes  
(ints de 316 bits) longlast3.buffer.byteLength// => 20: mas  
o buffer subjacente tem 20 bytes
```

A propriedade `buffer` é o ArrayBuffer da matriz. `byteOffset` é a posição inicial dos dados da matriz dentro do buffer subjacente. E `byteLength` é o comprimento dos dados da matriz em bytes. Para qualquer array tipado, a, essa invariável deve ser sempre verdadeira:

```
a.length * a.BYTES_PER_ELEMENT === a.byteLength => verdadeiro
```

ArrayBuffers são apenas pedaços opacos de bytes. Você pode acessar esses bytes com matrizes tipadas, mas um ArrayBuffer não é uma matriz tipada. Tenha cuidado, no entanto: você pode usar a indexação de matriz numérica com ArrayBuffers da mesma forma que com qualquer objeto JavaScript. Isso não lhe dá acesso aos bytes no buffer, mas pode causar bugs confusos:

```
let bytes = new Uint8Array(8); bytes[0] = 1;// Define o  
primeiro byte para 1bytes.buffer[0]// => undefined: buffer  
doesn't have index 0bytes.buffer[1] = 255;// Tenta  
incorrectamente definir um byte em
```

```
0  
bufferbytes.buffer[1]    => 255: isto apenas define um  
Propriedade  
JSBYTES[1]                 => 0: a linha acima não foi definida  
o byte
```

Vimos anteriormente que você pode criar um ArrayBuffer com o construtor ArrayBuffer() e, em seguida, criar matrizes tipadas que usam esse buffer. Outra abordagem é criar uma matriz tipada inicial e, em seguida, usar o buffer dessa matriz para criar outras visualizações:

```
let bytes = new Uint8Array(1024);           1024 bytes  
let ints = new Uint32Array(bytes.buffer);    ou 256  
integerslet floats = new Float64Array(bytes.buffer);  
ou 128duplicados
```

11.2.5 DataView e Endianness

As matrizes tipadas permitem que você visualize a mesma sequência de bytes em blocos de 8, 16, 32 ou 64 bits. Isso expõe o "endianness": a ordem em que os bytes são organizados em palavras mais longas. Para maior eficiência, os typedarrays usam o endianness nativo do hardware subjacente. Em sistemas little-endian, os bytes de um número são organizados em um ArrayBuffer do menos significativo para o mais significativo. Em plataformas big-endian, os bytes são organizados do mais significativo para o menos significativo. Você pode determinar o endianness da plataforma subjacente com um código como este:

```
Se o 0x00000001 inteiro estiver organizado na memória como  
01 0000 00, então// estamos em uma plataforma little-  
endian. Em uma plataforma big-endian, obteríamos // bytes  
00 00 00 01 em vez disso.let littleEndian = new  
Int8Array(new Int32Array([1]).buffer)
```

```
[0] === 1;
```

Hoje, as arquiteturas de CPU mais comuns são little-endian. Muitos protocolos de rede e alguns formatos de arquivo binário exigem ordenação de byte big-endian, no entanto. Se você estiver usando matrizes digitadas com dados que vieram da rede ou de um arquivo, você não pode simplesmente assumir que a endianidade da plataforma corresponde à ordem de bytes dos dados. Em geral, ao trabalhar com dados externos, você pode usar Int8Array e Uint8Array para exibir os dados como uma matriz de bytes individuais, mas não deve usar as outras matrizes tipadas com tamanhos de palavras multibyte. Em vez disso, você pode usar a classe DataView, que define métodos para ler e gravar valores de um ArrayBuffer com ordenação de bytes explicitamente especificada:

Suponha que temos uma matriz digitada de bytes de dados binários para processar. Primeiro, criamos um objeto DataView para que possamos ler e escrever de forma flexível// valores desses byteslet view = new DataView(bytes.buffer,

```
bytes.byteOffset,b  
ytes.byteLength);  
  
let int = view.getInt32(0);      Leia big-endian assinado int  
do byte 0int =  
view.getInt32(4, false);        O próximo int também é grande-  
endianint = view.getUint32(8,  
verdadeiro);                  O próximo int é little-endian  
e unsignedview.setUint32(8,  
int, false);                  Escreva de volta em grande-  
formato endian
```

DataView define 10 métodos get para cada uma das 10 classes de matriz tipadas (excluindo Uint8ClampedArray). Eles têm nomes como getInt16(), getInt32(), getBigInt64() e

`getFloat64()`. O primeiro argumento é o deslocamento de bytes dentro do `ArrayBuffer` no qual o valor começa. Todos esses métodos getter, exceto `getInt8()` e `getUint8()`, aceitam um valor opcional booleano como seu segundo argumento. Se o segundo argumento for omitido ou for falso, a ordenação de bytes big-endian será usada. Se o segundo argumento for verdadeiro, a ordenação little-endian será usada.

O `DataView` também define 10 métodos Set correspondentes que gravam valores no `ArrayBuffer` subjacente. O primeiro argumento é o deslocamento no qual o valor começa. O segundo argumento é o valor a ser gravado. Cada um dos métodos, exceto `setInt8()` e `setUint8()`, aceita um terceiro argumento opcional. Se o argumento for omitido ou for falso, o valor será escrito no formato big-endian com o byte mais significativo primeiro. Se o argumento for verdadeiro, o valor será escrito em little-endian format com o byte menos significativo primeiro.

As matrizes tipadas e a classe `DataView` fornecem todas as ferramentas necessárias para processar dados binários e permitem que você escreva programas JavaScript que fazem coisas como descompactar arquivos ZIP ou extrair metadados de arquivos JPEG.

11.3 Correspondência de padrões com RegularExpressions

Uma expressão regular é um objeto que descreve um padrão textual. A classe `RegExp` do JavaScript representa expressões regulares, e ambos `String` e `RegExp` definem métodos que usam expressões regulares para executar funções poderosas de correspondência de padrões e pesquisa e substituição

no texto. Para usar a API RegExp de forma eficaz, no entanto, você também deve aprender a descrever padrões de texto usando a expressão regular grammar, que é essencialmente uma mini linguagem de programação própria. Felizmente, a gramática de expressão regular JavaScript é bastante semelhante à gramática usada por muitas outras linguagens de programação, então você já deve estar familiarizado com ela. (E se você não estiver, o esforço que você investe no aprendizado de expressões regulares JavaScript provavelmente será útil para você em outros contextos de programação também.)

As subseções a seguir descrevem a gramática da expressão regular primeiro e, em seguida, depois de explicar como escrever expressões regulares, explicam como você pode usá-las com os métodos das classes String e RegExp.

11.3.1 Definindo expressões regulares

Em JavaScript, as expressões regulares são representadas por objetos RegExp. Os objetos RegExp podem ser criados com o construtor RegExp(), é claro, mas são criados com mais frequência usando uma sintaxe literal especial. Assim como os literais de string são especificados como caracteres entre aspas, os literais de expressão regular são especificados como caracteres dentro de um par de caracteres de barra (/). Assim, seu código JavaScript pode conter linhas comoesta:

```
let padrão = /s$/;
```

Essa linha cria um novo objeto RegExp e o atribui ao variablepattern. Esse objeto RegExp específico corresponde a qualquer cadeia de caracteres que termine com a letra "s". Essa expressão regular poderia ter equivalente

foi definido com o construtor RegExp(), assim:

```
let padrão = new RegExp("s$");
```

As especificações de padrão de expressão regular consistem em uma série de caracteres. A maioria dos caracteres, incluindo todos os caracteres alfanuméricos, simplesmente descreve os caracteres a serem correspondidos literalmente. Assim, a expressão regular /java/ corresponde a qualquer string que contenha a substring "java". Outros caracteres em expressões regulares não são correspondidos literalmente, mas têm um significado especial. Por exemplo, a expressão regular/s\$/ contém dois caracteres. O primeiro, "s", corresponde a si mesmo literalmente. O segundo, "\$", é um metacaractere especial que corresponde ao final de uma string. Assim, essa expressão regular corresponde a qualquer string que contenha a letra "s" como seu último caractere.

Como veremos, as expressões regulares também podem ter um ou mais caracteres de bandeira que afetam o seu funcionamento. Os sinalizadores são especificados após o segundo caractere de barra em literais RegExp ou como um segundo argumento de cadeia de caracteres para o construtor RegExp(). Se quiséssemos combinar strings que terminam com "s" ou "S", por exemplo, poderíamos usar o sinalizador i com nossa expressão regular para indicar que queremos correspondência sem distinção entre maiúsculas e minúsculas:

```
let padrão = /s$/i;
```

As seções a seguir descrevem os vários caracteres e metacaracteres usados em expressões regulares JavaScript.

CARACTERES LITERAIS Todos os caracteres alfabeticos e dígitos correspondem literalmente em

expressões regulares. A sintaxe de expressão regular JavaScript também suporta certos caracteres não alfabeticos por meio de sequências de escape que começam com uma barra invertida (\). Por exemplo, a sequência \n corresponde a um caractere literal newline em uma cadeia de caracteres. A Tabela 11-1 lista esses caracteres.

Tabela 11-1. Caracteres literais de expressão regular

Perso nagem	Corresponde
Caract ere alfanu mérico	Próprio
\0	O caractere NUL (\u0000)
\t	Guia (\u0009)
\n	Nova linha (\u000A)
\v	Aba vertical (\u000B)
\f	Alimentação de formulário (\u000C)
\r	Retorno de carro (\u000D)
\xnn	O caractere latino especificado pelo número hexadecimal nn; por exemplo, \xA é o mesmo que \n.
\uxxxx	O caractere Unicode especificado pelo número hexadecimal xxxx; Por exemplo, \u0009 é o mesmo que \t.
\u{n}	O caractere Unicode especificado pelo ponto de código n, onde n é de um a seis dígitos hexadecimais entre 0 e 10FFFF. Observe que essa sintaxe só é suportada em expressões regulares que usam o sinalizador u.
\cX	O caractere de controle ^X; por exemplo, \cJ é equivalente ao caractere de nova linha \n.

Vários caracteres de pontuação têm significados especiais em expressões regulares. Eles são:

```
^ $ . * + ? = ! : | \ / ( ) [ ] { }
```

Os significados desses caracteres são discutidos nas seções a seguir. Alguns desses caracteres têm um significado especial apenas em certos contextos de uma expressão regular e são tratados literalmente em outros contextos. Como regra geral, no entanto, se você quiser incluir qualquer um desses caracteres de pontuação literalmente em uma expressão regular, você deve precedê-los com um \. Outros caracteres de pontuação, como aspas e @, não têm significado especial e simplesmente se correspondem literalmente em uma expressão regular.

Se você não consegue se lembrar exatamente quais caracteres de pontuação precisam ser escapados com uma barra invertida, você pode colocar uma barra invertida com segurança antes de qualquer caractere de pontuação. Por outro lado, observe que muitas letras e números têm um significado especial quando precedidos por uma barra invertida, portanto, qualquer letra ou número que você deseja corresponder literalmente não deve ser escapado com uma barra invertida. Para incluir um caractere de barra invertida literalmente em uma expressão regular, você deve escapar dele com uma barra invertida, é claro. Por exemplo, a expressão regular a seguir corresponde a qualquer cadeia de caracteres que inclua uma barra invertida: \V. (E se você usar o construtor RegExp(), lembre-se de que todas as barras invertidas em sua expressão regular precisam ser duplicadas, pois as strings também usam barras invertidas como um caractere de escape.)

CLASSES DE CARACTERES Os caracteres literais individuais podem ser combinados em classes de caracteres por

colocando-os entre colchetes. Uma classe de caracteres corresponde a qualquer caractere contido nela. Assim, a expressão regular/[abc]/ corresponde a qualquer uma das letras a, b ou c. Classes de caracteres negadas também podem ser definidas; estes correspondem a qualquer caractere, exceto aqueles contidos entre colchetes. Uma classe de caracteres negada é especificada colocando um acento circunflexo (^) como o primeiro caractere dentro do colchete esquerdo. The RegExp /[^\w]/ corresponde a qualquer caractere que não seja a, b ou c. As classes de caracteres podem usar um hífen para indicar um intervalo de caracteres. Para corresponder a qualquer caractere minúsculo do alfabeto latino, use /[a-z]/, e para corresponder a qualquer letra ou dígito do alfabeto latino, use/[a-zA-Z0-9]/. (E se você quiser incluir um hífen real em sua classe de caracteres, basta torná-lo o último caractere antes do colchete direito.)

Como determinadas classes de caracteres são comumente usadas, a sintaxe JavaScript regular-expression inclui caracteres especiais e sequências de escape para representar essas classes comuns. Por exemplo, \s corresponde ao caractere de espaço, ao caractere de tabulação e a qualquer outro caractere Unicode whitespace; \S corresponde a qualquer caractere que não seja Unicode whitespace. A Tabela 11-2 lista esses caracteres e resume a sintaxe da classe de caracteres. (Observe que várias dessas sequências de escape de classe de caracteres correspondem apenas a caracteres ASCII e não foram estendidas para funcionar com caracteres Unicode. Você pode, no entanto, definir explicitamente suas próprias classes de caracteres Unicode; por exemplo, /[\u0400-\u04FF]/ corresponde a qualquer caractere cirílico.)

Tabela 11-2. Classes de caracteres de expressão regular

Ch

Caractere	Corresponde
[.]	Qualquer caractere entre colchetes.
[^ .]	Qualquer caractere que não esteja entre colchetes.
[^]	Qualquer caractere, exceto nova linha ou outro terminador de linha Unicode. Ou, se o RegExp usar o sinalizador s, um ponto corresponderá a qualquer caractere, incluindo terminadores de linha.
\w	Qualquer caractere de palavra ASCII. Equivalente a [a-zA-Z0-9_].
\W	Qualquer caractere que não seja um caractere de palavra ASCII. Equivalente a [^a-zA-Z0-9_].
\s	Qualquer caractere de espaço em branco Unicode.
\S	Qualquer caractere que não seja espaço em branco Unicode.
\d	Qualquer dígito ASCII. Equivalente a [0-9].
\D	Qualquer caractere que não seja um dígito ASCII. Equivalente a [^0-9].
\b	Um backspace literal (caso especial).

Observe que os escapes de classe de caractere especial podem ser usados entre colchetes. \s corresponde a qualquer caractere de espaço em branco e \d corresponde a qualquer dígito, então /[\s\d]/ corresponde a qualquer caractere ou dígito de espaço em branco. Observe que há um caso especial. Como você verá mais tarde, o \b tem um significado especial. Quando usado em uma classe de caracteres, no entanto, ele representa o caractere de backspace. Assim, para representar um backspace character literalmente em uma expressão regular, use a classe de caracteres com

um elemento: `/[\b]/`.

CLASSES DE CARACTERES UNICODE

No ES2018, se uma expressão regular usar o sinalizador `u`, haverá suporte para as classes de caracteres `\p{...}` e sua negação `\P{...}`. (No início de 2020, isso é implementado pelo Node, Chrome, Edge e Safari, mas não pelo Firefox.) Essas classes de caracteres são baseadas em propriedades definidas pelo padrão Unicode, e o conjunto de caracteres que elas representam pode mudar à medida que o Unicode evolui.

A classe de caracteres `\d` corresponde apenas a dígitos ASCII. Se você quiser corresponder a um dígito decimal de qualquer um dos sistemas de escrita do mundo, você pode usar `\p{Decimal_Number}u`. E se você quiser corresponder a qualquer caractere que não seja um dígito decimal em nenhum idioma, você pode colocar o `p` em maiúscula e escrever `\P{Decimal_Number}`. Se você quiser corresponder a qualquer caractere numérico, incluindo frações algarismos andrinos, você pode usar `\p{Number}`. Observe que "Decimal_Number" e "Number" não são específicos do JavaScript ou da gramática de expressões regulares: é o nome de uma categoria de caracteres definida pelo padrão Unicode.

A classe de caracteres `\w` só funciona para texto ASCII, mas com `\p`, podemos aproximar uma versão internacionalizada assim:

```
/[\p{Alfabético}\p{Decimal_Number}\p{Marco}]/u
```

(Embora para ser totalmente compatível com a complexidade das línguas do mundo, realmente precisamos adicionar as categorias "Connector_Punctuation" e "Join_Control" também.)

Como exemplo final, a sintaxe `\p` também nos permite definir expressões regulares que correspondem a caracteres de um determinado alfabeto ou script:

```
let greekLetter = /\p{Script=Greek}/u; let
cyrillicLetter = /\p{Script=Cyrillic}/u;
```

REPETIÇÃO Com a sintaxe de expressão regular que você aprendeu até agora, você pode descrever um número de dois dígitos como `\d\d` e um número de quatro dígitos como `\d\d\d\d`. Mas você não tem como descrever, por exemplo, um número que pode ter qualquer número de dígitos ou uma sequência de três letras seguido por um dígito opcional. Esses padrões mais complexos usam a sintaxe regular expression que especifica quantas vezes um elemento de um

a expressão regular pode ser repetida.

Os caracteres que especificam a repetição sempre seguem o padrão ao qual estão sendo aplicados. Como certos tipos de repetição são bastante usados, existem caracteres especiais para representar esses casos. Por exemplo, + corresponde a uma ou mais ocorrências do previouspattern.

A Tabela 11-3 resume a sintaxe de repetição.

Tabela 11-3. Caracteres de repetição de expressão regular

Perso nagem	Significado
{n,m}	Combine o item anterior pelo menos n vezes, mas não mais do que m vezes. }
{n, }	Corresponda ao item anterior n ou mais vezes.
{n}	Corresponda exatamente a n ocorrências do item anterior.
?	Corresponder a zero ou uma ocorrência do item anterior. Ou seja, o previousitem é opcional. Equivalente a {0,1}.
+	Corresponde a uma ou mais ocorrências do item anterior. Equivalente a {1,}.
*	Corresponder a zero ou mais ocorrências do item anterior. Equivalente a {0,}.

As linhas a seguir mostram alguns exemplos:

```
seja r = /\d{2,4}/; Correspondência entre dois e quatro
dígitos = /\w{3}\d?/;// Corresponde exatamente a três
caracteres de palavras e um dígito opcional =
/\s+java\s+;// Corresponde a "java" com um ou mais espaços
antes e depois
```

`r = /[^\]^*/;` *Corresponder a zero ou mais caracteres que são
não abre parênteses*

Observe que, em todos esses exemplos, os especificadores de repetição se aplicam ao caractere único ou à classe de caracteres que os precede. Se você quiser combinar repetições de expressões mais complicadas, precisará definir um grupo com parênteses, que são explicados nas seções a seguir.

Tenha cuidado ao usar o * e ? caracteres de repetição. Como esses caracteres podem corresponder a zero instâncias do que os precede, eles não podem corresponder a nada. Por exemplo, a expressão regular/a*/ na verdade corresponde à string "bbbb" porque a string contém zero ocorrências da letra a!

REPETIÇÃO NÃO GANANCIOSA Os caracteres de repetição listados na Tabela 11-3 correspondem o maior número possível de vezes, enquanto ainda permitem que as partes seguintes da expressão regular correspondam. Dizemos que essa repetição é "gananciosa". Também é possível especificar que a repetição deve ser feita de forma não gananciosa. Basta seguir o caractere ou caracteres de repetição com um ponto de interrogação: ??, +?, *? ou mesmo {1,5}?. Por exemplo, a expressão regular /a+/ corresponde a uma ou mais ocorrências da letra a. Quando aplicada à cadeia de caracteres "aaa", ela corresponde a todas as três letras. Mas /a+?/ corresponde a uma ou mais ocorrências da letra a, correspondendo a alguns caracteres conforme necessário. Quando aplicado à mesma cadeia de caracteres, this pattern corresponde apenas à primeira letra a.

Usar a repetição não gananciosa nem sempre pode produzir os resultados que você

esperar. Considere o padrão $/a+b/$, que corresponde a um ou mais a's, seguido pela letra b. Quando aplicado à cadeia de caracteres "aaab", ele corresponde a toda a cadeia de caracteres. Agora vamos usar a versão não gananciosa: $/a+?b/$. Isso deve corresponder à letra b precedida pelo menor número de a's possível. Quando aplicado à mesma cadeia de caracteres "aaab", você pode esperar que ele corresponda apenas a um a e a última letra b. Na verdade, no entanto, esse padrão corresponde a toda a cadeia de caracteres, assim como a versão gananciosa do padrão. Isso ocorre porque a correspondência de padrões de expressão regular é feita localizando a primeira posição na cadeia de caracteres na qual uma correspondência é possível. Como amatch é possível a partir do primeiro caractere da string, as correspondências mais curtas a partir dos caracteres subsequentes nunca são consideradas.

ALTERNÂNCIA, AGRUPAMENTO E REFERÊNCIAS A GRAMÁTICA DE EXPRESSÕES REGULARES INCLUI CARACTERES ESPECIAIS PARA ESPECIFICAR ALTERNATIVAS, AGRUPAR SUBEXPRESSÕES E REFERIR-SE A SUBEXPRESSÕES ANTERIORES. O | O personagem separa as alternativas. Por exemplo, /ab|cd|ef/ corresponde à string "ab" ou à string "cd" ou à string "ef". E /\d{3}\|[A-Z]{4}/ corresponde a três dígitos ou quatro letras minúsculas.

Observe que as alternativas são consideradas da esquerda para a direita até que uma correspondência seja encontrada. Se a alternativa esquerda corresponder, a alternativa direita será ignorada, mesmo que tenha produzido uma correspondência "melhor". Assim, quando o pattern/a|ab/ é aplicado à string "ab", ele corresponde apenas à primeira letra.

Os parênteses têm várias finalidades em expressões regulares. Um propósito é agrupar itens separados em uma única subexpressão para que os itens possam ser tratados como uma única unidade por |, *, +, ? e assim por diante. Por exemplo

`/java(script)?/` corresponde a "java" seguido pelo "script" opcional. E `/(ab|cd)+|ef/` corresponde à string "ef" ou a uma ou mais repetições de qualquer uma das strings "ab" ou "cd".

Outra finalidade dos parênteses em expressões regulares é definir subpadrões dentro do padrão completo. Quando uma expressão regular é correspondida com êxito a uma string de destino, é possível extrair as partes da string de destino que correspondem a qualquer subpadrão entre parênteses específico. (Você verá como essas subcadeias de caracteres correspondentes são obtidas posteriormente nesta seção.) Por exemplo, suponha que você esteja procurando uma ou mais letras minúsculas seguidas por um ou mais dígitos. Você pode usar o padrão `/[a-z]+d+/?`. Mas suponha que você só se importe com os dígitos no final de cada partida. Se você colocar essa parte do padrão entre parênteses (`/[a-z]+(\d+)/?`), poderá extrair os dígitos de qualquer correspondência que encontrar, conforme explicado mais adiante.

Um uso relacionado de subexpressões entre parênteses é permitir que você faça referência a uma subexpressão posteriormente na mesma expressão regular. Isso é feito seguindo um caractere `\` por um dígito ou dígitos. Os dígitos referem-se à posição da subexpressão entre parênteses dentro da expressão regular. Por exemplo, `\1` refere-se à primeira subexpressão e `\3` refere-se à terceira. Observe que, como as subexpressões podem ser aninhadas em outras, é a posição do parêntese esquerdo que é contada. Na expressão regular a seguir, por exemplo, a subexpressão aninhada `([Ss]cript)` é chamada de `\2`:

```
/([Jj]ava([Ss]cript)?)\sis\s(fun\bw*)/
```

Uma referência a uma subexpressão anterior de uma expressão regular não

não se referem ao padrão dessa subexpressão, mas sim ao texto que corresponde ao padrão. Assim, as referências podem ser usadas para impor uma restrição de que partes separadas de uma cadeia de caracteres contenham exatamente os mesmos caracteres. Por exemplo, a expressão regular a seguir corresponde a zero ou mais caracteres entre aspas simples ou duplas. No entanto, não requer que as aspas de abertura e fechamento correspondam (ou seja, aspas simples ou aspas duplas):

```
/[""][^"]*[""]/
```

Para exigir que as aspas correspondam, use uma referência:

```
/([""])[^"]*\1/
```

O \1 corresponde a qualquer que seja a primeira subexpressão entre parênteses. Neste exemplo, ele impõe a restrição de que a citação de fechamento corresponda à aspa de abertura. Esta expressão regular não permite aspas simples dentro de strings entre aspas duplas ou vice-versa. (Não é legal usar uma referência dentro de uma classe de caracteres, então você não pode escrever:/([""])[^\1]*\1/.)

Quando abordarmos a API RegExp posteriormente, você verá que esse tipo de referência a uma subexpressão entre parênteses é um recurso poderoso das operações de pesquisa e substituição de expressões regulares.

Também é possível agrupar itens em uma expressão regular sem criar uma referência numerada a esses itens. Em vez de simplesmente agrupar os itens dentro de (e), comece o grupo com (? : e termine com). Considere o seguinte padrão:

```
/([Jj]ava(?:[Ss]cript))\sis\s(fun\w*)/
```

Neste exemplo, a subexpressão `(?:[Ss]cript)` é usada simplesmente para agrupamento, então o `?` O caractere de repetição pode ser aplicado ao grupo. Esses parênteses modificados não produzem uma referência, portanto, nesta expressão regular, `\2` refere-se ao texto correspondente a `(fun\w*)`.

A Tabela 11-4 resume os operadores de alternância, agrupamento e referência de expressões regulares.

Tabela 11-4. Alternância de expressão regular, agrupamento e caracteres de referência

P er s o n a g e m	Significado
	Alternância: corresponde à subexpressão à esquerda ou à direita.
(Agrupamento: agrupe itens em uma única unidade que pode ser usada com <code>*</code> , <code>+</code> , <code>?</code> , <code> </code> e em breve. Lembre-se também dos caracteres que correspondem a esse grupo para uso com referências posteriores.
:	
)	
(Somente agrupamento: agrupe os itens em uma única unidade, mas não se lembre dos caracteres que correspondem a esse grupo.
:	
.	
.	
)	

- \ Corresponda aos mesmos caracteres que foram correspondidos quando o número do grupo
- n foi correspondido pela primeira vez. Grupos são subexpressões dentro de parênteses (possivelmente aninhados). Os números de grupo são atribuídos contando parênteses esquerdos da esquerda para a direita. Grupos formados com ?: não são numerados.

GRUPOS DE CAPTURA NOMEADOS

O ES2018 padroniza um novo recurso que pode tornar as expressões regulares mais autodocumentadas e mais fáceis de entender. Esse novo recurso é conhecido como "grupos de captura nomeados" e nos permite associar um nome a cada parêntese esquerdo em uma expressão regular para que possamos nos referir ao texto correspondente por nome e não por número. Igualmente importante: usar nomes permite que alguém que lê o código entenda mais facilmente o propósito dessa parte da expressão regular. No início de 2020, esse recurso foi implementado no Node, Chrome, Edge e Safari, mas ainda não pelo Firefox.

Para nomear um grupo, use (?<...>) em vez de (e coloque o nome entre os colchetes angulares. Por exemplo, aqui está uma expressão regular que pode ser usada para verificar a formatação da linha final de um endereço de correspondência dos EUA:

```
/(? <city>\w+) (?<state>[A-Z]{2}) (?<zipcode>\d{5})(?<zip9>-\d{4})?/
```

Observe quanto contexto os nomes dos grupos fornecem para tornar a expressão regular mais fácil de entender. Em §11.3.2, quando discutirmos os métodos String replace() e match() e o método RegExp exec(), você verá como a API RegExp permite que você se refira ao texto que corresponde a cada um desses grupos por nome em vez de por posição.

Se você quiser se referir a um grupo de captura nomeado em uma expressão regular, também poderá fazer esse sobrenome. No exemplo anterior, pudemos usar uma expressão regular "backreference" para escrever um RegExp que corresponderia a uma string entre aspas simples ou duplas em que as aspas abertas e fechadas tinham que corresponder. Poderíamos reescrever este RegExp usando um grupo de captura nomeado e um namedbackreference como este:

```
/(? <quote>[ ' ]) [^"]*\k<quote>/
```

O \k<quote> é uma referência inversa nomeada para o grupo nomeado que captura as aspas abertas.

ESPECIFICANDO POSIÇÕES DE CORRESPONDÊNCIA Como descrito anteriormente, muitos elementos de uma expressão regular correspondem a um único caractere em uma cadeia de caracteres. Por exemplo, ls corresponde a um único caractere de espaço em branco. Outros elementos de expressão regular correspondem às posições

entre caracteres em vez de caracteres reais. \b, por exemplo, corresponde a um limite de palavra ASCII — o limite entre um \w(caractere de palavra ASCII) e um \W (caractere que não é de palavra), ou o limite entre um caractere de palavra ASCII e o início ou o fim de uma string. Elementos como \b não especificam nenhum caractere a ser usado em uma cadeia de caracteres correspondente; O que eles especificam, no entanto, são posições legais nas quais uma correspondência pode ocorrer. Às vezes, esses elementos são chamados de âncoras de expressão regular porque ancoram o padrão a uma posição específica na cadeia de caracteres de pesquisa. Os elementos de âncora mais comumente usados são ^, que vincula o padrão ao início da cadeia de caracteres, e \$, que ancora o padrão ao final da cadeia de caracteres.

Por exemplo, para corresponder a palavra "JavaScript" em uma linha por si só, você pode usar a expressão regular /^JavaScript\$/ . Se você quiser pesquisar por "Java" como uma palavra por si só (não como um prefixo, como está em "JavaScript"), você pode tentar o padrão \sJava\s/, que requer um espaço antes e depois da palavra. Mas há dois problemas com esta solução. Primeiro, ele não corresponde a "Java" no início ou no final de uma string, mas apenas se aparecer com espaço em ambos os lados. Em segundo lugar, quando esse padrão encontra uma correspondência, a string correspondente que ele retorna tem espaços à esquerda e à direita, o que não é exatamente o que é necessário. Portanto, em vez de combinar caracteres de espaço reais com \s, combine (ou ancore a) limites de palavras com \b. A expressão resultante é /\bJava\b/. O elemento \B ancora a correspondência em um local que não é um limite de palavra. Assim, o padrão /\B[Ss]cript/ corresponde a "JavaScript" e "postscript", mas não a "script" ou "Scripting".

Você também pode usar expressões regulares arbitrárias como condições de âncora. Se

Você inclui uma expressão dentro de caracteres (?= e), é uma declaração de antecipação e especifica que os caracteres entre aspas devem corresponder, sem realmente correspondê-los. Por exemplo, para corresponder ao nome de uma linguagem de programação comum, mas apenas se for seguido por dois pontos, você pode usar /[Jj]ava([Ss]cript)? (?=\:)/. Esse padrão corresponde à palavra "JavaScript" em "JavaScript: The DefinitiveGuide", mas não corresponde a "Java" em "Java in a Nutshell" porque não é seguido por dois pontos.

Se, em vez disso, você introduzir uma asserção com (?!, ela será uma asserção lookahead negativa, que especifica que os caracteres a seguir não devem corresponder. Por exemplo, /Java(?! Script)([A-Z]\w*)/ corresponde a "Java" seguido por uma letra maiúscula e qualquer número de caracteres de palavra ASCII adicionais, desde que "Java" não seja seguido por "Script". Ele corresponde a "JavaBeans", mas não a "javanês", e corresponde a "JavaScrip", mas não a "JavaScript" ou "JavaScripter". A Tabela 11-5 resume as âncoras de expressão regular.

Tabela 11-5. Caracteres âncora de expressão regular

Pe rs on ag e m	Significado
^	Corresponda ao início da cadeia de caracteres ou, com o sinalizador m, ao início de uma linha.
\$	Combine o final da cadeia de caracteres e, com o sinalizador m, o final de uma linha.
\b	Corresponder a um limite de palavra. Ou seja, corresponda a posição entre um caractere \w e um caractere \W ou entre um caractere \w e o início ou o fim de uma cadeia de caracteres. (Observe, no entanto, que [\b] corresponde ao backspace.)

\ Corresponder a uma posição que não seja um limite de palavra.
B

(? Uma afirmação positiva de antecipação. Exija que os caracteres a seguir
=p correspondam ao padrão p, mas não inclua esses caracteres na correspondência.
)

(? Uma afirmação negativa de antecipação. Exija que os seguintes caracteres
! não correspondam ao padrão p.
p)

ASSERÇÕES DE LOOKBEHIND

O ES2018 estende a sintaxe de expressão regular para permitir asserções "lookbehind". São como lookahead assertions, mas referem-se ao texto antes da posição atual da correspondência. No início de 2020, eles são implementados no Node, Chrome e Edge, mas não no Firefox ou no Safari.

Especifique uma asserção lookbehind positiva com `(?<=...)` e uma asserção lookbehind negativa com `(?<!...)`. Por exemplo, se você estivesse trabalhando com endereços de correspondência dos EUA, poderia corresponder a um CEP de 5 dígitos, mas somente quando ele seguir uma abreviação de estado de duas letras, como esta:

```
/(?<= [A-Z]{2} )\d{5}/
```

E você pode combinar uma sequência de dígitos que não é precedida por um símbolo de moeda Unicode com uma afirmação lookbehind negativa como esta:

```
/(?<! [\p{Currency_Symbol}\d.])\d+(\.\d+)?/u
```

FLAGSEuma expressão muito regular pode ter um ou mais sinalizadores associados a ela para alterar seu comportamento correspondente. O JavaScript define seis sinalizadores possíveis, cada um dos quais é representado por uma única letra. Os sinalizadores são especificados após o segundo caractere / de um literal de expressão regular ou como uma string passada como o segundo argumento para o construtor RegExp(). O

Os sinalizadores suportados e seus significados são:

g

O sinalizador *g* indica que a expressão regular é "global" — ou seja, que pretendemos usá-la para encontrar todas as correspondências dentro de uma cadeia de caracteres, em vez de apenas encontrar a primeira correspondência. Esse sinalizador não altera a maneira como a correspondência de padrões é feita, mas, como veremos mais adiante, altera o comportamento do método `String match()` e do método `RegExexec()` de maneiras importantes.

eu

O sinalizador *i* especifica que a correspondência de padrões não deve diferenciar maiúsculas de minúsculas.

m

O sinalizador *m* especifica que a correspondência deve ser feita no modo "multilinha". Ele diz que o `RegExp` será usado com strings de várias linhas e que as âncoras `^` e `$` devem corresponder ao início e ao fim da string e também ao início e ao fim de linhas individuais dentro da string.

s

Como o sinalizador *m*, o sinalizador *s* também é útil ao trabalhar com texto que inclui novas linhas. Normalmente, um `".` em uma expressão regular corresponde a qualquer caractere, exceto um terminador de linha. No entanto, quando o sinalizador *s* é usado, `".` corresponderá a qualquer caractere, incluindo terminadores de linha. O sinalizador *s* foi adicionado ao JavaScript no ES2018 e, no início de 2020, é compatível com Node, Chrome, Edge e Safari, mas não com o Firefox.

No

O sinalizador *u* significa Unicode e faz a expressão regular

correspondem a pontos de código Unicode completos em vez de corresponder a valores de 16 bits. Este sinalizador foi introduzido no ES6, e você deve criar o hábito de usá-lo em todas as expressões regulares, a menos que tenha algum motivo para não fazê-lo. Se você não usar esse sinalizador, seus RegExps não funcionarão bem com texto que inclua emoji e outros caracteres (incluindo muitos caracteres chineses) que exijam mais de 16 bits. Sem o sinalizador `u`, o caractere `".` corresponde a qualquer valor 1 UTF-16 de 16 bits. Com o sinalizador, no entanto, `".` corresponde a um ponto de código Unicode, incluindo aqueles que têm mais de 16 bits. Definir o sinalizador `u` em `aRegExp` também permite que você use a nova sequência de escape `\u{...}` para caractere Unicode e também habilita a notação `\p{...}` para classes de caracteres Unicode.

`e`

O sinalizador `y` indica que a expressão regular é "pegajosa" e deve corresponder no início de uma string ou no primeiro caractere após a correspondência anterior. Quando usado com uma expressão regular projetada para encontrar uma única correspondência, ele efetivamente trata essa expressão regular como se começasse com `^` para ancorá-la no início da string. Esse sinalizador é mais útil com expressões regulares que são usadas repetidamente para localizar todas as correspondências dentro de uma string. Nesse caso, ele faz com que um comportamento especial do método `Stringmatch()` e do método `RegExp exec()` imponha que cada correspondência subsequente seja ancorada à posição da string na qual a última terminou.

Esses sinalizadores podem ser especificados em qualquer combinação e em qualquer ordem. Por exemplo, se você quiser que sua expressão regular seja uma correspondência sem distinção entre maiúsculas e minúsculas com reconhecimento de Unicode e pretender usá-la para localizar várias correspondências em uma cadeia de caracteres, especifique os sinalizadores `uig`, `gui` ou qualquer outra permutação dessas três letras.

11.3.2 Métodos de cadeia de caracteres para correspondência de padrões

Até agora, descrevemos a gramática usada para definir expressões regulares, mas não explicamos como essas expressões regulares podem realmente ser usadas no código JavaScript. Agora estamos mudando para cobrir a API para usar objetos RegExp. Esta seção começa explicando os métodos de cadeia de caracteres que usam expressões regulares para executar operações de correspondência de padrões e pesquisa e substituição. As seções que se seguem, esta continua, a discussão sobre correspondência de padrões com JavaScript expressões regulares, discutindo o objeto RegExp e seus métodos e propriedades.

SEARCH()Strings suportam quatro métodos que usam expressões regulares. O mais simples search(). Este método recebe um argumento de expressão regular e retorna a posição do caractere do início da primeira substring correspondente ou -1 se não houver correspondência:

```
"JavaScript".search(/script/ui)  => 4  
"Python".search(/script/ui)      => -1
```

Se o argumento para search() não for uma expressão regular, ele será primeiro convertido em uma, passando-o para o construtor RegExp. search() não suporta pesquisas globais; Ele ignora o sinalizador g de seu argumento regularExpression.

REPLACE() O método replace() executa uma operação de busca e substituição. Ele usa uma expressão regular como seu primeiro argumento e uma string de substituição

como segundo argumento. Ele pesquisa a cadeia de caracteres na qual é chamado para correspondências com o padrão especificado. Se a expressão regular tiver o gflag definido, o método replace() substituirá todas as correspondências na string pela string de substituição; caso contrário, ele substitui apenas a primeira correspondência encontrada. Se o primeiro argumento a ser replace() for uma string em vez de uma expressão regular, o método procurará essa string literalmente em vez de convertê-la em uma expressão regular com o construtor RegExp(), como search() faz. Por exemplo, você pode usar replace() da seguinte maneira para fornecer letras maiúsculas e minúsculas uniformes da palavra "JavaScript" em uma sequência de texto:

```
Não importa como esteja em maiúsculas, substitua-o  
pelotexto de capitalização  
correta.replace(/javascript/gi, "JavaScript");
```

replace() é mais poderoso do que isso, no entanto. Lembre-se de que as subexpressões entre parênteses de uma expressão regular são numeradas da esquerda para a direita e que a expressão regular lembra o texto que cada subexpressão corresponde. Se um \$ seguido por um dígito aparecer na cadeia de caracteres de substituição, replace() substituirá esses dois caracteres pelo texto que corresponde à subexpressão especificada. Este é um recurso muito útil. Você pode usá-lo, por exemplo, para substituir aspasem uma string por outros caracteres:

```
Uma aspa é uma aspa, seguida por qualquer número de//  
caracteres que não aspas (que capturamos), seguido// por  
outra aspa.let quote = "/[^"]*/g;// Substitua as aspas  
retas por guillemets// deixando o texto citado (armazenado  
em $1) inalterado.'Ele disse "stop"'.replace(quote,  
'«$1»')// => 'Ele disse«stop»'
```

Se o seu RegExp usa grupos de captura nomeados, você pode consultar o texto correspondente por nome em vez de por número:

```
let citação = /"(?<quotedText>[^"]*)"/g; ' Ele  
disse "pare".replace(citação, '«$<quotedText>»')      => 'Ele  
disse «pare»'
```

Em vez de passar uma string de substituição como o segundo argumento a `replace()`, você também pode passar uma função que será invocada para calcular o valor de substituição. A função de substituição é invocada com vários argumentos. O primeiro é todo o texto correspondente. Em seguida, se o RegExp tiver grupos de captura, as substrings que foram capturadas por esses grupos serão passadas como argumentos. O próximo argumento é a posição dentro da string na qual a correspondência foi encontrada. Depois disso, toda a string que `replace()` foi chamada é passada. E, finalmente, se o RegExp contiver algum grupo de captura nomeado, o último argumento para a função de substituição será um objeto cujos nomes de propriedade correspondem aos nomes do grupo de captura e cujos valores são o texto correspondente. Como exemplo, aqui está o código que usa uma replacementfunction para converter inteiros decimais em uma string em hexadecimal:

```
let s = "15 vezes 15 é 225"; s.replace(/\d+/gu,  
n => parseInt(n).toString(16))                      => "f  
vezes f é e1"
```

MATCH() O método `match()` é o mais geral dos métodos de expressão regular da String. Ele usa uma expressão regular como seu único argumento (ou converte seu argumento em uma expressão regular passando-o para o construtor `RegExp()`) e retorna uma matriz que contém os resultados

da correspondência ou null se nenhuma correspondência for encontrada. Se a expressão regular tiver o sinalizador g definido, o método retornará uma matriz de todas as correspondências que aparecem na string. Por exemplo:

```
"7 mais 8 é igual a           => ["7", "8", "15"]  
15".match(/\d+/g)
```

Se a expressão regular não tiver o sinalizador g definido, match() não fará uma pesquisa global; ele simplesmente procura a primeira correspondência. Neste caso não global, match() ainda retorna um array, mas os elementos do array são completamente diferentes. Sem o sinalizador g, o primeiro elemento da matriz virada é a string correspondente, e quaisquer elementos restantes são as substrings que correspondem aos grupos de captura entre parênteses da expressão regular. Assim, se match() retornar uma matriz a, a[0] contém a correspondência completa, a[1] contém a substring que correspondeu à primeira expressão entre parênteses e assim por diante. Para traçar um paralelo com o método replace(), a[1] é a mesma string que \$1, a[2] é o mesmo que \$2 e assim por diante.

Por exemplo, considere analisar uma URL com o seguinte código:

```
Uma análise de URL muito simples RegExplet url =  
/(\w+):\/\/([\w.]+)\/(|\$*/); let text = "Visite meu blog em  
http://www.example.com/~david"; let correspondência =  
texto.match(url); let fullurl, protocolo, host, caminho; if  
(match !== null) {  
  
    fullurl = correspondência[0]; fullurl ==  
    "http://www.example.com/~david"  
    protocolo = correspondência[1]; protocolo == "http"  
    host = correspondência[2]; anfitrião == "www.example.com"  
    caminho =                      caminho == "~david"  
}   correspondência[3];
```

Nesse caso não global, o array retornado por match() também possui algumas propriedades de objeto além dos elementos do array numerados. A propriedade input refere-se à string na qual match() foi chamado. A propriedade index é a posição dentro dessa cadeia de caracteres na qual a correspondência começa. E se a expressão regular contiver capturegroups nomeados, a matriz retornada também terá uma propriedade groups whosevalue é um objeto. As propriedades desse objeto correspondem aos nomes dos grupos nomeados e os valores são o texto correspondente. Poderíamos reescrever o exemplo anterior de análise de URL, por exemplo, assim:

```
let url = /(?(<protocol>)\w+):\//\/(?(<host>[\w.]+)\//(?
<path>\S*)/; let text = "Visite meu blog em
http://www.example.com/~david"; let correspondência =
texto.match(url); match[0]// =>
"http://www.example.com/~david"match.input// =>
textmatch.index// => 17match.groups.protocol// =>
"http"match.groups.host// =>
"www.example.com"match.groups.path// => "~david"
```

Vimos que match() se comporta de maneira bem diferente dependendo se o RegExp tem o sinalizador g definido ou não. Também existem diferenças importantes, mas menos dramáticas, no comportamento quando o sinalizador y é definido. Lembre-se que o sinalizador y torna uma expressão regular "pegajosa" restringindo onde as correspondências de string podem começar. Se um RegExp tiver os sinalizadores g andy definidos, então match() retorna uma matriz de strings correspondentes, assim como faz quando g é definido sem y. Mas a primeira correspondência deve começar no início da cadeia de caracteres e cada correspondência subsequente deve começar no caractere imediatamente após a correspondência anterior.

Se o sinalizador `y` for definido sem `g`, `match()` tentará encontrar uma única correspondência e, por padrão, essa correspondência será restrita ao início da string. No entanto, você pode alterar essa posição inicial de correspondência padrão, definindo a propriedade `lastIndex` do objeto `RegExp` no índice no qual você deseja corresponder. Se uma correspondência for encontrada, esse `lastIndex` será atualizado automaticamente para o primeiro caractere após a correspondência, portanto, se você chamar `match()` novamente, nesse caso, ele procurará uma correspondência subsequente. (`lastIndex` pode parecer um nome estranho para uma propriedade que especifica a posição na qual começar a próxima correspondência. Veremos isso novamente quando abordarmos o método `RegExp exec()`, e seu nome pode fazer mais sentido nesse contexto.)

```
letra da vogal = /[aeiou]/y; Correspondência de vogal pegajosa
"teste".match(vogal)      => nulo: "teste" não começa
com uma
vogal.vogal.lastIndex = 1; Especificar uma correspondência diferente
posição"teste".match(
vogal)[0]                  => "e": encontramos uma vogal em
posição
1vogal.lastIndex          => 2: lastIndex foi automaticamente
atualizado"teste".
match(vogal)               => nulo: sem vogal na posição 2
vogal.lastIndex            => 0: lastIndex é redefinido após
partida fracassada
```

Vale a pena notar que passar uma expressão regular não global para o método `match()` de uma string é o mesmo que passar a string para o método `exec()` da expressão regular: o array retornado e suas propriedades são os mesmos em ambos os casos.

MATCHALL() O método `matchAll()` é definido no ES2020 e no início de 2020

é implementado por navegadores da Web modernos e pelo Node.

`matchAll()` espera um `RegExp` com o sinalizador `g` definido. No entanto, em vez de retornar uma matriz de substrings correspondentes como `match()`, ele retorna um iterador que produz o tipo de objetos de correspondência que `match()` retorna quando usado com um `RegExp` não global. Isso torna `matchAll()` a maneira mais fácil e geral de percorrer todas as correspondências dentro de uma string.

Você pode usar `matchAll()` para percorrer as palavras em uma string de texto como esta:

Um ou mais caracteres alfabéticos Unicode entre limites de palavras
`const palavras = /\b\p{Alfabético}+\b/gu; \p ainda não é suportado`
`const text = "Este é um teste ingênuo do método matchAll()." ; for(let palavra of text.matchAll(words)) {`

```
console.log('Encontrado '${word[0]}' no índice${word.index}.');
```

Você pode definir a propriedade `lastIndex` de um objeto `RegExp` para `tellmatchAll()` em qual índice na string começar a corresponder. Ao contrário dos outros métodos de correspondência de padrões, no entanto, `matchAll()` nunca modifica a propriedade `lastIndex` do `RegExp` que você o chama, e isso torna muito menos provável que cause bugs em seu código.

SPLIT() O último dos métodos de expressão regular do objeto `String` é `split()`. Esse método quebra a cadeia de caracteres na qual ele é chamado em uma matriz de subcadeias de caracteres, usando o argumento como separador. Pode ser usado

com um argumento de string como este:

```
"123.456.789".split(",")          => ["123", "456",
"789"]
```

O método split() também pode receber uma expressão regular como seu argumento, e isso permite que você especifique separadores mais gerais. Aqui, chamamos isso com um separador que inclui uma quantidade arbitrária de espaço em branco em ambos os lados:

```
"1, 2, 3,\n4, 5".split(/\s*,\s*/)  => ["1", "2", "3", "4",
"5"]
```

Surpreendentemente, se você chamar split() com um delimitador RegExp e a expressão regular incluir grupos de captura, o texto que corresponderaos grupos de captura será incluído na matriz retornada. Por exemplo:

```
const htmlTag = /<([>]+)>/; // < seguido por um ou mais non-
>, seguido por >"Testing<br/>1,2,3".split(htmlTag)// =>
["Testing", "br/", "1,2,3"]
```

11.3.3 A classe RegExp

Esta seção documenta o construtor RegExp(), as propriedades das instâncias RegExp e dois métodos importantes de correspondência de padrões definidos pela classe RegExp.

O construtor RegExp() recebe um ou dois argumentos de string e cria um novo objeto RegExp. O primeiro argumento para esse construtor é uma cadeia de caracteres que contém o corpo da expressão regular — o texto que

apareceria dentro de barras em um literal de expressão regular. Observe que tanto os literais de string quanto as expressões regulares usam o caractere \ para sequências de escape, portanto, quando você passa uma expressão regular toRegExp() como um literal de string, você deve substituir cada caractere \ por \\. O segundo argumento para RegExp() é opcional. Se fornecido, indica os sinalizadores de expressão regular. Deve ser g, i, m, s, u, y ou qualquer combinação dessas letras.

Por exemplo:

*Encontre todos os números de cinco dígitos em uma string.
Observe o double\\ neste caso.*

```
let zipcode = new  
RegExp("\\d{5}", "g");
```

O construtor RegExp() é útil quando uma expressão regular está sendo criada dinamicamente e, portanto, não pode ser representada com a sintaxe literal da expressão regular. Por exemplo, para pesquisar uma string inserida pelo usuário, uma expressão regular deve ser criada em tempo de execução com RegExp().

Em vez de passar uma string como o primeiro argumento para RegExp(), você também pode passar um objeto RegExp. Isso permite que você copie uma expressão regular e altere seus sinalizadores:

```
let exactMatch = /JavaScript/; let  
caseInsensitive = new RegExp(exactMatch, "i");
```

PROPRIEDADES REGEXP OS objetos

REGEXP têm as seguintes propriedades:

fonte

Essa propriedade somente leitura é o texto de origem da expressão regular: os caracteres que aparecem entre as barras em um literal RegExp.

Sinalizadores

Essa propriedade somente leitura é uma cadeia de caracteres que especifica o conjunto de letras que representam os sinalizadores para o RegExp.

global

Uma propriedade booleana somente leitura que é verdadeira se o sinalizador g estiver definido.

ignoreCase

Uma propriedade booleana somente leitura que será verdadeira se o sinalizador i estiver definido.

multilinhas

Uma propriedade booleana somente leitura que será verdadeira se o sinalizador m estiver definido.

dotAll

Uma propriedade booleana somente leitura que será verdadeira se o sinalizador s estiver definido.

Unicode

Uma propriedade booleana somente leitura que é verdadeira se o sinalizador u estiver definido.

pegajoso

Uma propriedade booleana somente leitura que será verdadeira se o sinalizador y estiver definido.

lastIndex

Essa propriedade é um inteiro de leitura/gravação. Para padrões com os sinalizadores g ou y, ele especifica a posição do caractere na qual a próxima pesquisa é iniciada. Ele é usado pelos métodos exec() e test(), descritos nas próximas duas subseções.

TEST()O método `test()` da classe `RegExp` é a maneira mais simples de usar uma expressão regular. Ele usa um único argumento de cadeia de caracteres e retorna `true`if a string corresponder ao padrão ou `false` se não corresponder.

`test()` funciona simplesmente chamando o método (muito mais complicado)`exec()` descrito na próxima seção e retornando `true` if`exec()` retorna um valor não nulo. Por isso, se você usar `test()`com um `RegExp` que usa os sinalizadores `g` ou `y`, seu comportamento dependerá do valor da propriedade `lastIndex` do objeto `RegExp`, que pode mudar inesperadamente. Consulte "A propriedade `lastIndex` e `RegExpReuse`" para obter mais detalhes.

EXEC()O método `RegExp exec()` é a maneira mais geral e poderosa de usar expressões regulares. Ele usa um único argumento de cadeia de caracteres e procura uma correspondência nessa cadeia de caracteres. Se nenhuma correspondência for encontrada, ele retornará nulo. Se um matchfor encontrado, no entanto, ele retornará um array exatamente como o array retornado pelo método `match()` para pesquisas não globais. O elemento 0 da matriz contém a cadeia de caracteres que correspondeu à expressão regular e todos os elementos subsequentes da matriz contêm as subcadeias de caracteres que corresponderam a qualquer grupo de captura. A matriz retornada também tem propriedades nomeadas: a propriedade `index` contém a posição do caractere na qual a correspondência ocorreu e a propriedade `input` especifica a cadeia de caracteres que foi pesquisada e a propriedade `groups`, se definida, refere-se a um objeto que contém as subcadeias de caracteres correspondentes a qualquer grupo de captura nomeado.

Ao contrário do método String match(), exec() retorna o mesmo tipo de array, independentemente de a expressão regular ter ou não o sinalizador g global. Lembre-se de que match() retorna uma matriz de correspondências quando passada uma expressão globalregular. exec(), por outro lado, sempre retorna uma única correspondênciæ fornece informações completas sobre essa correspondência. Quando exec() é chamado em uma expressão regular que tem o sinalizador g global ou o sinalizador y sticky definido, ele consulta a propriedade lastIndex do objeto RegExp para determinar por onde começar a procurar uma correspondência. (E se a bandeira y estiver definida, ela também restringe a correspondência a começar nessa posição.) Para um objeto RegExp recém-criado, lastIndex é 0 e a pesquisa começa no início da cadeia de caracteres. Mas cada vez que exec() encontra uma correspondência com sucesso, ele atualiza a propriedade lastIndex para o índice do caractere imediatamente após o texto correspondente. Se exec() não conseguir encontrar amatch, ele redefinirá lastIndex para 0. Esse comportamento especial permite que você chame exec() repetidamente para percorrer todas as correspondências de expressão regular em uma string. (Embora, como descrevemos, no ES2020 e posterior, o método matchAll() de String seja uma maneira mais fácil de percorrer todas as correspondências.) Por exemplo, o loop no código a seguir será executado duas vezes:

```
let padrão = /Java/g; let text = "JavaScript  
> Java"; vamos combinar; while((match =  
pattern.exec(text)) !== null) {  
  
  console.log('Correspondência ${correspondência[0]}'  
  correspondente em ${correspondência.índice});  
  console.log('A próxima pesquisa começa  
em${pattern.lastIndex}');}
```

Como você já viu, a API de expressão regular do JavaScript é complicada. O uso da propriedade lastIndex com os sinalizadores g e y é uma parte particularmente estranha dessa API. Ao usar esses sinalizadores, você precisa ter um cuidado especial ao chamar os métodos match(), exec() ou test() porque o comportamento desses métodos depende de lastIndex e o valor de lastIndex depende do que você fez anteriormente com o objeto RegExp. Isso facilita a escrita de código com bugs.

Suponha, por exemplo, que quiséssemos encontrar o índice de todas as <p> tags dentro de uma cadeia de caracteres de texto HTML. Podemos escrever um código como este:

```
let match, posições = []; while((match = /<p>/g.exec(html)) !== null)  
{ // POSSÍVEL LOOP INFINITO  
  posições.push(match.index);}
```

Este código não faz o que queremos. Se a string html contiver pelo menos uma <p> tag, ela fará um loop para sempre. O problema é que usamos um literal RegExp na condição de loop while. Para cada iteração do loop, estamos criando um novo objeto RegExp com lastIndex definido como 0, então exec() sempre começa no início da string e, se houver uma correspondência, ela continuará correspondendo. A solução, é claro, é definir o RegExp uma vez e salvá-lo em uma variável para que estejamos usando o mesmo objeto RegExp para cada iteração do loop.

Por outro lado, às vezes reutilizar um objeto RegExp é a coisa errada a fazer. Suponha, por exemplo, que queremos percorrer todas as palavras em um dicionário para encontrar palavras que contenham pares de letras duplas:

```
let dicionário = [ "maçã", "livro", "café" ];  
let doubleLetterWords = []; let doubleLetter =  
/(\w)\1/g;  
  
for(let palavra do dicionário) {  
  if (doubleLetter.test(word)) {  
    doubleLetterWords.push(palavra);}}doubleLetterWords// =>  
["apple", "coffee"]; "book" está faltando!
```

Como definimos o sinalizador g no RegExp, a propriedade lastIndex é alterada após successfulmatches, e o método test() (que é baseado em exec()) começa a procurar uma correspondência na posição especificada por lastIndex. Depois de combinar o "pp" em "apple", lastIndex é 3, e então começamos pesquisando a palavra "livro" na posição 3 e não vemos o "oo" que ela contém.

Poderíamos corrigir esse problema removendo o sinalizador g (que não é realmente necessário neste exemplo específico), ou movendo o literal RegExp para o corpo do loop para que ele seja recriado em cada iteração, ou redefinindo explicitamente lastIndex para zero antes de cada chamada para test().

A moral aqui é que lastIndex torna a API RegExp propensa a erros. Portanto, tenha muito cuidado ao usar os sinalizadores g ou y e fazer loops. E no ES2020 e posterior, use o método String matchAll()

em vez de exec() para contornar esse problema, já que matchAll() não modifica lastIndex.

11.4 Datas e horários

A classe Date é a API do JavaScript para trabalhar com datas e horas. Crie um objeto Date com o construtor Date(). Sem argumentos, ele retorna um objeto Date que representa a data e a hora atuais:

```
let now = new Date();      A hora atual
```

Se você passar um argumento numérico, o construtor Date() interpretará esse argumento como o número de milissegundos desde a época de 1970:

```
let época = new Date(0);  Meia-noite, 1º de janeiro de 1970, GMT
```

Se você especificar dois ou mais argumentos inteiros, eles serão interpretados como ano, mês, dia do mês, hora, minuto, segundo e milissegundo no fuso horário local, como no seguinte:

```
let century = new Date(2100,          Ano 2100
                      0,              Janeiro
                      1,              1º
                      2, 3, 4, 5);  02:03:04.005, local
                      Hora
```

Uma peculiaridade da API Date é que o primeiro mês de um ano é o número 0, mas o primeiro dia de um mês é o número 1. Se você omitir os campos de hora, o construtor Date() padronizará todos eles como 0, definindo a hora para meia-noite.

Observe que, quando invocado com vários números, o Date()

interpreta-os usando qualquer fuso horário para o qual o localcomputer está definido. Se você quiser especificar uma data e hora em UTC (Tempo Coordenado Universal, também conhecido como GMT), poderá usar theDate.UTC(). Esse método estático usa os mesmos argumentos que o construtor Date(), interpreta-os em UTC e retorna um carimbo de data/hora de milissegundos que você pode passar para o construtor Date():

```
Meia-noite na Inglaterra, 1º de janeiro de  
2100let century = new Date(Date.UTC(2100, 0, 1));
```

Se você imprimir uma data (com console.log(século), por exemplo), ela será, por padrão, impressa em seu fuso horário local. Se você quiser exibir uma data em UTC, você deve convertê-la explicitamente em uma string com toUTCString() ou toISOString().

Finalmente, se você passar uma string para o construtor Date(), ele tentará analisar essa string como uma especificação de data e hora. O construtor pode analisar datas especificadas nos formatos produzidos pelos métodos toString(), toUTCString() e toISOString():

```
let século = new Data("2100-01-01T00:00:00Z"); Um ISO  
Formato de data
```

Depois de ter um objeto Date, vários métodos get e set permitem que você consulte e modifique os campos ano, mês, dia do mês, hora, minuto, segundo e milissegundo da Data. Cada um desses métodos tem duas formas: uma que obtém ou define usando a hora local e outra que obtém ou define usando a hora UTC. Para obter ou definir o ano de um objeto Date, por exemplo, você usaria getFullYear(), getUTCFullYear(), setFullYear() ou setUTCFullYear():

```
let d = new Date();           Comece com o  
atual  
d.getFullYear(d.getFullYear() + Incrementar o ano  
1);
```

Para obter ou definir os outros campos de uma data, substitua "FullYear" no nome do método por "Mês", "Data", "Horas", "Minutos", "Segundos" ou "Milissegundos". Alguns dos métodos de definição de datas permitem que você defina mais de um campo por vez. `setFullYear()` e `setUTCFullYear()` também permitem que você defina o mês e o dia do mês. E `setHours()` e `setUTCHours()` permitem que você especifique os campos de minutos, segundos e milissegundos além do campo de horas.

Observe que os métodos para consultar o dia do mês são `getDate()` e `getUTCDate()`. As funções de som mais natural `getDay()` e `getUTCDay()` retornam o dia da semana (0 para domingo a 6 para sábado). O dia da semana é somente leitura, portanto, não há um método `setDay()` correspondente.

11.4.1 Carimbos de data/hora

O JavaScript representa datas internamente como inteiros que especificam o número de milissegundos desde (ou antes) da meia-noite de 1º de janeiro de 1970, horário UTC. Números inteiros tão grandes quanto 8.640.000.000.000.000 são suportados, portanto, o JavaScript não ficará sem milissegundos por mais de 270.000 anos.

Para qualquer objeto Date, o método `getTime()` retorna esse valor interno e o método `setTime()` o define. Portanto, você pode adicionar 30 segundos a uma data com um código como este, por exemplo:

```
d.setTime(d.getTime() + 30000);
```

Esses valores de milissegundos às vezes são chamados de carimbos de data/hora e, às vezes, é útil trabalhar com eles diretamente, em vez de com objetos Date. O método estático Date.now() retorna a hora atual como um carimbo de data/hora e é útil quando você deseja medir quanto tempo seu código leva para ser executado:

```
deixe startTime = Date.now(); reticulateSplines(); Faça
alguma operação demorada endTime = Date.now();
console.log('A reticulação de spline levou ${endTime -
startTime}ms.');
```

CARIMBOS DE DATA/HORA DE ALTA RESOLUÇÃO

Os carimbos de data/hora retornados por Date.now() são medidos em milissegundos. Um milissegundo é na verdade um tempo relativamente longo para um computador e, às vezes, você pode querer medir o tempo decorrido com maior precisão. A função performance.now() permite isso: ela também retorna um carimbo de data/hora baseado em milissegundos, mas o valor de retorno não é um número inteiro, portanto, inclui frações de milissegundo. O valor retornado por performance.now() não é um carimbo de data/hora absoluto como o valor Date.now(). Em vez disso, ele simplesmente indica quanto tempo se passou desde que uma página da web foi carregada ou desde que o processo Node foi iniciado.

O objeto de desempenho faz parte de uma API de desempenho maior que não é definida pelo ECMAScript standard, mas é implementada por navegadores da Web e pelo Node. Para usar o objeto de desempenho no Node, você deve importá-lo com:

```
const { performance } = require("perf_hooks");
```

Permitir tempo de alta precisão na web pode permitir que sites inescrupulosos identifiquem os visitantes, portanto, os navegadores (principalmente o Firefox) podem reduzir a precisão do performance.now() por padrão. Como desenvolvedor web, você deve ser capaz de reativar o tempo de alta precisão de alguma forma (como definindo privacy.reduceTimerPrecision como false no Firefox).

11.4.2 Aritmética de data

Os objetos de data podem ser comparados com os operadores de comparação padrão `<`, `<=`, `>` e `>=` do JavaScript. E você pode subtrair um objeto Date de outro para determinar o número de milissegundos entre as duas datas. (Isso funciona porque a classe Date define um método `valueOf()` que retorna um carimbo de data/hora.)

Se você quiser adicionar ou subtrair um número especificado de segundos, minutos ou horas de uma data, geralmente é mais fácil simplesmente modificar as tampas de tempo demonstradas no exemplo anterior, quando adicionamos 30 segundos a uma data. Essa técnica se torna mais complicada se você quiser adicionar dias, e não funciona por meses e anos, pois eles têm números variados de dias. Para fazer aritmética de data envolvendo dias, meses e anos, você pode usar `setDate()`, `setMonth()` e `setYear()`. Aqui, por exemplo, está o código que adiciona três meses e duas semanas à data atual:

```
let d = new Date(); d.setMonth(d.getMonth() +  
3, d.getDate() + 14);
```

Os métodos de configuração de data funcionam corretamente mesmo quando transbordam. Quando adicionamos três meses ao mês atual, podemos acabar com um valor maior que 11 (que representa dezembro). O `setMonth()` lida com isso incrementando o ano conforme necessário. Da mesma forma, quando definimos o dia do mês para um valor maior que o número de dias no mês, o mês é incrementado adequadamente.

11.4.3 Formatação e análise de strings de data

Se você estiver usando a classe Date para realmente acompanhar datas e horas (em vez de apenas medir intervalos de tempo), é provável que você

precisa exibir datas e horas para os usuários do seu código. A classe Date define vários métodos diferentes para converter objetos Date em cadeias de caracteres. Aqui estão alguns exemplos:

```
let d = new Date(2020, 0, 1, 17, 10, 30); 17:10:30 do dia de
Ano Novo 2020d.toString()// => "Wed Jan 01 2020 17:10:30
GMT-0800(Horário Padrão do Pacífico)"d.toUTCString()// =>
"Thu, 02 Jan 2020 01:10:30 GMT"d.toLocaleDateString()// =>
"1/1/2020": 'en-US' localized.toLocaleTimeString()// =>
"5:10:30 PM": 'en-US' localized.toISOString()// => "2020-01-
02T01:10:30.000Z"
```

Esta é uma lista completa dos métodos de formatação de string da classe Date:

toString()

Esse método usa o fuso horário local, mas não formata a data e a hora de maneira compatível com a localidade.

toUTCString()

Esse método usa o fuso horário UTC, mas não formata a data de maneira sensível à localidade.

toISOString()

Este método imprime a data e a hora no formato padrão ano-mês-dia horas:minutos:segundos.ms do padrão ISO-8601. A letra "T" separa a parte da data da saída da parte da hora da saída. A hora é expressa em UTC, e isso é indicado com a letra "Z" como a última letra da saída.

toLocaleString()

Esse método usa o fuso horário local e um formato apropriado para a localidade do usuário.

toDateString()

Esse método formata apenas a parte da data da Data e omite a hora. Ele usa o fuso horário local e não faz formatação apropriada para localidade.

toLocaleDateString()

Esse método formata apenas a data. Ele usa o fuso horário local e um formato de data apropriado para localidade.

toTimeString()

Esse método formata apenas a hora e omite a data. Ele usa o fuso horário local, mas não formata a hora de maneira sensível à localidade.

toLocaleTimeString()

Esse método formata a hora de maneira sensível à localidade e usa o fuso horário local.

Nenhum desses métodos de data para cadeia de caracteres é ideal ao formatar datas e horas a serem exibidas aos usuários finais. Consulte §11.7.2 para obter uma técnica de formatação de data e hora com reconhecimento de localidade e uso mais geral.

Por fim, além desses métodos que convertem um objeto Date em astring, há também um método estático Date.parse() que usa uma string como argumento, tenta analisá-la como data e hora e retorna atimestamp representando essa data. Date.parse() é capaz de analisar as mesmas strings que o construtor Date() pode e é garantido que seja capaz de analisar a saída de toISOString(), toUTCString() e toString().

11.5 Classes de erro

As instruções `throw` e `catch` do JavaScript podem lançar e capturar qualquer valor JavaScript, incluindo valores primitivos. Não há tipo de exceção que deve ser usado para sinalizar erros. No entanto, o JavaScript define uma classe `Error`, e é tradicional usar instâncias de `Error` ou uma subclasse ao sinalizar um erro com `throw`. Um bom motivo para usar um objeto `Error` é que, quando você cria um `Error`, ele captura o estado da pilha JavaScript e, se a exceção não for capturada, o rastreamento de pilha será exibido com a mensagem de erro, o que o ajudará a depurar o problema. (Observe que o rastreamento de pilha mostra onde o objeto `Error` foi criado, não onde a instrução `throw` o lança. Se você sempre criar o objeto logo antes de jogá-lo com `throw new Error()`, isso não causará nenhuma confusão.)

Os objetos de erro têm duas propriedades: `message` e `name` e o método `toString()`. O valor da propriedade `message` é o valor que você passou para o construtor `Error()`, convertido em uma string, se necessário. Para objetos de erro criados com `Error()`, a propriedade `name` é sempre `"Error"`. O método `toString()` simplesmente retorna o valor da propriedade `name` seguido por dois pontos e espaço e o valor da propriedade `message`.

Embora não faça parte do padrão ECMAScript, os navegadores Node e allmodern também definem uma propriedade de pilha em objetos `Error`. O valor dessa propriedade é uma cadeia de caracteres de várias linhas que contém um rastreamento de pilha da pilha de chamadas JavaScript no momento em que o objeto `Error` foi criado. Esta pode ser uma informação útil para registrar quando um inesperado

erro é detectado.

Além da classe Error, o JavaScript define várias subclasses que ele usa para sinalizar tipos específicos de erros definidos pelo ECMAScript. Essas subclasses são EvalError, RangeError, ReferenceError, SyntaxError, TypeError e URIError. Você pode usar essas classes de erro em seu próprio código se elas parecerem apropriadas. Como a classe base Error, cada uma dessas subclasses tem um construtor que recebe um único argumento de mensagem. E as instâncias de cada uma dessas subclasses têm uma propriedade name cujo valor é o mesmo que o construtor.name.

Você deve se sentir à vontade para definir suas próprias subclasses de erro que melhor encapsulam as condições de erro de seu próprio programa. Observe que você não está limitado às propriedades name e message. Se você criar uma subclasse, poderá definir novas propriedades para fornecer detalhes do erro. Se você estiver escrevendo um analisador, por exemplo, pode ser útil definir uma classe ParseError com propriedades de linha e coluna que especificam o local exato da falha de análise. Ou, se você estiver trabalhando com solicitações HTTP, talvez queira definir uma classe HTTPError que tenha uma propriedade status que contenha o código de status HTTP (como 404 ou 500) da solicitação com falha.

Por exemplo:

```
class HTTPError extends Error {  
    constructor(status, statusText, url) {  
        super('${status} ${statusText}:  
${url}'); this.status = status;  
        this.statusText = statusText;
```

```
    this.url = url; }

get name() { return "HTTPError"; }

let error = new HTTPError(404, "Não
encontrado","http://example.com/");
error.status// => 404error.message// =>
"404 Não
encontrado:http://example.com/"error.name/
/ => "HTTPError"
```

11.6 Serialização e análise JSON

Quando um programa precisa salvar dados ou transmitir dados através de uma conexão de rede para outro programa, ele deve converter suas estruturas de dados na memória em uma sequência de bytes ou caracteres que podem ser salvos ou transmitidos e, posteriormente, ser analisados para restaurar as estruturas de dados originais na memória. Esse processo de conversão de estruturas de dados em fluxos de bytes ou caracteres é conhecido como serialização (ou marshaling ou mesmo decapagem).

A maneira mais fácil de serializar dados em JavaScript usa um formato de serialização conhecido como JSON. Este acrônimo significa "JavaScript ObjectNotation" e, como o nome indica, o formato usa a sintaxe literal do objeto e do array JavaScript para converter estruturas de dados que consistem em objetos e arrays em strings. JSON suporta números primitivos e strings e também os valores true, false e null, bem como arrays e objetos criados a partir desses valores primitivos. O JSON não oferece suporte a outros tipos de JavaScript, como Map, Set, RegExp, Date ou matrizes tipadas. No entanto, provou ser um formato de dados notavelmente versátil

e é de uso comum mesmo com programas não baseados em JavaScript.

O JavaScript suporta serialização e desserialização JSON com as duas funções `JSON.stringify()` e `JSON.parse()`, que recuperamos brevemente em §6.8. Dado um objeto ou array (aninhado arbitrariamente profundamente) que não contém nenhum valor não serializável como RegEx objects ou arrays tipados, você pode serializar o objeto simplesmente passando-o para `JSON.stringify()`. Como o nome indica, o valor de retorno de esta função é uma string. E dada uma string retornada por `JSON.stringify()`, você pode recriar a estrutura de dados original passando a string para `JSON.parse()`:

```
let o = {s: "", n: 0, a: [verdadeiro, falso, nulo]};  
let s = JSON.stringify(o); // s == '{"s": "", "n": 0, "a": [  
    true, false, null]}'  
let copy = JSON.parse(s); // copy ==  
{s: "", n: 0, a: [true, false, null]}
```

Se deixarmos de fora a parte em que os dados serializados são salvos em um arquivo ou enviados pela rede, podemos usar esse par de funções como uma maneira um tanto ineficiente de criar uma cópia profunda de um objeto:

Faça uma cópia profunda de qualquer objeto ou matriz serializável

```
function deepcopy(o) {  
    return JSON.parse(JSON.stringify(o));}
```

JSON É UM SUBCONJUNTO DO JAVASCRIPT

Quando os dados são serializados para o formato JSON, o resultado é um código-fonte JavaScript válido para uma expressão que é avaliada como uma cópia da estrutura de dados original. Se você prefixar uma string JSON com `var data =` e passar o resultado para `eval()`, obterá uma cópia da estrutura de dados original atribuída aos dados variáveis. Você

nunca deve fazer isso, no entanto, porque é uma enorme falha de segurança - se um invasor puder injetar código JavaScript arbitrário em um arquivo JSON, ele poderá fazer seu programa executar seu código. É mais rápido e seguro usar apenas JSON.parse() para decodificar dados formatados em JSON.

Às vezes, o JSON é usado como um formato de arquivo de configuração legível. Se você estiver editando manualmente um arquivo JSON, observe que o formato JSON é um subconjunto muito restrito do JavaScript. Comentários não são permitidos e os nomes de propriedade devem ser colocados entre aspas duplas, mesmo quando o JavaScript não exigir isso.

Normalmente, você passa apenas um único argumento para JSON.stringify() e JSON.parse(). Ambas as funções aceitam um segundo argumento opcional que nos permite estender o formato JSON, e estes são descritos a seguir. JSON.stringify() também recebe um terceiro argumento opcional que discutiremos primeiro. Se você quiser que sua string formatada em JSON seja legível por humanos (se estiver sendo usada como um arquivo de configuração, por exemplo), você deve passar null como o segundo argumento e passar um número ou string como o terceiro argumento. Este terceiro argumento informa a JSON.stringify() que ele deve formatar os dados em várias linhas recuadas. Se o terceiro argumento for um número, ele usará esse número de espaços para cada nível de recuo. Se o terceiro argumento for uma string de espaço em branco (como '\t'), ele usará essa string para cada nível de recuo.

```
let o = {s: "teste", n: 0};
JSON.stringify(o, null, 2) => '{\n      "s": "teste", \n      "n":\n      0\n}'
```

JSON.parse() ignora o espaço em branco, portanto, passar um terceiro argumento para JSON.stringify() não afeta nossa capacidade de converter o

string de volta em uma estrutura de dados.

11.6.1 Personalizações JSON

Se `JSON.stringify()` for solicitado a serializar um valor que não é suportado nativamente pelo formato JSON, ele verificará se esse valor tem um método `toJSON()` e, em caso afirmativo, ele chamará esse método e, em seguida, stringificará o valor de retorno no lugar do valor original. Objetos de data implementam `toJSON()`: ele retorna a mesma string que o método `toISOString()`. Isso significa que, se você serializar um objeto que inclui uma data, a data será automaticamente convertida em uma cadeia de caracteres para você. Quando você analisa a cadeia de caracteres serializada, a estrutura de dados recriada não será exatamente a mesma com a qual você começou, pois ela terá uma cadeia de caracteres em que o objeto original tinha uma data.

Se você precisar recriar objetos Date (ou modificar o objeto analisado de qualquer outra forma), você pode passar uma função "reviver" como o segundo argumento para `JSON.parse()`. Se especificada, essa função "reviver" é invocada uma vez para cada valor primitivo (mas não os objetos ou matrizes que contêm esses valores primitivos) analisados da cadeia de caracteres de entrada. A função é invocada com dois argumentos. O primeiro é um nome de propriedade, um nome de propriedade de objeto ou um índice de matriz convertido em uma cadeia de caracteres. O segundo argumento é o valor primitivo dessa propriedade de objeto ou elemento de matriz. Além disso, a função é invocada como um método do objeto ou matriz que contém o valor primitivo, então você pode se referir a aquele contendo objeto com a palavra-chave `this`.

O valor retornado da função reviver torna-se o novo valor da propriedade nomeada. Se ele retornar seu segundo argumento, a propriedade irá

permanecem inalterados. Se retornar undefined, a propriedade nomeada será excluída do objeto ou array antes que JSON.parse() retorne ao usuário.

Como exemplo, aqui está uma chamada para JSON.parse() que usa uma reviverfunction para filtrar algumas propriedades e recriar objetos Date:

```
let dados = JSON.parse(texto, função(chave, valor) {  
  Remova todos os valores cujo nome de propriedade comece  
  com um sublinhado  
  if (key[0] === "_") retorna indefinido;  
  
  Se o valor for uma string no formato de data ISO  
  8601 converte-o em uma data.  
  if (typeof valor === "string" &&  
    /^\\d\\d\\d\\d-\\d\\d-  
    \\d\\dT\\d\\d:\\d\\d:\\d\\.\\d\\.\\d\\d\\dZ$/.test(valor)) {  
    return new Date(value);}  
  
  Caso contrário, retorne o valor  
  inalterado valor de retorno;});
```

Além do uso de toJSON() descrito anteriormente, JSON.stringify() também permite que sua saída seja personalizada ignorando uma matriz ou uma função como o segundo argumento opcional.

Se uma matriz de strings (ou números - eles são convertidos em strings) for passada como o segundo argumento, elas serão usadas como os nomes das propriedades do objeto (ou elementos da matriz). Qualquer propriedade cujo nome não esteja na matriz será omitida da stringificação. Além disso, a string virada incluirá propriedades na mesma ordem em que aparecem na matriz (o que pode ser muito útil ao escrever testes).

Se você passar uma função, ela será uma função replacer - efetivamente o inverso da função reviver opcional que você pode passar para JSON.parse(). Se especificado, a função replacer é invocada para cada valor a ser stringificado. O primeiro argumento para a função replacer é o nome da propriedade do objeto ou o índice da matriz do valor dentro desse objeto, e o segundo argumento é o próprio valor. A função replacer é invocada como um método do objeto ou matriz que contém o valor a ser stringificado. O valor retornado da função replacer é stringificado no lugar do valor original. Se o substituto retornar undefined ou não retornar nada, esse valor (e seu elemento de matriz ou propriedade de objeto) será omitido da stringificação.

```
Especifique quais campos serializar e em que ordem  
serializá-los inlet text = JSON.stringify(address,  
["city", "state", "country"]);
```

```
Especifique uma função de substituição que omita RegExp-  
valuepropertieslet json = JSON.stringify(o, (k, v) => v  
instanceof RegExp ?undefined : v);
```

As duas chamadas JSON.stringify() aqui usam o segundo argumento de maneira benigna, produzindo uma saída serializada que pode ser desserializada sem exigir uma função especial de reviver. Em geral, porém, se você definir um método toJSON() para um tipo ou se usar uma função replacer que realmente substitui valores não serializáveis por serializáveis, normalmente precisará usar uma função reviver personalizada com JSON.parse() para recuperar sua estrutura de dados original. Se você fizer isso, você deve entender que está definindo um formato de dados personalizado e sacrificando a portabilidade e a compatibilidade com um grande ecossistema de

Ferramentas e linguagens compatíveis com JSON.

11.7 A API de Internacionalização

A API de internacionalização JavaScript consiste em três classes `Intl.NumberFormat`, `Intl.DateTimeFormat` e `Intl.Collator` que nos permitem formatar números (incluindo valores monetários e porcentagens), datas e horas de maneiras apropriadas à localidade e comparar cadeias de caracteres de maneiras apropriadas à localidade. Essas classes não fazem parte do padrão ECMAScript, mas são definidas como parte do padrão ECMA402 e são bem suportadas por navegadores da web. A API Internacional também é suportada no Node, mas no momento da redação deste artigo, os binários pré-construídos do Node não são fornecidos com os dados de localização necessários para fazê-los funcionar com localidades diferentes do inglês dos EUA. Portanto, para usar essas classes com o Node, pode ser necessário baixar um pacote de dados separado ou usar uma compilação personalizada do Node.

Uma das partes mais importantes da internacionalização é a exibição de texto que foi traduzido para o idioma do usuário. Existem várias maneiras de conseguir isso, mas nenhuma delas está dentro do escopo da `IntlAPI` descrita aqui.

11.7.1 Formatando números

Usuários de todo o mundo esperam que os números sejam formatados de maneiras diferentes. Os pontos decimais podem ser pontos ou vírgulas. Os separadores de milhares podem ser vírgulas ou pontos e não são usados a cada três dígitos em todos os lugares. Algumas moedas são divididas em centésimos, outras em milésimos e algumas não têm subdivisões. Por último, embora o

chamados de "algarismos arábicos" de 0 a 9 são usados em muitos idiomas, isso não é universal, e os usuários em alguns países esperam ver números escritos usando os dígitos de seus próprios scripts.

A classe Intl.NumberFormat define um método format() que leva em conta todas essas possibilidades de formatação. O construtor usa dois argumentos. O primeiro argumento especifica a localidade para a qual o número deve ser formatado e o segundo é um objeto que especifica mais detalhes sobre como o número deve ser formatado. Se o primeiro argumento for omitido ou indefinido, a localidade do sistema (que presumimos ser a localidade preferida do usuário) será usada. Se o primeiro argumento for uma string, ele especifica uma localidade desejada, como "en-US" (inglês usado nos Estados Unidos), "fr" (francês) ou "zh-Hans-CN" (chinês, usando o sistema de escrita Han simplificado, na China). O primeiro argumento também pode ser uma matriz de cadeias de caracteres de localidade e, nesse caso, Intl.NumberFormat escolherá a mais específica que é bem suportada.

O segundo argumento para o construtor Intl.NumberFormat(), ifspecified, deve ser um objeto que define uma ou mais das seguintes propriedades:

estilo

Especifica o tipo de formatação numérica necessária. O padrão é "decimal". Especifique "porcentagem" para formatar um número como uma porcentagem ou especifique "moeda" para especificar um número como uma quantia em dinheiro.

moeda

Se style for "moeda", essa propriedade será necessária para especificar o código de moeda ISO de três letras (como "USD" para dólares americanos ou "GBP" para libras esterlinas) da moeda desejada.

moedaExibir

Se style for "currency", essa propriedade especificará como a moeda é exibida. O valor padrão "símbolo" usa um símbolo de moeda se a moeda tiver um. O valor "código" usa o código ISO de três letras e o valor "nome" soletra o nome da moeda em formato longo.

useGrouping

Defina essa propriedade como false se você não quiser que os números tenham separadores de milhares (ou seus equivalentes apropriados para a localidade).

minimumIntegerDigits

O número mínimo de dígitos a serem usados para exibir a parte inteira do número. Se o número tiver menos dígitos do que isso, ele será preenchido à esquerda com zeros. O valor padrão é 1, mas você pode usar valores tão altos quanto 21.

minimumFractionDigits, maximumFractionDigits

Essas duas propriedades controlam a formatação da parte fracionária do número. Se um número tiver menos dígitos fracionários do que o mínimo, ele será preenchido com zeros à direita. Se tiver mais do que o máximo, a parte fracionária será arredondada. Os valores legais para ambas as propriedades estão entre 0 e 20. O default minimum é 0 e o default maximum é 3, exceto ao formatar valores monetários, quando o comprimento da parte fracionária varia dependendo da moeda especificada.

minimumSignificant Digits, maximumSignificant Digits

Essas propriedades controlam o número de dígitos significativos usados na formatação de um número, tornando-as adequadas na formatação de dados científicos, por exemplo. Se especificadas, essas propriedades substituem as propriedades de dígitos inteiros e fracionários listadas anteriormente. Os valores legais estão entre 1 e 21.

Depois de criar um objeto Intl.NumberFormat com a localidade e as opções desejadas, use-o passando um número para seu método format(), que retorna uma cadeia de caracteres formatada adequadamente. Por exemplo:

```
let euros = Intl.NumberFormat("es", {style: "currency", currency: "EUR"}); euros.format(10)// => "10,00 €": dez euros, formatação em espanhol
```

```
let libras = Intl.NumberFormat("pt-br", {style: "moeda", moeda: "GBP"}); libras.format(1000) // => "£1.000,00": mil libras, formatação em inglês
```

Um recurso útil de Intl.NumberFormat (e das outras classes Intl também) é que seu método format() está vinculado ao objeto NumberFormataao qual ele pertence. Então, em vez de definir uma variável que se refere ao objeto de formatação e, em seguida, invocar o método format() nisso, você pode simplesmente atribuir o método format() a uma variável e usá-lo como se fosse uma função autônoma, como neste exemplo:

```
let dados = [0,05, 0,75, 1]; let formatData = Intl.NumberFormat(indefinido, { style: "percent", minimumFractionDigits: 1, maximumFractionDigits: 1}).format; data.map(formatData) => ["5,0%", "75,0%", "100,0%"]: em
```

en-US local

Alguns idiomas, como o árabe, usam seu próprio script para dígitos decimais:

```
let árabe = Intl.NumberFormat("ar",
{useGrouping:false}).format; árabe(1234567890)////
=> "١٢٣٤٥٦٧٨٩."
```

Outros idiomas, como o hindi, usam um script que tem seu próprio conjunto de dígitos, mas tendem a usar os dígitos ASCII de 0 a 9 por padrão. Se você quiser substituir o script padrão usado para dígitos, adicione -u-nu- ao código do idioma e siga-o com um nome de script abreviado. Você pode formatar números com agrupamento no estilo indiano e dígitos Devanagari como este, por exemplo:

```
Hindi tardio = INTL.Numberformat("ele-em-você-para-
deva"). formato; Hindi(1234567890)// => "1,23,45,67,890"
```

-u- em uma localidade especifica que o que vem a seguir é um Unicode extension.nu é o nome da extensão para o sistema de numeração, e deva é a abreviação de Devanagari. O padrão Intl API define nomes para vários outros sistemas de numeração, principalmente para os idiomas índicos do sul e sudeste da Ásia.

11.7.2 Formatação de datas e horas

A classe Intl.DateTimeFormat é muito parecida com a classe Intl.NumberFormat. O construtor Intl.DateTimeFormat() usa os mesmos dois argumentos que Intl.NumberFormat(): uma localidade ou matriz de localidades e um objeto de opções de formatação. E a maneira como você usa um

`Intl.DateTimeFormat` é chamando seu método `format()` para converter um objeto `Date` em uma string.

Conforme mencionado na seção 11.4, a classe `Date` define os métodos `simpletoLocaleDateString()` e `toLocaleTimeString()` que produzem uma saída apropriada para a localidade do usuário. Mas esses métodos não oferecem nenhum controle sobre quais campos da data e hora são exibidos. Talvez você queira omitir o ano, mas adicionar um dia da semana ao formato de data. Você quer que o mês seja representado numericamente ou escrito por nome? A classe `Intl.DateTimeFormat` fornece controle refinado sobre o que é gerado com base nas propriedades no objeto `options` que é passado como o segundo argumento para o construtor. Observe, no entanto, que `Intl.DateTimeFormat` nem sempre pode exibir exatamente o que você solicita. Se você especificar opções para formatar horas e segundos, mas omitir minutos, verá que o formatador exibe os minutos de qualquer maneira. A ideia é que você use o objeto de opções para especificar quais campos de data e hora você gostaria de apresentar ao usuário e como você gostaria que eles fossem formatados (por nome ou por número, por exemplo), então o formatador procurará um formato apropriado para a localidade que mais se aproxime do que você pediu.

As opções disponíveis são as seguintes. Especifique apenas as propriedades dos campos de data e hora que você gostaria que aparecessem na saída formatada.

ano

Use "numérico" para um ano completo de quatro dígitos ou "2 dígitos" para uma abreviação de dois dígitos.

mês

Use "numérico" para um número possivelmente curto como "1" ou "2 dígitos" para uma representação numérica que sempre tem dois dígitos, como "01". Use "long" para um nome completo como "January", "short" para um nome abreviado como "Jan" e "narrow" para um nome altamente abreviado como "J" que não é garantido como único.

dia

Use "numérico" para um número de um ou dois dígitos ou "2 dígitos" para um número de dois dígitos para o dia do mês.

Semana

Use "long" para um nome completo como "Monday", "short" para um nome anabbreviado como "Mon" e "narrow" para um nome altamente abreviado como "M" que não é garantido como único.

era

Essa propriedade especifica se uma data deve ser formatada com anera, como CE ou BCE. Isso pode ser útil se você estiver formatando datas de muito tempo atrás ou se estiver usando um calendário japonês. Os valores legais são "longo", "curto" e "estreito".

hora, minuto, segundo

Essas propriedades especificam como você gostaria que o tempo fosse exibido. Use "numérico" para um campo de um ou dois dígitos ou "2 dígitos" para forçar números de um dígito a serem preenchidos à esquerda com um 0.

fuso horário

Essa propriedade especifica o fuso horário desejado para o qual a data deve ser formatada. Se omitido, o fuso horário local será usado. As implementações sempre reconhecem "UTC" e também podem reconhecer nomes de fuso horário da IANA (Internet Assigned Numbers Authority),

como "América / Los_Angeles".

nome_do_fuso_horário

Essa propriedade especifica como o fuso horário deve ser exibido em uma data ou hora formatada. Use "long" para um nome de fuso horário totalmente escrito e "short" para um fuso horário abreviado ou numérico.

hora12

Essa propriedade booleana especifica se o tempo de 12 horas deve ou não ser usado. O padrão é dependente da localidade, mas você pode substituí-lo por thisproperty.

hourCycle

Essa propriedade permite que você especifique se meia-noite é gravada como 0 horas, 12 horas ou 24 horas. O padrão é dependente da localidade, mas você pode substituir o padrão por essa propriedade. Observe que hour12 tem precedência sobre essa propriedade. Use o valor "h11" para especificar que meia-noite é 0 e a hora antes da meia-noite é 23h. Use "h12" para especificar que meia-noite é 12. Use "h23" para especificar que meia-noite é 0 e a hora antes da meia-noite é 23. E use "h24" para especificar que meia-noite é 24.

Aqui estão alguns exemplos:

```
let d = new Date("2020-01-02T13:14:15Z"); 2 de janeiro de  
2020, 13:14:15 UTC
```

```
Sem opções, obtemos uma data numérica básica  
formatIntl.DateTimeFormat("en-US").format(d) // =>  
"1/2/2020"Intl.DateTimeFormat("fr-FR").format(d) // => "02/01/2020"
```

```
Soletrado dia da semana e mês  
let opts = { weekday: "long",  
month: "long", year: "numeric", day: "numeric" };Intl.  
DateTimeFormat("pt-BR", opts).format(d) // => "Quinta-feira,"
```

```
2 de janeiro de 2020" Intl.DateTimeFormat("pt-BR",  
opts).format(d) // => "Quinta-feira, 2 de janeiro de 2020"
```

```
A hora em Nova York, para um canadense francófono = {  
hour: "numeric", minute: "2-digit",  
timeZone:"America/New_York" };Intl. DateTimeFormat("fr-CA", opts).format(d) // => "8 h 14"
```

Intl.DateTimeFormat pode exibir datas usando calendários diferentes do calendário juliano padrão baseado na era cristã. Embora algumas localidades possam usar um calendário não cristão por padrão, você sempre pode especificar explicitamente o calendário a ser usado adicionando -u-ca- à localidade e seguindo com o nome do calendário. Possíveis nomes de calendário incluem "budista", "chinês", "copta", "etíope", "gregory", "hebraico", "indiano", "islâmico", "iso8601", "japonês" e "persa". Continuando o exemplo anterior, podemos determinar o ano em vários calendários não cristãos:

```
let opts = { ano: "numérico", era: "curto"
};Intl.DateTimeFormat("pt-br", opta).format(d) //  
=> "2020 AD"Intl.DateTimeFormat("pt-pt-ao-ca-  
iso8601", opts).format(d) //  
=> "2020 AD"Intl.DateTimeFormat("pt-br  
  
=> "5780 AM"Intl.DateTimeFormat("pt-br-u-ca-buddhist",  
opts).format(d) //  
=> "2563 BE"Intl.DateTimeFormat("pt-br-u-ca-islamic",  
opts).format(d) //  
=> "1441 AH"Intl.DateTimeFormat("pt-br-u-ca-persian",  
opts).format(d) //  
=> "1398 AP"Intl.DateTimeFormat("pt-br-u-ca-indian",  
opts).format(d) //  
=> "1941 Saka"Intl.DateTimeFormat("pt-br-u-ca-  
chinese", opts).format(d) //  
=> "36 78"Intl.DateTimeFormat("pt-br-u-ca-japanese",  
opts).format(d) //  
=> "2 Reiwa"
```

11.7.3 Comparando strings

O problema de classificar strings em ordem alfabética (ou alguma "ordem de agrupamento" mais geral para scripts não alfábéticos) é mais desafiador do que os falantes de inglês geralmente percebem. O inglês usa um alfabeto relativamente pequeno, sem letras acentuadas, e temos o benefício de uma codificação de caracteres (ASCII, desde que incorporada ao Unicode) cujos valores numéricos correspondem perfeitamente à nossa cadeia de caracteres padrão ordem de classificação. As coisas não são tão simples em outras línguas. O espanhol, por exemplo, trata ñ como uma letra distinta que vem depois de n e antes de o. O lituano alfabetiza Y antes de J, e o galês trata dígrafos como CH e DD como letras únicas, com CH vindo depois de C e DD classificando depois de D.

Se você deseja exibir strings para um usuário em uma ordem que ele encontrará natural, não é suficiente usar o método `sort()` em uma matriz de strings. Mas se você criar um objeto `Intl.Collator`, poderá passar o método `compare()` desse objeto para o método `sort()` para executar a classificação apropriada da localidade das strings. Os objetos `Intl.Collator` podem ser configurados para que o método `compare()` execute comparações sem distinção entre maiúsculas e minúsculas ou mesmo comparações que considerem apenas a letra base e ignorem acentos e outros diacríticos.

Como `Intl.NumberFormat()` e `Intl.DateTimeFormat()`, o construtor `Intl.Collator()` recebe dois argumentos. O primeiro especifica uma localidade ou uma matriz de localidades, e o segundo é um objeto opcional cujas propriedades especificam exatamente que tipo de comparação de cadeia de caracteres deve ser feita. As propriedades suportadas são estas:

uso

Essa propriedade especifica como o objeto collator deve ser usado. O valor padrão é "classificar", mas você também pode especificar "pesquisar". A ideia é que, ao classificar cadeias de caracteres, você normalmente deseja um agrupador que diferencie o maior número possível de cadeias de caracteres para produzir uma ordenação confiável. Mas ao comparar duas strings, algumas localidades podem querer uma comparação menos estrita que ignora acentos, por exemplo.

sensibilidade

Essa propriedade especifica se o ordenador é sensível a letras maiúsculas e minúsculas ao comparar cadeias de caracteres. O valor "base" provoca comparações que ignoram maiúsculas e minúsculas e acentos, considerando apenas a letra base para cada caractere.

(Note-se, no entanto, que algumas línguas consideram certos caracteres acentuados como letras de base distintas.) "sotaque" considera acentos em comparações, mas ignora o caso. "case" considera case e ignora acentos. E "variante" realiza comparações estritas que consideram tanto o caso quanto os acentos. O valor padrão para essa propriedade é "variant" quando o uso é "sort". Se o uso for "pesquisa", a sensibilidade padrão dependerá da localidade.

ignorePontuação

Defina essa propriedade como true para ignorar espaços e pontuação ao comparar cadeias de caracteres. Com essa propriedade definida como true, as cadeias de caracteres "anyone" e "anyone", por exemplo, serão consideradas iguais.

numérico

Defina essa propriedade como true se as cadeias de caracteres que você está comparando forem inteiras ou contiverem inteiros e você quiser que elas sejam classificadas em ordem numérica em vez de ordem alfabética. Com esta opção definida, a string "Versão 9" será classificada antes de "Versão 10", por exemplo.

caso em primeiro lugar

Essa propriedade especifica qual letra maiúscula e minúscula deve vir primeiro. Se você especificar "upper", então "A" será classificado antes de "a". E se você especificar "inferior", então "a" será classificado antes de "A". Em ambos os casos, observe que as variantes maiúsculas e minúsculas da mesma letra estarão próximas umas das outras na ordem de classificação, que é diferente da ordenação Unicodelexicographic (o comportamento padrão do método Array sort()) em que todas as letras maiúsculas ASCII vêm antes de todas as letras minúsculas ASCII. O padrão para essa propriedade é localedependente e as implementações podem ignorar essa propriedade e não permitir que você substitua a ordem de classificação de maiúsculas e minúsculas.

Depois de criar um objeto Intl.Collator para a localidade desejada andoptions, você pode usar seu método compare() para comparar duas strings. Esse método retorna um número. Se o valor retornado for menor que zero, a primeira string virá antes da segunda string. Se for maior que zero, a primeira string virá depois da segunda string. E ifcompare() retorna zero, então as duas strings são iguais no que diz respeito a thiscollator.

Esse método compare() que usa duas strings e retorna um número menor que, igual ou maior que zero é exatamente o que o método Arraysort() espera para seu argumento opcional. Além disso, Intl.Collator vincula automaticamente o método compare() à sua instância, para que você possa passá-lo diretamente para sort() sem precisar escrever uma função wrapper e invocá-la por meio do objeto collator. Aqui estão alguns exemplos:

Um comparador básico para classificar na localidade do usuário.// Nunca classifique strings legíveis por humanos sem passar algo assim:

```
const collator = new Intl.Collator().compare; ["a", "z",
"A", "Z"].sort(agrupador)// => ["a", "A", "z", "Z"]
```

Os nomes dos arquivos geralmente incluem números, portanto, devemos classificar thoseespecially

```
const filenameOrder = new Intl.Collator(undefined, { numeric:true }).compare;
["página10", "página9"].sort(filenameOrder)// =>
["página9", "página10"]
```

Encontre todas as strings que correspondem vagamente a um destino

```
stringconst fuzzyMatcher = new Intl.Collator(undefined, {
  sensibilidade: "base", ignorePontuação:
  verdadeiro}).compare; let strings = ["comida", "tolo",
  "Føø Bar"]; strings.findIndex(s => fuzzyMatcher(s,
  "foobar") === 0)
// => 2
```

Algumas localidades têm mais de uma ordem de ordenação possível. Na Alemanha, por exemplo, as listas telefónicas utilizam uma ordem de ordenação ligeiramente mais fonética do que os dicionários. Em Espanha, antes de 1994, "ch" e "ll" eram tratados como letras separadas, pelo que este país tem agora uma ordem de ordenação moderna e uma ordem de ordenação tradicional. E na China, a ordem de agrupamento pode ser baseada em codificações de caracteres, no radical base e nos traços de cada caractere ou na romanização Pinyin de caracteres. Essas variantes de agrupamento não podem ser selecionadas por meio do argumento de opções Intl.Collator, mas podem ser selecionadas adicionando -u-co- à cadeia de caracteres de localidade e adicionando o nome da variante desejada. Use "de-DE-u-co-phonebk" para pedidos de livros telefônicos na Alemanha, por exemplo, e "zh-TW-u-co-pinyin" para pedidos de Pinyin em Taiwan.

Antes de 1994, CH e LL eram tratados como letras separadas

```
emSpainconst modernSpanish = Intl.Collator("es-ES").compare;
```

```
const traditionalSpanish = Intl.Collator("es-ES-u-co-trad").compare; let palavras = ["luz", "chama", "como", "menino"]; words.sort(modernSpanish)// => ["menino", "como", "chama", "luz"]words.sort(traditionalSpanish) // => ["curtir", "menino","luz", "chama"]
```

11.8 A API do console

Você viu a função `console.log()` usada ao longo deste livro: em navegadores da web, ele imprime uma string na guia "Console" do painel de ferramentas do desenvolvedor do navegador, o que pode ser muito útil durante a depuração. No Node, `console.log()` é uma função de saída de uso geral e imprime seus argumentos no fluxo `stdout` do processo, onde normalmente aparece para o usuário em uma janela de terminal como saída do programa.

A API do console define várias funções úteis além de `toconsole.log()`. A API não faz parte de nenhum padrão ECMAScript, mas é suportada por navegadores e pelo Node e foi formalmente escrita e padronizada em <https://console.spec.whatwg.org>.

A API do Console define as seguintes funções:

console.log()

Esta é a mais conhecida das funções do console. Ele converte seus argumentos em strings e os envia para o console. Ele inclui espaços entre os argumentos e inicia uma nova linha após a saída de todos os argumentos.

console.debug(), console.info(), console.warn(), console.error()

Essas funções são quase idênticas a `console.log()`. `InNode`, `console.error()` envia sua saída para o fluxo `stderr` em vez do fluxo `stdout`, mas as outras funções são aliases de `console.log()`. Nos navegadores, as mensagens de saída geradas por cada uma dessas funções podem ser prefixadas por um ícone que indica seu nível ou gravidade, e o console do desenvolvedor também pode permitir que os desenvolvedores filtrem as mensagens do console por nível.

`console.assert()`

Se o primeiro argumento for verdadeiro (ou seja, se a afirmação for aprovada), então esta função não faz nada. Mas se o primeiro argumento for falso ou outro valor falso, os argumentos restantes serão impressos como se tivessem sido passados para `console.error()` com um prefixo "Assertionfailed". Observe que, ao contrário das funções típicas de `assert()`, `console.assert()` não lança uma exceção quando uma asserção falha.

`console.clear()`

Esta função limpa o console quando isso é possível. Isso funciona em navegadores e no Node quando o Node está exibindo sua saída para um terminal. Se a saída do Node tiver sido redirecionada para um arquivo ou um pipe, no entanto, chamar essa função não terá efeito.

`console.table()`

Esta função é um recurso notavelmente poderoso, mas pouco conhecido, para produzir saída tabular, e é particularmente útil em programas Node, que precisam produzir uma saída que resume data. `console.table()` tenta exibir seu argumento em forma tabular (embora, se não puder fazer isso, ele o exiba usando a formatação regular `console.log()`). Isso funciona melhor quando o argumento é uma matriz relativamente curta de objetos e todos os objetos na matriz têm o mesmo conjunto (relativamente pequeno) de propriedades. Nesse caso, cada objeto na matriz é formatado como uma linha da tabela,

e cada propriedade é uma coluna da tabela. Você também pode passar uma matriz de nomes de propriedade como um segundo argumento opcional para especificar o conjunto desejado de colunas. Se você passar um objeto em vez de uma matriz de objetos, a saída será uma tabela com uma coluna para nomes de propriedade e uma coluna para valores de propriedade. Ou, se esses valores de propriedade forem objetos, seus nomes de propriedade se tornarão colunas na tabela.

`console.trace()`

Essa função registra seus argumentos como `console.log()` faz e, além disso, segue sua saída com um rastreamento de pilha. No Node, a saída vai para `stderr` em vez de `stdout`.

`console.count()`

Essa função usa um argumento de cadeia de caracteres e registra essa cadeia de caracteres, seguido pelo número de vezes que ela foi chamada com essa cadeia de caracteres. Isso pode ser útil ao depurar um manipulador de eventos, por exemplo, se você precisar acompanhar quantas vezes o manipulador de eventos foi acionado.

`console.countReset()`

Essa função usa um argumento de cadeia de caracteres e redefine o contador para essa cadeia de caracteres.

`console.group()`

Essa função imprime seus argumentos no console como se tivessem sido passados para `console.log()` e, em seguida, define o estado interno do console para que todas as mensagens subsequentes do console (até a próxima chamada `console.groupEnd()`) sejam recuadas em relação à mensagem que acabou de imprimir. Isso permite que um grupo de mensagens relacionadas seja agrupado visualmente com recuo. Em navegadores da web, o console do desenvolvedor normalmente permite que as mensagens agrupadas sejam recolhidas e expandidas como um grupo. Os argumentos para

`console.group()` são normalmente usados para fornecer um nome explicativo para o grupo.

`console.groupCollapsed()`

Esta função funciona como `console.group()`, exceto que em navegadores da web, o grupo será "recolhido" por padrão e as mensagens que ele contém serão ocultadas, a menos que o usuário clique para expandir o grupo. No Node, essa função é sinônimo de `console.group()`.

`console.groupEnd()`

Esta função não aceita argumentos. Ele não produz nenhuma saída própria, mas encerra o recuo e o agrupamento causados pela chamada mais recente para `console.group()` ou `console.groupCollapsed()`.

`console.time()`

Essa função usa um único argumento de string, anota a hora em que foi chamada com essa string e não produz nenhuma saída.

`console.timeLog()`

Essa função usa uma cadeia de caracteres como seu primeiro argumento. Se essa string tiver sido passada anteriormente para `console.time()`, ela imprimirá essa string seguida pelo tempo decorrido desde a chamada `console.time()`. Se houver argumentos adicionais to `console.timeLog()`, eles serão impressos como se tivessem sido passados para `console.log()`.

`console.timeEnd()`

Essa função usa um único argumento de cadeia de caracteres. Se esse argumento tiver sido passado anteriormente para `console.time()`, ele imprimirá esse argumento e o tempo decorrido. Depois de chamar `console.timeEnd()`, não é mais legal chamar

```
console.timeLog() sem primeiro chamar  
console.time() novamente.
```

11.8.1 Saída formatada com console

As funções de `console` que imprimem seus argumentos como `console.log()` têm um recurso pouco conhecido: se o primeiro argumento for uma cadeia de caracteres que inclui `%s`, `%i`, `%d`, `%f`, `%o`, `%0` ou `%c`, esse primeiro argumento será tratado como cadeia de caracteres de formato e os valores dos argumentos subsequentes serão substituídos na cadeia de caracteres no lugar das sequências `%` de dois caracteres.⁶

Os significados das sequências são os seguintes:

`%s`

O argumento é convertido em uma cadeia de caracteres.

`%i` e `%d`

O argumento é convertido em um número e, em seguida, truncado em um inteiro.

`%f`

O argumento é convertido em um número

`%o` e `%0`

O argumento é tratado como um objeto e os nomes e valores das propriedades são exibidos. (Em navegadores da Web, essa exibição é normalmente interativa e os usuários podem expandir e recolher propriedades para explorar uma estrutura de dados aninhada.) `%o` e `%O` exibem detalhes do objeto. A variante maiúscula usa um formato de saída dependente da implementação que é considerado mais útil para desenvolvedores de software.

`%C`

Em navegadores da Web, o argumento é interpretado como uma string de CSSstyles e usado para estilizar qualquer texto que se segue (até o próximo `%csequence` ou o final da string). No Node, a sequência `%c` e seu argumento correspondente são simplesmente ignorados.

Observe que muitas vezes não é necessário usar uma string de formato com as funções do console: geralmente é fácil obter uma saída adequada simplesmente passando um ou mais valores (incluindo objetos) para a função e permitindo que a implementação os exiba de maneira útil. Como exemplo, observe que, se você passar um objeto Error para `console.log()`, ele será impresso automaticamente junto com seu rastreamento de pilha.

11.9 APIs de URL

Como o JavaScript é tão comumente usado em navegadores e servidores da Web, é comum que o código JavaScript precise manipular URLs. A classe URL analisa URLs e também permite a modificação (adicionando parâmetros de pesquisa ou alterando caminhos, por exemplo) de URLs existentes. Ele também lida adequadamente com o complicado tópico de escapar e desescapar os vários componentes de um URL.

A classe URL não faz parte de nenhum padrão ECMAScript, mas funciona em Node e em todos os navegadores da Internet, exceto o Internet Explorer. É padronizado em <https://url.spec.whatwg.org>.

Crie um objeto URL com o construtor `URL()`, passando uma string `absoluteURL` como argumento. Ou passe uma URL relativa como o primeiro argumento e a URL absoluta à qual ele é relativo como o segundo

argumento. Depois de criar o objeto URL, suas várias propriedades permitem que você consulte versões sem escape das várias partes do URL:

```
let url = nova URL("https://example.com:8000/path/name?  
q=term#fragment"); url.href// =>  
"https://example.com:8000/path/name?  
q=term#fragment"url.origin// =>  
"https://example.com:8000"url.protocol// =>  
"https://"url.host// => "example.com:8000"url.hostname// =>  
"example.com"url.port// => "8000"url.pathname// =>  
"/path/name"url.search// => "?q=term"url.hash// =>  
"#fragment"
```

Embora não seja comumente usado, os URLs podem incluir um nome de usuário ou umusername e senha, e a classe de URL também pode analisar esses URLcomponents:

```
let url = nova  
URL("ftp://admin:1337!@ftp.example.com/"); url.href// =>  
"ftp://admin:1337!@ftp.example.com/"url.origin// =>  
"ftp://ftp.example.com"url.username// =>  
"admin"url.password// => "1337!"
```

A propriedade origin aqui é uma combinação simples do protocolo URL e do host (incluindo a porta, se for especificada). Como tal, é uma propriedade somente leitura. Mas cada uma das outras propriedades demonstradas no exemplo anterior é leitura/gravação: você pode definir qualquer uma dessas propriedades para definir a parte correspondente da URL:

```
let url = nova URL("https://example.com"); Comece com nosso
```

```
serverurl.pathname =
  "api/pesquisa";                                Adicione um caminho para
uma API endpointurl.search
  = "q=test";                                     Adicionar uma consulta
  url.paraString
g()          => "https://example.com/api/search?q=test"
```

Uma das características importantes da classe URL é que ela adiciona pontuação corretamente e escapa caracteres especiais em URLs quando necessário:

```
let url = nova URL("https://example.com");
url.pathname = "caminho com espaços"; url.search =
  "q=foo#bar"; url.pathname// =>
  "/path%20with%20spaces" url.search// => "?"
  q=foo%23bar" url.href// =>"https://example.com/path%20with%20spaces?
  q=foo%23bar"
```

A propriedade href nesses exemplos é especial: ler href é equivalente a chamar `toString()`: ele reagrupa todas as partes da URL na forma de string canônica da URL. E definir href como uma nova string executa novamente o analisador de URL na nova string como se você tivesse chamado o construtor `URL()` novamente.

Nos exemplos anteriores, usamos a propriedade `search` para referir-se a toda a parte de consulta de uma URL, que consiste nos caracteres de um ponto de interrogação no final da URL ou no primeiro caractere hash. Às vezes, é suficiente tratar isso apenas como uma propriedade `singleURL`. Muitas vezes, no entanto, as solicitações HTTP codificam os valores de vários campos de formulário ou vários parâmetros de API na parte de consulta de uma URL usando o formato `application/x-www-form-urlencoded`. Nesse formato, a parte de consulta da URL é um ponto de interrogação

seguido por um ou mais pares de nome/valor, que são separados uns dos outros por e comercial. O mesmo nome pode aparecer mais de uma vez, resultando em um parâmetro de pesquisa nomeado com mais de um valor.

Se você quiser codificar esses tipos de pares de nome/valor na parte de consulta de uma URL, a propriedade searchParams será mais útil do que a propriedade search. A propriedade de pesquisa é uma cadeia de caracteres de leitura/gravação que permite obter e definir toda a parte da consulta do URL. A propriedade searchParams é uma referência somente leitura ao objeto aURLSearchParams, que tem uma API para obter, definir, adicionar, excluir e classificar os parâmetros codificados na parte de consulta da URL:

```
let url = nova URL("https://example.com/search");
url.search// => "": sem consulta
yeturl.searchParams.append("q", "term"); Adicione um
parâmetrourl.search// => "?q=term" url.searchParams.set("q",
"x");// Altere o valor deste parâmetrourl.search// => "?"
q=x"url.searchParams.get("q")// => "x": consulte o
parâmetro valueurl.searchParams.has("q")// => true: there
is aq parameterurl.searchParams.has("p")// => false: there
isno p parameterurl.searchParams.append("opts", "1");
Adicione outro searchparameterurl.search// => "?"
q=x&opts=1"url.searchParams.append("opts", "&"); Adicione
outro valor para o mesmo nomeurl.search// => "?"
q=x&opts=1&opts=%26": note
escapeurl.searchParams.get("opts")// => "1": o primeiro
valor
```

```

urlSearchParams.getAll("opta")           => ["1", "&"]: todos
values urlSearchParams.
sort();                                Coloque parâmetros em
ordem
alfabética url.search
opts=1&opts=%26&q=x" urlSearchParams.set("opts", "y");
param url.
search
searchParams é iterável[...]
urlSearchParams]                      => [{"opta": "y"}, {
["q",
"x"]]] urlSearchParams.delete("opta"); Exclua as opções
param url.
search                                         => "?q=x"
url.href                                     // =>
"https://example.com/search?q=x"

```

O valor da propriedade `searchParams` é um objeto `URLSearchParams`. Se você deseja codificar parâmetros de URL em uma string de consulta, você pode criar um objeto `URLSearchParams`, anexar parâmetros, convertê-lo em uma string e defini-lo na propriedade de pesquisa de um URL:

```

let url = nova URL("http://example.com"); let params = new
URLSearchParams(); params.append("q", "termo");
params.append("opta", "exato"); params.toString() // =>
"q=term&opts=exact" url.search = params; url.href // =>
"http://example.com/?q=term&opts=exact"

```

11.9.1 Funções de URL herdadas

Antes da definição da API de URL descrita anteriormente, houve várias tentativas de oferecer suporte ao escape e ao cancelamento de URL na linguagem JavaScript principal. A primeira tentativa foi a definição global

`escape()` e `unescape()`, que agora estão obsoletas, mas ainda amplamente implementadas. Eles não devem ser usados.

Quando `escape()` e `unescape()` foram descontinuados, o ECMAScript introduziu dois pares de funções globais alternativas:

encodeURI() e decodeURI()

`encodeURI()` recebe uma string como argumento e retorna uma nova string na qual caracteres não ASCII mais certos caracteres ASCII (como espaço) são escapados. `decodeURI()` reverte o processo. Os caracteres que precisam ser escapados são primeiro convertidos em sua codificação UTF-8 e, em seguida, cada byte dessa codificação é substituído por uma sequência %xxescape, em que xx são dois dígitos hexadecimais. Como `encodeURI()` se destina a codificar URLs inteiros, ele não escapa caracteres separadores de URL, como /, ?, e #. Mas isso significa que `encodeURI()` não pode funcionar corretamente para URLs que possuem esses caracteres em seus vários componentes.

encodeURIComponent() e decodeURIComponent()

Esse par de funções funciona exatamente como `encodeURI()` e `decodeURI()`, exceto que elas se destinam a escapar de componentes individuais de um URI, portanto, também escapam caracteres como /, ?, e # que são usados para separar esses componentes. Essas são as funções de URL legadas mais úteis, mas lembre-se de que `encodeURIComponent()` escapará / caracteres em um nome de caminho que você provavelmente não deseja escapar. E converterá espaços em um parâmetro de consulta para %20, mesmo que os espaços devam ser escapados com um + nessa parte de uma URL.

O problema fundamental com todas essas funções herdadas é que elas procuram aplicar um único esquema de codificação a todas as partes de uma URL quando o

o fato é que diferentes partes de um URL usam codificações diferentes. Se você deseja um URL formatado e codificado corretamente, a solução é simplesmente usar a classe de URL para todas as manipulações de URL que você fizer.

11.10 Temporizadores

Desde os primórdios do JavaScript, os navegadores da web definiram duas funções — `setTimeout()` e `setInterval()` — que permitem que os programas solicitem ao navegador que invoque uma função após um determinado período de tempo ou que invoque a função repetidamente em um intervalo especificado. Essas funções nunca foram padronizadas como parte da linguagem principal, mas funcionam em todos os navegadores e no Node e são uma parte de fato da biblioteca padrão JavaScript.

O primeiro argumento para `setTimeout()` é uma função e o segundo argumento é um número que especifica quantos milissegundos devem decorrer antes que a função seja invocada. Após o período de tempo especificado (e talvez um pouco mais se o sistema estiver ocupado), a função será invocada sem argumentos. Aqui, por exemplo, estão as chamadas `threesetTimeout()` que imprimem mensagens do console após um segundo, dois segundos e três segundos:

```
setTimeout(() => { console.log("Pronto..."); }, 1000);
setTimeout(() => { console.log("set..."); }, 2000);
setTimeout(() => { console.log("ir!"); }, 3000);
```

Observe que `setTimeout()` não espera que o tempo decorra antes de retornar. Todas as três linhas de código neste exemplo são executadas quase instantaneamente, mas nada acontece até que 1.000 milissegundos decorram.

Se você omitir o segundo argumento para `setTimeout()`, o padrão será 0. Isso não significa, no entanto, que a função especificada é invocada imediatamente. Em vez disso, a função é registrada para ser chamada "o mais rápido possível". Se um navegador estiver particularmente ocupado lidando com a entrada do usuário ou outros eventos, pode levar 10 milissegundos ou mais antes que a função seja invocada.

`setTimeout()` registra uma função a ser invocada uma vez. Às vezes, essa função chamará `setTimeout()` para agendar outra invocação em um momento futuro. Se você quiser invocar uma função repetidamente, no entanto, geralmente é mais simples usar `setInterval()`. `setInterval()` usa os mesmos dois argumentos que `setTimeout()`, mas invoca a função repetidamente toda vez que o número especificado de milissegundos (aproximadamente) tiver decorrido.

Tanto `setTimeout()` quanto `setInterval()` retornam um valor. Se você salvar esse valor em uma variável, poderá usá-lo posteriormente para cancelar a execução da função passando-a para `clearTimeout()` ou `clearInterval()`. O valor retornado é normalmente um número em navegadores da Web e é um objeto em Node. O tipo real não importa e você deve tratá-lo como um valor opaco. A única coisa que você pode fazer com esse valor é passá-lo para `clearTimeout()` para cancelar a execução de uma função registrada com `setTimeout()` (supondo que ainda não tenha sido invocada) ou para interromper a execução repetida de uma função registrada com `setInterval()`.

Aqui está um exemplo que demonstra o uso de `setTimeout()`, `setInterval()` e `clearInterval()` para exibir um

relógio digital com a API do console:

```
Uma vez por segundo: limpe o console e imprima o
currenttimelet clock = setInterval(() => {

  console.clear(); console.log(new
Date().toLocaleTimeString());}, 1000);

Após 10 segundos: pare o código repetido
acima.setTimeout(() => { clearInterval(clock); }, 10000);
```

Veremos setTimeout() e setInterval() novamente quando abordarmos a programação assíncrona no Capítulo 13.

11.11 Resumo

Aprender uma linguagem de programação não é apenas dominar a gramática. É igualmente importante estudar a biblioteca padrão para que você esteja familiarizado com todas as ferramentas que são fornecidas com a linguagem. Este capítulo documentou a biblioteca padrão do JavaScript, que inclui:

- Estruturas de dados importantes, como Set, Map e matrizes tipadas. As classes Date e URL para trabalhar com datas e URLs. A gramática de expressão regular do JavaScript e sua classe RegExp para correspondência de padrões textuais.
- Biblioteca de internacionalização do JavaScript para formatação de datas, hora e números e para classificação de strings. O objeto JSON para serializar e desserializar estruturas de dados simples e o objeto de console para registrar mensagens.

-
- 1 Nem tudo documentado aqui é definido pela especificação da linguagem JavaScript:^{algumas} das classes e funções documentadas aqui foram implementadas pela primeira vez na web navegadores e, em seguida, adotados pelo Node, tornando-os membros de fato da biblioteca padrão JavaScript.
 - 2 Essa ordem de iteração previsível é outra coisa sobre os conjuntos JavaScript que os programadores do Python podem achar surpreendente.
 - 3 As matrizes tipadas foram introduzidas pela primeira vez no JavaScript do lado do cliente quando os navegadores da Web adicionaram suporte para gráficos WebGL. O que há de novo no ES6 é que eles foram elevados a um recurso de linguagem principal.
 - 4 Exceto dentro de uma classe de caracteres (colchetes), onde \b corresponde ao backspacecharacter.
 - 5 Analisar URLs com expressões regulares não é uma boa ideia. Consulte §11.9 para obter um analisador de URL mais robusto.
 - 6 Os programadores C reconhecerão muitas dessas sequências de caracteres da função printf().

Capítulo 12. Iteradores e geradores

Objetos iteráveis e seus iteradores associados são um recurso do ES6 que vimos várias vezes ao longo deste livro. Arrays (incluindo TypedArrays) são iteráveis, assim como strings e objetos Set e Map. Isso significa que o conteúdo dessas estruturas de dados pode ser iterado — em loop — com o loop for/of, como vimos em §5.4.4:

```
seja soma = 0; for(let i of [1,2,3]) { // Loop uma vez para
cada um desses valores
soma +=
i;}soma// =>
6
```

Os iteradores também podem ser usados com o ... para expandir ou "espalhar" um objeto iterável em um inicializador de array ou invocação de função, como vimos em §7.1.2:

```
let chars = [..."abcd"]; caracteres == ["a", "b", "c",
"d"]let dados = [1, 2, 3, 4, 5]; Math.max(... dados)//
=> 5
```

Os iteradores podem ser usados com atribuição de desestruturação:

```
let purpleHaze = Uint8Array.of(255, 0, 255,
128); seja [r, g, b, a] = purpleHaze; a == 128
```

Quando você itera um objeto Map, os valores retornados são [key,

value], que funcionam bem com a atribuição de desestruturação em um loop for/of:

```
let m = new Map([["um", 1], ["dois", 2]]); for(let [k,v] de m) console.log(k, v); Registros «um 1» e «dois 2»
```

Se você quiser iterar apenas as chaves ou apenas os valores em vez de opares, você pode usar os métodos keys() e values():

```
[... m] => [["um", 1], ["dois", 2]]: padrão  
iteração[...  
m.entries() => [["um", 1], ["dois", 2]]: entradas()  
método é o mesmo[... m.keys()]/> ["um", "dois"]: keys()  
metodiera apenas mapear chaves[... m.values()]/> [1, 2]:  
o método values() itera os valores do JustMap
```

Finalmente, várias funções e construtores integrados que são comumente usados com objetos Array são realmente gravados (no ES6 e posterior) para aceitar iteradores arbitrários. O construtor Set() é uma dessas APIs:

```
As strings são iteráveis, então os dois conjuntos são os mesmos:  
new Set("abc") // => new Set(["a", "b", "c"])
```

Este capítulo explica como os iteradores funcionam e demonstra como criar suas próprias estruturas de dados que são iteráveis. Depois de explicar os iteradores básicos, este capítulo aborda os geradores, um novo recurso poderoso do ES6 que é usado principalmente como uma maneira particularmente fácil de criar iteradores.

12.1 Como funcionam os iteradores

O loop for/of e o operador spread funcionam perfeitamente com objetos iteráveis, mas vale a pena entender o que realmente está acontecendo para fazer a iteração funcionar. Existem três tipos separados que você precisa entender para entender a iteração em JavaScript. Primeiro, existem os objetos iteráveis: são tipos como Array, Set e Map que podem ser iterados. Em segundo lugar, há o próprio objeto iterador, que executa a iteração. E terceiro, há o objeto de resultado da iteração que contém o resultado de cada etapa da iteração.

Um objeto iterável é qualquer objeto com um método iterador especial que retorna um objeto iterador. Um iterador é qualquer objeto com um método next() que retorna um objeto de resultado de iteração. E um resultobject de iteração é um objeto com propriedades chamadas value e done. Para iteratean objeto iterável, primeiro chame seu método iterator para obter um iteratorobject. Em seguida, você chama o método next() do objeto iterador repetidamente até que o valor retornado tenha sua propriedade done definida como true. O complicado disso é que o método iterador de um iterableobject não tem um nome convencional, mas usa o Symbol.iterator como seu nome. Portanto, um simples loop for/of sobre objeto iterável também pode ser escrito da maneira mais difícil, assim:

```
let iterável = [99]; let iterador =
iterable[Symbol.iterator](); for(let resultado =
iterator.next(); !resultado.feito; resultado
=iterator.next()) {
    console.log(resultado.valor)
}
```

O objeto iterador dos tipos de dados iteráveis internos é iterável.

(Ou seja, ele tem um método chamado `Symbol.iterator` que apenas retorna.) Isso é ocasionalmente útil em códigos como o seguinte quando você deseja iterar por meio de um iterador "parcialmente usado":

```
let list = [1,2,3,4,5]; let iter =
list[Symbol.iterator](); let head =
iter.next().value;// head == 1let tail = [...,
iter];// cauda == [2,3,4,5]
```

12.2 Implementando objetos iteráveis

Objetos iteráveis são tão úteis no ES6 que você deve considerar tornar seus próprios tipos de dados iteráveis sempre que eles representarem algo que possa ser iterado. As classes `Range` mostradas nos Exemplos 9-2 e 9-3 no Capítulo 9 eram iteráveis. Essas classes usavam funções geradoras para se tornarem iteráveis. Documentaremos os geradores mais adiante neste capítulo, mas primeiro, implementaremos a classe `Range` mais uma vez, tornando-a iterável sem depender de um gerador.

Para tornar uma classe iterável, você deve implementar um método whose name é o `Symbol.iterator`. Esse método deve retornar um objeto iterador que tenha um método `next()`. E o método `next()` deve retornar um objeto de resultado de iteração que tenha uma propriedade `value` e/ou uma propriedade booleana `done`. O exemplo 12-1 implementa uma classe `Range`, iterável, e demonstra como criar objetos iteráveis, iterador e `iterationresult`.

Exemplo 12-1. Uma classe numérica iterável Range

```
/*
* Um objeto Range representa um intervalo de números {x: de
<= x<= a}
```

* Range define um método `has()` para testar se um determinado número é um membro* do intervalo. Range é iterável e itera todos os inteiros dentro do intervalo.

```
*/class  
Intervalo {  
    construtor (de, para) {  
        this.from = de;  
        this.to = para;}
```

```
Faça um intervalo agir como um conjunto de números has(x) {  
  return typeof x === "number" && this.from <= x && x <=  
  this.to; }
```

Representação de string de retorno do intervalo usando setnotation

```
toString() { return '{ x | ${this.from} ≤ x ≤ ${this.to}}';  
}
```

Torne um Range iterável retornando um objeto iterador.//
Observe que o nome desse método é um símbolo especial, não
uma string.

```
[Símbolo.iterator]() {
```

Cada instânciado iterador deve iterar o intervalo independentemente de

Outros. Portanto, precisamos de uma variável de estado para rastrear nosso localização no

iteração. Começamos no primeiro inteiro inteiro >= from.let next = Math.ceil(this.from); // Este é o próximo valor que devolvemos

deixe durar = this.to; *Nós não vamos voltar*
qualquer coisa > isso

```
        retornar {  
    objeto iterador
```

Este método next() é o que torna este um objeto iterator.

Ele deve retornar um objeto de resultado do iterador.next().

return (próximo <= último) Se não tivermos

Último valor retornado ainda

? { value: next++ } // retorna o próximo valor
e incrementá-lo

: { feito: verdadeiro }; caso contrário, indique

que terminamos.

},

Por conveniência, fazemos o próprio iterador iterável.

```
[Símbolo.iterator](){ return this; }};}}
```

```
for(let x de new Range(1,10)) console.log(x); Logs números de  
1 a 10 [... new Range(-2,2)]// => [-2, -1, 0, 1, 2]
```

Além de tornar suas classes iteráveis, pode ser bastante útil definir funções que retornam valores iteráveis. Considere estas alternativas baseadas em iteráveis para os métodos map() e filter() das matrizes JavaScript:

Retorna um objeto iterável que itera o resultado de aplicando f()// a cada valor da fonte

```
iterablefunction map(iterable, f) {
```

```
let iterador = iterable[Symbol.iterator](); return {  
  Este objeto é iterador e iterável  
  [Símbolo.iterator](){ return this;  
  },next() {  
    let v = iterator.next();  
    if (v.done) {  
      retornar v;  
    } else {  
      return { valor: f(v.valor) };  
    }  
  }  
};}
```

Mapeie um intervalo de inteiros para seus quadrados e converta em uma matriz

```
[... map(novo intervalo(1,4), x => => [1, 4, 9, 16]
x*x)]
Retorna um objeto iterável que filtra o
especificadoiterável, // iterando apenas os elementos
para os quais o predicadore retorna truefunction
filter(iterable, predicado) {

    let iterador = iterable[Symbol.iterator](); return {
        // Este objeto é iterador e iterável
        [Símbolo.iterator]() { return this;
        },next() {
            for(;;) {
                let v = iterator.next(); if
                    (v.done || predicado(v.valor)) {
                return v;}}}}};}
```

```
Filtre um intervalo para que fiquemos apenas com
números pares [... filter(new Range(1,10), x => x % 2
== 0)]// =>[2,4,6,8,10]
```

Uma característica importante dos objetos iteráveis e iteradores é que eles são inherentemente preguiçosos: quando a computação é necessária para calcular o próximo valor, essa computação pode ser adiada até que o valor seja realmente necessário. Suponha, por exemplo, que você tenha uma sequência muito longa de texto que deseja tokenizar em palavras separadas por espaço. Você poderia simplesmente usar o método split() de sua string, mas se você fizer isso, então toda a string deve ser processada antes que você possa usar até mesmo a primeira palavra. E você acaba alocando muita memória para o returnedarray e todas as strings dentro dele. Aqui está uma função que permite que você iterar preguiçosamente as palavras de uma string sem mantê-las todas na memória de uma vez (no ES2020, essa função seria muito mais fácil de

implementar usando o método matchAll() de retorno do iterador descrito em §11.3.2):

```
palavra(s) de função {
    var r = /\s+$/g;                                Combine um ou
    mais espaços ou fim
    Comece a lastIndex dependendo da posição
    de onde a string espaço

    retornar {                                         Retornar um
    objeto iterador iterável                         Isso nos torna
        [Símbolo.iterador](): {                      iterável
            return
            this;},next() {                           Isso nos torna um
        }
    Iterador
        let start = r.lastIndex;                     Retome onde o
        última partida terminou
        if (início < s.length) {                   Se não estivermos
            terminado
            let correspondência =                  Combine o próximo
            limite da                            r.exec(s);
            palavra                          if (correspondência) {   Se encontrássemos um,
            devolver a palavra                    return { valor: s.substring(iniciar,
            match.index) };
            }}return { done: true
            };
    que terminamos
};};}
```

Caso contrário, digamos

```
[... palavras(" abc  Ghi! ")] // => ["abc", "def", "ghi!"]
```

12.2.1 "Fechando" um iterador: o método de retorno

Imagine uma variante JavaScript (do lado do servidor) do iterador words()

que, em vez de usar uma string de origem como argumento, pega o nome de um arquivo, abre o arquivo, lê as linhas dele e itera as palavras dessas linhas. Na maioria dos sistemas operacionais, os programas que abrem arquivos para ler precisam se lembrar de fechar esses arquivos quando terminarem de ler, portanto, esse iterador hipotético certamente fechará o arquivo depois que o método next() retornar a última palavra nele.

Mas os iteradores nem sempre são executados até o fim: um loop for/of pode ser encerrado com uma quebra ou retorno ou por uma exceção. Da mesma forma, quando um iterador é usado com atribuição de desestruturação, o método next() é chamado apenas vezes suficientes para obter valores para cada uma das variáveis especificadas. O iterador pode ter muito mais valores que poderia retornar, mas eles nunca serão solicitados.

Se nosso iterador hipotético de palavras em um arquivo nunca for executado até o fim, ele ainda precisará fechar o arquivo aberto. Por esse motivo, iteratorobjects podem implementar um método return() para acompanhar o método next(). Se a iteração parar antes que next() tenha retornado o resultado da aniteração com a propriedade done definida como true (mais comumente porque você deixou um loop for/of mais cedo por meio de uma instrução break), o interpretador verificará se o objeto iterador tem um método return(). Se esse método existir, o interpretador o invocará com noarguments, dando ao iterador a chance de fechar arquivos, liberar memória e limpar depois de si mesmo. O método return() deve retornar um objeto de resultado do iterador. As propriedades do objeto são ignoradas, mas é um erro retornar um valor que não seja de objeto.

O loop for/of e o operador de propagação são recursos realmente úteis do

JavaScript, portanto, ao criar APIs, é uma boa ideia usá-las quando possível. Mas ter que trabalhar com um objeto iterável, seu objeto iterador e os objetos de resultado do iterador torna o processo um pouco complicado. Felizmente, os geradores podem simplificar drasticamente a criação de iteradores personalizados, como veremos no restante deste capítulo.

12.3 Geradores

Um gerador é um tipo de iterador definido com uma nova e poderosa sintaxe ES6; É particularmente útil quando os valores a serem iterados não são os elementos de uma estrutura de dados, mas o resultado de um cálculo.

Para criar um gerador, você deve primeiro definir uma função geradora. Uma função geradora é sintaticamente como uma função JavaScript regular, mas é definida com a palavra-chave `function*` em vez de `function`. (Tecnicamente, esta não é uma nova palavra-chave, apenas um * após a função de palavra-chave e antes do nome da função.) Quando você invoca uma função geradora, ela não executa o corpo da função, mas retorna um objeto gerador. Esse objeto gerador é um iterador. Chamar seu método `next()` faz com que o corpo da função geradora seja executado desde o início (ou qualquer que seja sua posição atual) até atingir uma instrução `yield`. `yield` é novo no ES6 e é algo como uma instrução `return`. O valor da instrução `yield` torna-se o valor retornado pela chamada `next()` no iterador. Um exemplo torna isso mais claro:

Uma função geradora que produz o conjunto de um dígito (base 10) primes.function oneDigitPrimes() { // Invocar esta função não executa o código*

```
    rendimento 2;           mas apenas retorna um gerador
  objeto. Vocações
    rendimento 3;           o método next() desse método
  O gerador funciona
    rendimento 5;           o código até um rendimento
  declaração fornece
    rendimento 7;           o valor retornado para o
  next().}                  next()
```

Quando invocamos a função geradora, obtemos um generatorlet
`primes = oneDigitPrimes();`

Um gerador é um objeto iterador que itera os valores produzidos
`primes.next().value// =>`
`2primes.next().value// => 3primes.next().value// =>`
`5primes.next().value// => 7primes.next().done// =>`
`true`

Os geradores têm um método Symbol.iterator para torná-los primos iteráveis
`[Symbol.iterator]// => primos`

Podemos usar geradores como outros tipos iteráveis
`[... oneDigitPrimes()]// =>`
`[2,3,5,7]let soma = 0; for(let prime de`
`oneDigitPrimes()) soma += primo; soma// => 17`

Neste exemplo, usamos uma instrução `function*` para definir um gerador. Como funções regulares, no entanto, também podemos definir geradores em forma de expressão. Mais uma vez, apenas colocamos um asterisco após a palavra-chave `function`:

```
const seq = função*(de,para) {
  for(let i = from; i <= to; i++) produz i;};
  [... seq(3,5)]// => [3, 4, 5]
```

Em classes e literais de objetos, podemos usar a notação abreviada para omitir totalmente a palavra-chave `function` quando definimos métodos. Para definir um gerador neste contexto, simplesmente usamos um asterisco antes do nome do método onde a palavra-chave `function` estaria, se a tivéssemos usado:

```
seja o = {  
  x: 1, y: 2, z: 3, // Um gerador que produz cada uma das  
  chaves deste objeto  
  
  *g() {  
    for(let chave de Object.keys(this)) {  
      chave de rendimento;}}}; [... o.g()]  
// => ["x", "y", "z", "g"]
```

Observe que não há como escrever uma função geradora usando a sintaxe `arrowfunction`.

Os geradores geralmente facilitam particularmente a definição de classes iteráveis. Podemos substituir o método `[Symbol.iterator]()` show inExample 12-1 por uma função geradora *

`[Symbol.iterator]()` muito mais curta que se parece com esta:

```
*[Symbol.iterator]() {  
  for(let x = Math.ceil(this.from); x <= this.to;  
  x++) yield x;}
```

Consulte o Exemplo 9-3 no Capítulo 9 para ver essa função iteradora baseada em gerador no contexto.

12.3.1 Exemplos de geradores

Os geradores são mais interessantes se realmente gerarem os valores que produzem fazendo algum tipo de computação. Aqui, por exemplo, está uma função geradora que produz números de Fibonacci:

```
função* fibonacciSequence() {  
    seja x = 0, y =  
    1; for(;;) {  
        rendimento e; [x,  
        y] = [y, x+y];      Nota: atribuição de desestruturação  
    }  
}
```

Observe que a função geradora fibonacciSequence() aqui tem um loop infinito e produz valores para sempre sem retornar. Se este gerador for usado com o ... spread, ele fará um loop até que a memória se esgote e o programa trave. Com cuidado, é possível usá-lo em um loop for/of, no entanto:

```
Retorna a enésima função numérica  
de Fibonacci fibonacci(n) {  
    for(let f de fibonacciSequence()) {  
        if (n-- <= 0) return  
    f;}Fibonacci(20)// => 10946
```

Esse tipo de gerador infinito se torna mais útil com um gerador take() como este:

```
Produza os primeiros n elementos da  
iterableobjectfunction* especificada* take(n,  
iterable) {  
    let it = iterable[Symbol.iterator](); Obter iterador para
```

```

objeto iterável
    while(n-- > 0) { // Loop n vezes:
        let próximo = it.next(); Obter o próximo item do Iterador.
        if (next.done) retornar; Se não houver mais valores, retornar mais cedo
        else yield next.value; caso contrário, produza o valor}}

```

Uma matriz dos primeiros 5 números de Fibonacci [... take(5, fibonacciSequence())]// => [1, 1, 2, 3, 5]

Aqui está outra função geradora útil que intercala os elementos de vários objetos iteráveis:

```

Dada uma matriz de iteráveis, produza seus elementos inintercalado order.function* zip(... iteráveis) {

    Obter um iterador para cada iterablelet iterators =
    iterables.map(i => i[Symbol.iterator]()); let índice = 0;
    while(iterators.length > 0) { // Embora ainda existam alguns iteradores

        if (índice >= iterators.comprimento) { Se chegássemos
        0 último iterador
            índice = 0;                                Volte para o
            primeiro.
            }let item = iterators[índice].next(); Obter
            próximo item
        do próximo iterador.
            if (item.done) {                            Se isso
            iterador está pronto
                iterators.splice(índice, 1);           em seguida, remova-o
            da matriz.
                }else
                {
                    item de rendimento.valor;          Caso contrário
                    valor iterado
                    índice++;                         produzir o
                e passar para
            o próximo iterador.

```

```
}}}
```

```
Intercalar três objetos iteráveis[...  
zip(oneDigitPrimes(),"ab",[0])] // =>  
[2,"a",0,3,"b",5,7]
```

12.3.2 Rendimento* e Geradores Recursivos

Além do gerador zip() definido no exemplo anterior, pode ser útil ter uma função geradora semelhante que produza os elementos de vários objetos iteráveis sequencialmente, em vez de intercalá-los.

Poderíamos escrever esse gerador assim:

```
função* sequência(... iteráveis) {  
    for(let iterável de iteráveis) {  
        for(let item de iterável) {  
            item de rendimento;}}}
```

```
[... sequência("abc",oneDigitPrimes())] // =>  
["a","b","c",2,3,5,7]
```

Esse processo de produzir os elementos de algum outro objeto iterável é comum o suficiente em funções geradoras que ES6 tem sintaxe especial para isso. A palavra-chave yield* é como yield, exceto que, em vez de produzir um único valor, ela itera um objeto iterável e produz cada um dos valores resultantes. A função geradora sequence() que usamos pode ser simplificada com yield* assim:

```
função* sequência(... iteráveis) {  
    for(let iterável de iteráveis) {
```

```
rendimento* iterável;}}
```



```
[... sequência("abc",oneDigitPrimes())]    // =>
["a", "b", "c", 2, 3, 5, 7]
```

O método array forEach() geralmente é uma maneira elegante de fazer um loop sobre os elementos de um array, então você pode ficar tentado a escrever a função sequence() assim:

```
função* sequência(... iteráveis) {
    iterables.forEach(iterable => yield* iterable );  // Erro
}
```

Isso não funciona, no entanto. yield e yield* só podem ser usados dentro de funções geradoras, mas a função de seta aninhada neste código é uma função regular, não uma função geradora function*, portanto, yield não é permitido.

yield* pode ser usado com qualquer tipo de objeto iterável, incluindo iteráveis implementados com geradores. Isso significa que yield* nos permite definir geradores recursivos, e você pode usar esse recurso para permitir uma iteração simples não recursiva sobre uma estrutura de árvore definida recursivamente, por exemplo.

12.4 Recursos avançados do gerador

O uso mais comum de funções geradoras é criar iteradores, mas a característica fundamental dos geradores é que eles nos permitem pausar a computação, produzir resultados intermediários e, em seguida, retomar o

computação mais tarde. Isso significa que os geradores têm recursos além dos iteradores, e exploramos esses recursos nas seções a seguir.

12.4.1 O valor de retorno de uma função geradora

As funções geradoras que vimos até agora não tiveram returnstatements, ou se tiveram, foram usadas para causar um earlyreturn, não para retornar um valor. No entanto, como qualquer função, uma função geradora pode retornar um valor. Para entender o que acontece neste caso, lembre-se de como funciona a iteração. O valor de retorno da next()function é um objeto que tem uma propriedade value e/ou uma doneproperty. Com iteradores e geradores típicos, se a propriedade value for definida, a propriedade done será indefinida ou falsa. E se feito for verdadeiro, então o valor é indefinido. Mas no caso de um gerador que retorna um valor, a chamada final para next retorna um objeto que tem valor e done definidos. A propriedade value contém o valor retornado da função geradora e a propriedade done é true, indicando que não há mais valores para iterar. Esse valor final é ignorado pelo loop for/of e pelo operador spread, mas está disponível para o código que itera manualmente com chamadas explícitas tonext():

```
function *oneAndDone() {  
  rendimento 1;  
  return  
  "concluído";}
```

O valor retornado não aparece na iteração normal. [...
oneAndDone()]]// => [1]

Mas está disponível se você chamar explicitamente next()let generator = oneAndDone(); generator.next()// => { value: 1, done: false}generator.next()// => { value: "done", done: true}// Se o gerador já estiver pronto, o valor de retorno não será retornado againgenerator.next()// => { value: undefined, done:true }

12.4.2 O valor de uma expressão yield

Na discussão anterior, tratamos yield como uma declaração que assume um valor, mas não tem valor próprio. Na verdade, no entanto, yield é uma expressão e pode ter um valor.

Quando o método next() de um gerador é invocado, a função generator é executada até atingir uma expressão yield. A expressão que segue a palavra-chave yield é avaliada e esse valor se torna o valor return da invocação next(). Neste ponto, a função generator para de ser executada bem no meio da avaliação da expressão yield. Na próxima vez que o método next() do gerador for chamado, o argumento passado para next() se torna o valor da expressão yield que foi pausada. Portanto, o gerador retorna valores para seu chamador com yield, e o chamador passa valores para o gerador com next(). O gerador e o chamador são dois fluxos separados de execução passando valores (e controle) para frente e para trás. O código a seguir ilustra:

```
função* smallNumbers() {  
  console.log("next() invocado pela primeira vez;  
  argumento descartado");  
  seja y1 = rendimento y1 == "b"  
  1;
```

```
console.log("next() invocado uma segunda vez com argumento",y1);
    seja y2 = rendimento 2; y2 == "c"
console.log("next() invocado uma terceira vez com argumento",y2);
    seja y3 = rendimento 3; y3 == "d"
console.log("next() invocado uma quarta vez com argumento",y3);
    retornar 4;}

let g = smallNumbers(); console.log("gerador criado;
nenhum código é executado ainda"); let n1 =
g.next("a");// n1.valor == 1console.log("gerador produzido",
n1.valor); let n2 = g.next("b");// n2.valor ==
2console.log("gerador produzido", n2.valor); let n3 =
g.next("c");// n3.valor == 3console.log("gerador produzido",
n3.valor); let n4 = g.next("d");// n4 == {
valor: 4, feito: true }console.log("gerador retornado",
n4.valor);
```

Quando esse código é executado, ele produz a seguinte saída que demonstra o vaivém entre os dois blocos de código:

```
gerador criado; nenhum código é executado
yetnext() invocado pela primeira vez; argumento
descartado gerador rendeu 1next() invocado uma
segunda vez com argumento bgenerator rendeu
2next() invocado uma terceira vez com argumento
cgenerator rendeu 3next() invocou uma quarta vez
com argumento dgenerator retornou 4
```

Observe a assimetria neste código. A primeira invocação de next() inicia o gerador, mas o valor passado para essa invocação não é acessível ao gerador.

12.4.3 Os métodos `return()` e `throw()` do aGenerator

Vimos que você pode receber valores produzidos ou retornados por uma função gerador. E você pode passar valores para um gerador em execução ignorando esses valores ao chamar o método `next()` do gerador.

Além de fornecer entrada para um gerador com `next()`, você também pode alterar o fluxo de controle dentro do gerador chamando os métodos `itsreturn()` e `throw()`. Como os nomes sugerem, chamar esses métodos em um gerador faz com que ele retorne um valor ou lance uma exceção como se a próxima instrução no gerador fosse um retorno ou lançamento.

Lembre-se do início do capítulo que, se um iterador define um método `return()` e a iteração para mais cedo, o interpretador chama automaticamente o método `return()` para dar ao iterador a chance de fechar arquivos ou fazer outra limpeza. No caso de geradores, você não pode definir um método `return()` personalizado para lidar com a limpeza, mas você pode estruturar o código do gerador para usar uma instrução `try/finally` que garante que a limpeza necessária seja feita (no bloco `finally`) quando o gerador retornar. Ao forçar o gerador a retornar, o método `return()` integrado do gerador garante que o código de limpeza seja executado quando o gerador não for mais usado.

Assim como o método `next()` de um gerador nos permite passar valores arbitrários para um gerador em execução, o método `throw()` de um gerador nos dá uma maneira de enviar sinais arbitrários (na forma de exceções) para

um gerador. Chamar o método `throw()` sempre causa uma exceção dentro do gerador. Mas se a função do gerador for escrita com código de tratamento de exceção apropriado, a exceção não precisa ser fatal, mas pode ser um meio de alterar o comportamento do gerador. Imagine, por exemplo, um gerador de contador que produz uma sequência cada vez maior de números inteiros. Isso pode ser escrito para que uma exceção enviada com `throw()` redefina o contador para zero.

Quando um gerador usa `yield*` para produzir valores de algum objeto `other iterable`, uma chamada para o método `next()` do gerador causa uma chamada para o método `next()` do objeto iterável. O mesmo vale para os métodos `return()` e `throw()`. Se um gerador usar `yield*` em um objeto iterável que tenha esses métodos definidos, chamar `return()` ou `throw()` no gerador fará com que o método `return()` ou `throw()` do iterador seja chamado por sua vez. Todos os iteradores devem ter um método `next()`. Os iteradores que precisam limpar a iteração após a iteração incompleta devem definir um método `return()`. E qualquer iterador pode definir um método `throw()`, embora eu não conheça nenhuma razão prática para fazê-lo.

12.4.4 Uma nota final sobre geradores

Os geradores são uma estrutura de controle generalizada muito poderosa. Eles nos dão a capacidade de pausar um cálculo com `yield` e reiniciá-lo novamente em algum momento posterior arbitrário com um valor de entrada arbitrário. É possível usar geradores para criar um tipo de sistema de threading cooperativo dentro de código JavaScript de thread único. E é possível usar geradores para mascarar partes assíncronas do seu programa para que seu código apareça

sequenciais e síncronas, embora algumas de suas chamadas de função sejam realmente assíncronas e dependam de eventos da rede.

Tentar fazer essas coisas com geradores leva a um código que é incrivelmente difícil de entender ou explicar. Isso foi feito, no entanto, e o único caso de uso realmente prático foi para gerenciar código assíncrono. O JavaScript agora tem palavras-chave assíncronas e `await` (consulte o Capítulo 13) para esse propósito, no entanto, e não há mais nenhuma razão para abusar dos geradores dessa maneira.

12.5 Resumo

Neste capítulo, você aprendeu:

- O loop `for/of` e o ... spread operador trabalhar com iteráveis objetos. Um objeto é iterável se tiver um método com o nome simbólico `[Symbol.iterator]` que retorna um objeto iterador. Um objeto iterador tem um método `next()` que retorna o objeto de resultado de iteração. Um objeto de resultado de iteração tem uma propriedade `value` que contém o próximo valor iterado, se houver. Se a iteração tiver sido concluída, o objeto de resultado deverá ter uma propriedade `done` definida como `true`. Você pode implementar seus próprios objetos iteráveis definindo um método `[Symbol.iterator]()` que retorna um objeto com um método `next()` que retorna objetos de resultado de iteração. Você também pode implementar funções que aceitam `IteratorArguments` e retornam valores de iterador.

- As funções geradoras (funções definidas com `function*` em vez de `function`) são outra maneira de definir iteradores. Quando você invoca uma função geradora, o corpo da função não é executado imediatamente; Em vez disso, o valor de retorno é um objeto iterador iterável. Cada vez que o método `next()` do iterador é chamado, outro pedaço da função do gerador é executado. As funções geradoras podem usar o operador `yield` para especificar os valores retornados pelo iterador. Cada chamada para `next()` faz com que a função do gerador seja executada até a próxima `yieldexpression`. O valor dessa expressão `yield` torna-se então o valor retornado pelo iterador. Quando não há mais expressões `yield`, a função geradora retorna e a iteração é concluída.

Capítulo 13. JavaScript assíncrono

Alguns programas de computador, como simulações científicas e modelos de aprendizado de máquina, são vinculados à computação: eles são executados continuamente, sem pausa, até que tenham calculado seu resultado. A maioria dos programas de computador do mundo real, no entanto, é significativamente assíncrona. Isso significa que muitas vezes eles precisam parar de computar enquanto esperam que os dados cheguem ou que algum evento ocorra. Os programas JavaScript em um navegador da Web geralmente são orientados a eventos, o que significa que eles esperam que o usuário clique ou toque antes de realmente fazer qualquer coisa. E os servidores baseados em JavaScript normalmente esperam que as solicitações do cliente cheguem pela rede antes de fazer qualquer coisa.

Esse tipo de programação assíncrona é comum em JavaScript, e este capítulo documenta três recursos importantes da linguagem que ajudam a facilitar o trabalho com código assíncrono. Promessas, novas no ES6, são objetos que representam o resultado ainda não disponível de uma operação assíncrona. As palavras-chave `async` e `await` foram introduzidas no ES2017 e fornecem uma nova sintaxe que simplifica a programação assíncrona, permitindo que você estruture seu código baseado em `Promise` como se fosse síncrono. Por fim, iteradores assíncronos e o loop `for/await` foram introduzidos no ES2018 e permitem que você trabalhe com fluxos de eventos assíncronos usando loops simples que parecem síncronos.

Ironicamente, embora o JavaScript forneça esses recursos poderosos para trabalhar com código assíncrono, não há recursos da linguagem principal que sejam assíncronos. Para demonstrar Promises, async, await e for/await, portanto, primeiro faremos um desvio para o JavaScript do lado do cliente e do lado do servidor para explicar alguns dos recursos assíncronos dos navegadores da web e do Node. (Você pode aprender mais sobre JavaScript do lado do cliente e do lado do servidor nos Capítulos 15 e 16.)

13.1 Programação assíncrona com retornos de chamada

Em seu nível mais fundamental, a programação assíncrona em JavaScript é feita com retornos de chamada. Um retorno de chamada é uma função que você escreve e depois passa para alguma outra função. Essa outra função invoca ("chama de volta") sua função quando alguma condição é atendida ou algum evento (assíncrono) ocorre. A invocação da função de retorno de chamada que você fornece notifica você sobre a condição ou o evento e, às vezes, a invocação incluirá argumentos de função que fornecem detalhes adicionais. Isso é mais fácil de entender com alguns exemplos concretos, e as subseções a seguir demonstram várias formas de programação assíncrona baseada em retorno de chamada usando JavaScript e Node do lado do cliente.

13.1.1 Temporizadores

Um dos tipos mais simples de assincronia é quando você deseja executar algum código após um determinado período de tempo. Como vimos em §11.10, você pode fazer isso com a função setTimeout():

```
setTimeout(checkForUpdates, 60000);
```

O primeiro argumento para setTimeout() é uma função e o segundo é um intervalo de tempo medido em milissegundos. No código anterior, uma função checkForUpdates() hipotética será chamada 60.000 milissegundos (1 minuto) após a chamada

setTimeout().checkForUpdates() é uma função de retorno de chamada que seu programa pode definir, e setTimeout() é a função que você invoca para registrar sua função de retorno de chamada e especificar em quais condições assíncronas ela deve ser invocada.

setTimeout() chama a função de retorno de chamada especificada uma vez, não passando nenhum argumento e depois a esquece. Se você está escrevendo uma função que realmente verifica se há atualizações, provavelmente deseja que ela seja executada repetidamente. Você pode fazer isso usando setInterval() em vez de setTimeout():

*Chame checkForUpdates em um minuto e novamente a cada minuto depois dissodeixe updateIntervalId =
setInterval(checkForUpdates, 60000);*

setInterval() retorna um valor que podemos usar para interromper as invocações repetidas// chamando clearInterval(). (Da mesma forma, setTimeout()// retorna um valor que você pode passar para clearTimeout() function stopCheckingForUpdates() {

clearInterval(updateIntervalId);}

13.1.2 Eventos

Os programas JavaScript do lado do cliente são quase universalmente orientados a eventos:

Em vez de executar algum tipo de computação predeterminada, eles normalmente esperam que o usuário faça algo e, em seguida, respondem às ações do usuário. O navegador da Web gera um evento quando o usuário pressiona uma tecla no teclado, move o mouse, clica em um botão do mouse ou toca em um dispositivo com tela sensível ao toque. Os programas JavaScript orientados a eventos registram funções de retorno de chamada para tipos especificados de eventos em contextos especificados, e o navegador da Web invoca essas funções sempre que os eventos especificados ocorrem. Essas funções de retorno de chamada são chamadas de manipuladores de eventos ou ouvintes de eventos e são registradas com `addEventListener()`:

*Peça ao navegador da web para retornar um objeto que represente oElemento HTML// <button> que corresponda a este seletor CSSlet okay =
document.querySelector('#confirmUpdateDialogbutton.okay');*

Agora registre uma função de retorno de chamada a ser invocada quando ousuário// clicar nesse botão.okay.addEventListener('click', applyUpdate);

Neste exemplo, `applyUpdate()` é uma função de retorno de chamada hipotética que assumimos que está implementada em outro lugar. A chamada `document.querySelector()` retorna um objeto que representa um único elemento especificado na página da Web. Chamamos `addEventListener()` nesse elemento para registrar nosso retorno de chamada. Em seguida, o primeiro argumento para `addEventListener()` é uma cadeia de caracteres que especifica o tipo de evento em que estamos interessados — um clique do mouse ou um toque na tela sensível ao toque, neste caso. Se o usuário clicar ou tocar nesse elemento específico da página da web, o navegador invocará a função de retorno de chamada `ourapplyUpdate()`, passando um objeto que inclui

detalhes (como a hora e as coordenadas do ponteiro do mouse) sobre o evento.

13.1.3 Eventos de rede

Outra fonte comum de assincronia na programação JavaScript é solicitações de rede. O JavaScript em execução no navegador pode buscar dados de um servidor web com um código como este:

```
function getCurrentVersionNumber(versionCallback) { //  
    Argumento Notecallback  
Faça uma solicitação HTTP com script para uma versão de  
    back-end APIlet request = new XMLHttpRequest();  
    request.open("GET", "http://www.example.com/api/version");  
  
    request.send();  
  
Registre um retorno de chamada que será invocado quando  
    a resposta chegar  
    request.onload = function() {if  
        (request.status === 200) {  
            Se o status HTTP for bom, obtenha o número da versão e  
            retorno de chamada de chamada.  
            let atualVersion =  
            parseFloat(request.responseText);  
            versionCallback(null, currentVersion);}  
        else {  
            Caso contrário, relate um erro para o  
            callbackversionCallback(response.statusText, null);}};  
Registre outro retorno de chamada que será invocado para  
    erros de rede  
  
    request.onerror = request.ontimeout = function(e) {  
        versionCallback(e.type, null);};}
```

O código JavaScript do lado do cliente pode usar a classe XMLHttpRequest mais funções de retorno de chamada para fazer solicitações HTTP e lidar de forma assíncrona com a resposta do servidor quando ela chega. A função getCurrentVersionNumber() definida aqui (podemos imaginar que ela é usada pela função checkForUpdates() hipotética que discutimos em §13.1.1) faz uma solicitação HTTP e define manipuladores de eventos que serão invocados quando a resposta do servidor for recebida ou quando um tempo limite ou outro erro fizer com que a solicitação falhe.

Observe que o exemplo de código acima não chama addEventListener() como nosso exemplo anterior. Para a maioria das webAPIs (incluindo esta), os manipuladores de eventos podem ser definidos invocando addEventListener() no objeto que gera o evento e passando o nome do evento de interesse junto com a função de retorno de chamada.

Normalmente, porém, você também pode registrar um único ouvinte de eventos atribuindo-o diretamente a uma propriedade do objeto. Isso é o que fazemos neste código de exemplo, atribuindo funções às propriedades onload, onerror e ontimeout. Por convenção, propriedades de ouvinte de eventos como essas sempre têm nomes que começam com on.addEventListener() é a técnica mais flexível porque permite vários manipuladores de eventos. Mas nos casos em que você tem certeza de que nenhum outro código precisará registrar um ouvinte para o mesmo objeto e tipo de evento, pode ser mais simples simplesmente definir a propriedade apropriada para seu retorno de chamada.

Outra coisa a ser observada sobre a função getCurrentVersionNumber() neste código de exemplo é que, como ela faz uma solicitação assíncrona, ela não pode retornar o valor de forma síncrona (o

número da versão atual) em que o chamador está interessado. Em vez disso, o chamador passa uma função de retorno de chamada, que é invocada quando o resultado está pronto ou quando ocorre um erro. Nesse caso, o chamador fornece uma função de retorno de chamada que espera dois argumentos. Se o XMLHttpRequest funcionar corretamente, getCurrentVersionNumber() invocará o retorno de chamada com um primeiro argumento nulo e o número da versão como o segundo argumento. Ou, se ocorrer um erro, então getCurrentVersionNumber() invoca o retorno de chamada com errorDetails no primeiro argumento e null como o segundo argumento.

13.1.4 Retornos de chamada e eventos no nó

O ambiente JavaScript do lado do servidor Node.js é profundamente assíncrono e define muitas APIs que usam retornos de chamada e eventos. A API padrão para ler o conteúdo de um arquivo, por exemplo, é assíncrona e invoca uma função de retorno de chamada quando o conteúdo do arquivo é lido:

```
const fs = require("fs"); O módulo "fs" tem APIs
relacionadas ao sistema de arquivoslet options = {// Um
objeto para conter opções para nosso programa
as opções padrão iriam aqui};
```

Leia um arquivo de configuração e chame o retorno de chamada

```
functionfs.readFile("config.json", "utf-8", (err, text) => {
    se (err) {
        Se houver um erro, exiba um aviso, mas
        continuar
        console.warn("Não foi possível ler o arquivo de
        configuração:", err);} else {
            Caso contrário, analise o conteúdo do arquivo e atribua a
            o objeto Opções
            Object.assign(opções, JSON.parse(texto));
```

```
}
```

Em ambos os casos, agora podemos começar a executar o programstartProgram(options);});

A função fs.readFile() do Node recebe um retorno de chamada de dois parâmetros como seu último argumento. Ele lê o arquivo especificado de forma assíncrona e, em seguida, invoca o retorno de chamada. Se o arquivo foi lido com êxito, ele passa o conteúdo do arquivo como o segundo argumento de retorno de chamada. Se houver um erro, ele passará o erro como o primeiro argumento de retorno de chamada. Neste exemplo, expressamos o retorno de chamada como uma função de seta, que é uma sintaxe sucinta e natural para esse tipo de operação simples.

O Node também define várias APIs baseadas em eventos. A função a seguir mostra como fazer uma solicitação HTTP para o conteúdo de uma URL no Node. Ele tem duas camadas de código assíncrono tratadas com ouvintes de eventos. Observe que o Node usa um método on() para registrar ouvintes de eventos em vez de addEventListener():

```
const https = require("https");
```

Leia o conteúdo de texto da URL e passe-o de forma assíncrona para o callback.function getText(url, callback){

Inicie uma solicitação HTTP GET para
URLrequest = https.get(url);

Registre uma função para lidar com a "resposta"
event.request.on("response", response => {
O evento de resposta significa que os cabeçalhos de resposta
foram recebidos
let httpStatus = response.statusCode;

O corpo da resposta HTTP não foi
recebido ainda.

Portanto, registramos mais manipuladores de eventos para serem chamados quando chegar.

resposta.setEncoding("utf-8"); Estamos esperando Texto Unicode que vamos ver a um momento aqui;

Esse manipulador de eventos é chamado quando uma parte do o corpo está pronto

```
response.on("dados", chunk => { body += chunk; });
```

Esse manipulador de eventos é chamado quando a resposta é completar

```
response.on("end", () => {
  if (httpStatus === 200) {    Se o HTTP
A resposta foi boa          callback(null, body); // Passar o corpo da resposta
para o retorno de chamada
  } else {                  Caso contrário, passe um
erro
  callback(httpStatus, null);}});});
```

Também registramos um manipulador de eventos para erros de rede de nível inferior

```
request.on("erro", (err) => {
callback(err, null);});}
```

13.2 Promessas

Agora que vimos exemplos de retorno de chamada e programação assíncrona baseada em eventos em ambientes JavaScript do lado do cliente e do lado do servidor, podemos apresentar o Promises, um recurso de linguagem principal projetado para simplificar a programação assíncrona.

Uma promessa é um objeto que representa o resultado de uma computação assíncrona. Esse resultado pode ou não estar pronto ainda, e a PromiseAPI é intencionalmente vaga sobre isso: não há como obter de forma síncrona o valor de uma Promise; você só pode pedir ao Promise para chamar uma função de retorno de chamada quando o valor estiver pronto. Se você estiver definindo uma API anassíncrona como a função `getText()` na seção anterior, mas quiser torná-la baseada em promessas, omita o argumento de retorno de chamada e, em vez disso, retorne um objeto Promise. O chamador pode então registrar um ou mais retornos de chamada nesse objeto Promise e eles serão invocados quando a computação assíncrona for concluída.

Então, no nível mais simples, as promessas são apenas uma maneira diferente de trabalhar com retornos de chamada. No entanto, há benefícios práticos em usá-los. Um problema real com a programação assíncrona baseada em retorno de chamada é que é comum acabar com retornos de chamada dentro de retornos de chamada dentroretornos de chamada, com linhas de código tão recuadas que é difícil de ler. As promessas permitem que esse tipo de retorno de chamada aninhado seja reexpresso como uma cadeia de promessas mais linear que tende a ser mais fácil de ler e mais fácil de raciocinar.

Outro problema com os retornos de chamada é que eles podem dificultar o tratamento de erros. Se uma função assíncrona (ou um callback invocado de forma assíncrona) lançar uma exceção, não há como essa exceção se propagar de volta para o iniciador da operação assíncrona. Este é um fato fundamental sobre a programação assíncrona: ela interrompe o tratamento de exceções. A alternativa é rastrear e propagarmeticulosamente os erros com argumentos de retorno de chamada e valores de retorno, mas isso é difícil e difícil de acertar. As promessas ajudam aqui, padronizando para lidar com erros e fornecendo uma maneira de os erros se propagarem

corretamente através de uma cadeia de promessas.

Observe que as promessas representam os resultados futuros de cálculos assíncronos únicos. No entanto, eles não podem ser usados para representar cálculos assíncronos repetidos. Mais adiante neste capítulo, escreveremos uma alternativa baseada em Promise para a função setTimeout(), por exemplo. Mas não podemos usar Promises para substituir setInterval() porque essa função invoca uma função de retorno de chamada repetidamente, o que é algo que as promessas simplesmente não foram projetadas para fazer. Da mesma forma, poderíamos usar aPromise em vez do manipulador de eventos "load" de um objeto XMLHttpRequest, já que esse retorno de chamada é chamado apenas uma vez. Mas normalmente não usaríamos um Promise no lugar de um manipulador de eventos de "clique" de um objeto de botão HTML, já que normalmente queremos permitir que o usuário clique em um botão várias vezes.

As subseções a seguir irão:

- Explicar a terminologia do Promise e mostrar o uso básico do
- PromiseMostrar como as promessas podem ser encadeadasDemonstrar
- como criar suas próprias APIs baseadas em Promise

IMPORTANTE

As promessas parecem simples no início, e o caso de uso básico das promessas é, na verdade, direto e simples. Mas eles podem se tornar surpreendentemente confusos para qualquer coisa além dos casos de uso mais simples. As promessas são um poderoso idioma para programação assíncrona, mas você precisa entendê-las profundamente para usá-las corretamente e com confiança. Vale a pena dedicar um tempo para desenvolver esse entendimento profundo, no entanto, e peço que você estude este longo capítulo com cuidado.

13.2.1 Usando promessas

Com o advento do Promises na linguagem JavaScript principal, os navegadores da web começaram a implementar APIs baseadas em Promise. Na seção anterior, implementamos uma função `getText()` que fez uma solicitação HTTP assíncrona e passou o corpo da resposta HTTP para uma função de retorno de chamada especificada como uma string. Imagine uma variante dessa função, `getJSON()`, que analisa o corpo da resposta HTTP como JSON e retorna uma promessa em vez de aceitar um argumento de retorno de chamada. Implementaremos uma função `getJSON()` mais adiante neste capítulo, mas, por enquanto, vamos ver como usariamos essa função utilitária de retorno de promessa:

```
getJSON(url).then(jsonData => {  
  Esta é uma função de retorno de chamada que  
  será assíncrona  
  invocado com o valor JSON analisado quando ele se torna  
  disponível.});
```

`getJSON()` inicia uma solicitação HTTP assíncrona para a URL especificada e, em seguida, enquanto essa solicitação está pendente, ela retorna um objeto Promise. O objeto Promise define um método de instância `then()`. Em vez de passar nossa função de retorno de chamada diretamente para `getJSON()`, nós a passamos para o método `then()`. Quando a resposta HTTP chega, o corpo dessa resposta é analisado como JSON e o valor analisado resultante é passado para a função que passamos para `then()`.

Você pode pensar no método `then()` como um método de registro de retorno de chamada, como o método `addEventListener()` usado para registrar manipuladores de eventos em JavaScript do lado do cliente. Se você chamar o comando `then()`

de um objeto Promise várias vezes, cada uma das funções especificadas será chamada quando a computação prometida for concluída.

Ao contrário de muitos ouvintes de eventos, no entanto, uma Promise representa uma única computação, e cada função registrada com `then()` será invocada apenas uma vez. Vale a pena notar que a função que você passa a `then()` é invocada de forma assíncrona, mesmo que a computação assíncrona já esteja concluída quando você chama `then()`.

Em um nível sintático simples, o método `then()` é a característica distintiva de Promises, e é idiomático acrescentar `.then()` diretamente à invocação da função que retorna a Promise, sem a etapa intermediária de atribuir o objeto Promise a uma variável.

Também é idiomático nomear funções que retornam Promises e funções que usam os resultados de Promises com verbos, e essas expressões idiomáticas levam a um código que é particularmente fácil de ler:

Suponha que você tenha uma função como esta para exibir uma função `userprofile`

```
displayUserProfile(profile) { /* implementation omitted */ }
```

Veja como você pode usar essa função com uma promessa.// Observe como essa linha de código é lida quase como uma frase em inglês:

```
getJSON("/api/user/profile").then(displayUserProfile);
```

LIDANDO COM ERROS COM PROMISES As operações assíncronas, particularmente aquelas que envolvem rede, normalmente podem falhar de várias maneiras, e um código robusto precisa ser escrito para lidar com os erros que inevitavelmente ocorrerão.

Para Promises, podemos fazer isso passando uma segunda função para o método then():

```
getJSON("/api/user/profile").then(displayUserProfile  
, handleProfileError);
```

Uma Promise representa o resultado futuro de uma computação assíncrona que ocorre após a criação do objeto Promise. Como a computação é executada depois que o objeto Promise é retornado para nós, não há como a computação tradicionalmente retornar um valor ou lançar uma exceção que possamos capturar. As funções que passamos tothen() fornecem alternativas. Quando uma computação síncrona é concluída normalmente, ela simplesmente retorna seu resultado para o chamador. Quando a computação assíncrona baseada em aPromise é concluída normalmente, ela passa seu resultado para a função que é o primeiro argumento para then().

Quando algo dá errado em uma computação síncrona, ele lança uma exceção que propaga a pilha de chamadas até que haja uma cláusula catch para lidar com isso. Quando uma computação assíncrona é executada, seu chamador não está mais na pilha, portanto, se algo der errado, simplesmente não é possível lançar uma exceção de volta para o chamador.

Em vez disso, os cálculos assíncronos baseados em promessas passam a exceção (normalmente como um objeto Error de algum tipo, embora isso não seja necessário) para a segunda função passada para then(). Então, no código acima, ifgetJSON() é executado normalmente, ele passa seu resultado para displayUserProfile(). Se houver um erro (o usuário não está conectado, o servidor está inativo, a conexão com a Internet do usuário caiu, a solicitação expirou etc.), getJSON() passa um objeto Error para

handleProfileError().

Na prática, é raro ver duas funções passadas para then(). Existe uma maneira melhor e mais idiomática de lidar com erros ao trabalhar com Promises. Para entendê-lo, primeiro considere o que acontece se getJSON() for concluído normalmente, mas ocorrer um erro em displayUserProfile(). Essa função de retorno de chamada é invocada de forma assíncrona quando getJSON() retorna, portanto, também é assíncrona e não pode lançar uma exceção de forma significativa (porque não há código na pilha de chamadas para lidar com isso).

A maneira mais idiomática de lidar com erros neste código é assim:

```
getJSON("/api/usuário/perfil").then(displayUserProfile).catch(handleProfileError);
```

Com esse código, um resultado normal de getJSON() ainda é passado para displayUserProfile(), mas qualquer erro em getJSON() ou displayUserProfile() (incluindo quaisquer exceções lançadas por displayUserProfile) é passado para handleProfileError(). O método catch() é apenas uma abreviação para chamar then() com um primeiro argumento nulo e a função de manipulador de erro especificada como o segundo argumento.

Teremos mais a dizer sobre catch() e esse idioma de tratamento de erros quando discutirmos as cadeias de promessas na próxima seção.

TERMINOLOGIA DE PROMESSA

Antes de discutirmos mais as promessas, vale a pena fazer uma pausa para definir alguns termos. Quando não estamos

programação e falamos sobre promessas humanas, dizemos que uma promessa é "mantida" ou "quebrada". Ao discutir promessas de JavaScript, os termos equivalentes são "cumpridos" e "rejeitados". Imagine que você chamou o método `then()` de uma promessa e passou duas funções de retorno de chamada para ele. Dizemos que a promessa foi cumprida se e quando o primeiro retorno de chamada for chamado. E dizemos que a Promise foi rejeitada se e quando o segundo retorno de chamada for chamado. Se uma promessa não é cumprida nem rejeitada, então ela está pendente. E uma vez que uma promessa é cumprida ou rejeitada, dizemos que ela está resolvida. Observe que uma promessa nunca pode ser cumprida e rejeitada. Uma vez que uma promessa é estabelecida, ela nunca mudará de cumprida para rejeitada ou vice-versa.

Lembre-se de como definimos Promises no início desta seção: "uma Promise é um objeto que representa o resultado de uma operação assíncrona". É importante lembrar que as promessas não são apenas maneiras abstratas de registrar retornos de chamada para serem executados quando algum código assíncrono é concluído - elas representam os resultados desse código assíncrono. Se o código assíncrono for executado normalmente (e a promessa for cumprida), esse resultado será essencialmente o valor retornado do código. E se o código assíncrono não for concluído normalmente (e a promessa for rejeitada), o resultado será um objeto `Error` ou algum outro valor que o código poderia ter gerado se não fosse assíncrono. Qualquer Promise que tenha sido liquidada tem um valor associado a ela, e esse valor não será alterado. Se a promessa for cumprida, o valor será um valor de retorno que será passado para qualquer função de retorno de chamada registrada como o primeiro argumento de `then()`. Se a promessa for rejeitada, o valor será um erro de algum tipo que será passado para qualquer função de retorno de chamada registrada com `catch()` ou como o segundo argumento de `then()`.

A razão pela qual quero ser preciso sobre a terminologia da promessa é que as promessas também podem ser resolvidas. É fácil confundir esse estado resolvido com o estado cumprido ou com o estado estabelecido, mas não é exatamente o mesmo que nenhum dos dois. Compreender o estado resolvido é uma das chaves para uma compreensão profunda das promessas, e voltarei a isso depois de discutirmos as cadeias de promessas abaixo.

13.2.2 Encadeamento de promessas

Um dos benefícios mais importantes das Promises é que elas fornecem uma maneira natural de expressar uma sequência de operações assíncronas como uma cadeia linear de invocações do método `then()`, sem ter que aninhar cada operação dentro do retorno de chamada da anterior. Aqui, por exemplo, está uma cadeia de promessas hipotética:

```
fetch(URL do documento)          Fazer um HTTP  
pedir  
    .then(resposta => resposta.json())  Solicite o JSON  
corpo da resposta  
    .then(documento => {                Quando obtivermos o  
JSON analisado  
        retornar render(documento);      exiba o  
documento para o usuário
```

```
}).then(renderizado
  o => {
    Quando obtivermos o
    documento renderizado
      cacheInDatabase(renderizado);
    banco de dados local.
  }).catch(erro =>
  handle(erro));
  Quando armazene-o em cache no arquivo
  que ocorrem
  Lidar com quaisquer erros
```

Esse código ilustra como uma cadeia de promessas pode facilitar a expressão de uma sequência de operações assíncronas. No entanto, não vamos discutir essa cadeia de promessas em particular. No entanto, continuaremos a explorar a ideia de usar cadeias de promessas para fazer solicitações HTTP.

Anteriormente neste capítulo, vimos o objeto XMLHttpRequest usado para fazer uma solicitação HTTP em JavaScript. Esse objeto de nome estranho tem uma API antiga e estranha e foi amplamente substituída pela API Fetch mais recente baseada em promessas (§15.11.1). Em sua forma mais simples, esta nova API HTTP é apenas a função `fetch()`. Você passa uma URL e ela retorna uma promessa. Essa promessa é cumprida quando a resposta HTTP começa a chegar e o status HTTP e os cabeçalhos estão disponíveis:

```
fetch("/api/usuário/perfil").then(resposta => {
  Quando a promessa é resolvida, temos status e headers if
  (response.ok &&
    response.headers.get("Tipo de conteúdo") ===
    "aplicativo/json") {
    O que podemos fazer aqui? Na verdade, não temos o
    corpo de resposta ainda.
  });
  
```

Quando a promessa retornada por `fetch()` é cumprida, ela passa um

Responde à função que você passou para seu método `then()`. Esse objeto de resposta fornece acesso ao status e aos cabeçalhos da solicitação e também define métodos como `text()` e `json()`, que fornecem acesso ao corpo da resposta em formulários de texto e analisados por JSON, respectivamente. Mas, embora a promessa inicial seja cumprida, o corpo da resposta pode ainda não ter chegado. Portanto, esses métodos `text()` e `json()` para acessar o corpo da resposta retornam Promises. Aqui está uma maneira ingênua de usar os métodos `fetch()` e `theresponse.json()` para obter o corpo de uma resposta HTTP:

```
fetch("/api/usuário/perfil").then(resposta => {
    response.json().then(perfil => { Peça o JSON-
    corpo analisado
        Quando o corpo da resposta chegar, será
        automaticamente
        analisado como JSON e passado para esta
        função.displayUserProfile(profile);});});
```

Essa é uma maneira ingênua de usar Promises porque as aninhamos, como callbacks, o que anula o propósito. O idioma preferido é usar Promises em uma cadeia sequencial com código como este:

```
fetch("/api/usuário/perfil")
    .then(resposta => {
        return response.json();}).then(perfil => {
            displayUserProfile(perfil);});
```

Vejamos as invocações de método neste código, ignorando o

argumentos que são passados para os métodos:

```
fetch().then().then()
```

Quando mais de um método é invocado em uma única expressão como esta, chamamos isso de cadeia de métodos. Sabemos que a função `fetch()` retorna um objeto `Promise` e podemos ver que o primeiro `.then()` nesta cadeia invoca um método nesse objeto `Promise` retornado. Mas há um `second.then()` na cadeia, o que significa que a primeira invocação do método `then()` deve retornar uma `Promise`.

Às vezes, quando uma API é projetada para usar esse tipo de encadeamento de métodos, há apenas um único objeto, e cada método desse objeto retorna o próprio objeto para facilitar o encadeamento. Não é assim que as promessas funcionam, no entanto. Quando escrevemos uma cadeia de invocações `.then()`, não estamos registrando vários retornos de chamada em um objeto `singlePromise`. Em vez disso, cada invocação do método `then()` retorna um novo objeto `Promise`. Esse novo objeto `Promise` não é cumprido até que a função passada para `then()` seja concluída.

Vamos retornar a uma forma simplificada da cadeia `fetch()` original acima. Se definirmos as funções passadas para as invocações `then()` em outro lugar, podemos refatorar o código para ficar assim:

```
fetch(aURL)           tarefa 1; retorna promessa 1
    .then(retorno de chamada1) tarefa 2; Retorna a promessa 2
    .then(retorno de      tarefa 3; Retorna Promessa 3
          chamada2);
```

Vamos examinar este código em detalhes:

1. Na primeira linha, `fetch()` é invocado com um URL. Ele inicia uma solicitação HTTP GET para essa URL e retorna uma `Promise`. Chamaremos essa solicitação HTTP de "tarefa 1" e chamaremos a `Promise` de "promessa 1".
2. Na segunda linha, invocamos o método `then()` da promessa 1, passando a função `callback1` que queremos que seja invocada quando a promessa 1 for cumprida. O método `then()` armazena nosso retorno de chamada em algum lugar e, em seguida, retorna uma nova `Promise`. Chamaremos a nova `Promise` retornada nesta etapa de "promise 2" e diremos que a "tarefa 2" começa quando `callback1` é invocado.
3. Na terceira linha, invocamos o método `then()` de `promise2`, passando a função `callback2` que queremos invocar quando a promessa 2 for cumprida. Este método `then()` lembra nosso retorno de chamada e retorna mais uma promessa. Diremos que "task3" começa quando `callback2` é invocado. Podemos chamar esta última promessa de "promessa 3", mas realmente não precisamos de um nome para ela porque não a usaremos.
4. As três etapas anteriores acontecem de forma síncrona quando a expressão é executada pela primeira vez. Agora temos uma pausa assíncrona enquanto a solicitação HTTP iniciada na etapa 1 é enviada pela Internet.
5. Eventualmente, a resposta HTTP começa a chegar. A parte assíncrona da chamada `fetch()` envolve o status HTTP e os cabeçalhos em um objeto `Response` e cumpre a promessa 1 com esse objeto `Response` como o valor.
6. Quando a promessa 1 é cumprida, seu valor (o objeto `Response`) é passado para nossa função `callback1()` e a tarefa 2 começa. O trabalho desta tarefa, dado um objeto `Response` como entrada, é obter o corpo da resposta como um objeto JSON.
7. Vamos supor que a tarefa 2 seja concluída normalmente e seja capaz de

analise o corpo da resposta HTTP para produzir um objeto JSON. Esse objeto JSON é usado para cumprir a promessa 2.

8. O valor que cumpre a promessa 2 torna-se a entrada para a tarefa 3 quando é passado para a função callback2(). Esta terceira tarefa agora exibe os dados para o usuário de alguma forma não especificada. Quando a tarefa 3 estiver concluída (supondo que ela seja concluída normalmente), a promessa 3 será cumprida. Mas como nunca fizemos nada com a promessa 3, nada acontece quando essa promessa se estabelece, e a cadeia de computação assíncrona termina neste ponto.

13.2.3 Resolvendo promessas

Ao explicar a cadeia de promessas de busca de URL com a lista na última seção, falamos sobre as promessas 1, 2 e 3. Mas, na verdade, há um quarto objeto de promessa envolvido também, e isso nos leva à nossa importante discussão sobre o que significa uma promessa ser "resolvida".

Lembre-se de que fetch() retorna um objeto Promise que, quando cumprido, passa um objeto Response para a função de retorno de chamada que registramos. Este objeto Response tem .text(), .json() e outros métodos para solicitar o corpo da resposta HTTP de várias formas. Mas como o corpo pode ainda não ter chegado, esses métodos devem retornar Promiseobjects. No exemplo que estamos estudando, a "tarefa 2" chama o método the.json() e retorna seu valor. Este é o quarto objeto Promise, e é o valor de retorno da função callback1().

Vamos reescrever o código de busca de URL mais uma vez de uma forma detalhada e não idiomática que torna os retornos de chamada e promessas explícitos:

```
função c1(resposta) {
```

retorno de chamada 1

```

        seja p4 = response.json();
        retorno p4;                                Retorna a promessa 4
    }

    função c2(perfil) {                         retorno de chamada 2
        displayUserProfile(perfil);}

let p1 = fetch("/api/usuário/perfil");      Promessa 1, Tarefa 1
seja p2 = p1.then(c1);                      Promessa 2, Tarefa 2
seja p3 = p2.then(c2);                      Promessa 3, Tarefa 3

```

Para que as cadeias de promessas funcionem de forma útil, a saída da tarefa 2 deve se tornar a entrada para a tarefa 3. E no exemplo que estamos considerando aqui, a entrada para a tarefa 3 é o corpo da URL que foi buscada, analisada como um objeto JSON. Mas, como acabamos de discutir, o valor de retorno de callbackc1 não é um objeto JSON, mas Promise p4 para esse objeto JSON. Isso parece uma contradição, mas não é: quando p1 é cumprido, c1 é invocado e a tarefa 2 começa. E quando p2 é cumprido, c2 é invocado e a tarefa 3 começa. Mas só porque a tarefa 2 começa quando c1 é invocado, isso não significa que a tarefa 2 deva terminar quando c1 retornar. Afinal, as promessas são sobre o gerenciamento de tarefas assíncronas e, se a tarefa 2 for assíncrona (o que é, neste caso), essa tarefa não estará concluída quando o retorno de chamada retornar.

Agora estamos prontos para discutir o detalhe final que você precisa entender para realmente dominar as promessas. Quando você passa um retorno de chamada c para o método then(), then() retorna uma promessa p e organiza a invocação assíncrona de c em algum momento posterior. O retorno de chamada executa alguma computação e retorna um valor v. Quando o retorno de chamada retorna, pis resolvido com o valor v. Quando uma promessa é resolvida com um valor

isso não é em si uma promessa, é imediatamente cumprido com esse valor. Portanto, se c retorna uma não-promessa, esse valor de retorno se torna o valor de p, p é cumprido e pronto. Mas se o valor de retorno v for ele mesmo aPromise, então p é resolvido, mas ainda não cumprido. Neste estágio, p não pode ser liquidado até que a Promessa v se estabeleça. Se v for cumprido, então p será cumprido com o mesmo valor. Se v for rejeitado, p será rejeitado pelo mesmo motivo. Isso é o que o estado "resolvido" de uma Promessa significa: a Promessa tornou-se associada ou "bloqueada" em outra Promessa. Ainda não sabemos se p será cumprido ou rejeitado, mas nosso retorno de chamada c não tem mais controle sobre isso. p é "resolvido" no sentido de que seu destino agora depende inteiramente do que acontece com Promisev.

Vamos trazer isso de volta ao nosso exemplo de busca de URL. Quando c1 retorna p4, p2 é resolvido. Mas ser resolvido não é o mesmo que ser cumprido, então a tarefa 3 ainda não começa. Quando o corpo completo da resposta HTTP se torna disponível, o método .json() pode analisá-lo e usar esse valor analisado para cumprir p4. Quando p4 é preenchido, p2 também é preenchido automaticamente, com o mesmo valor JSON analisado. Nesse ponto, o objeto JSON analisado é passado para c2 e a tarefa 3 é iniciada.

Esta pode ser uma das partes mais difíceis de entender do JavaScript evocê pode precisar ler esta seção mais de uma vez. A Figura 13-1 apresenta o processo de forma visual e pode ajudar a esclarecer o para você.

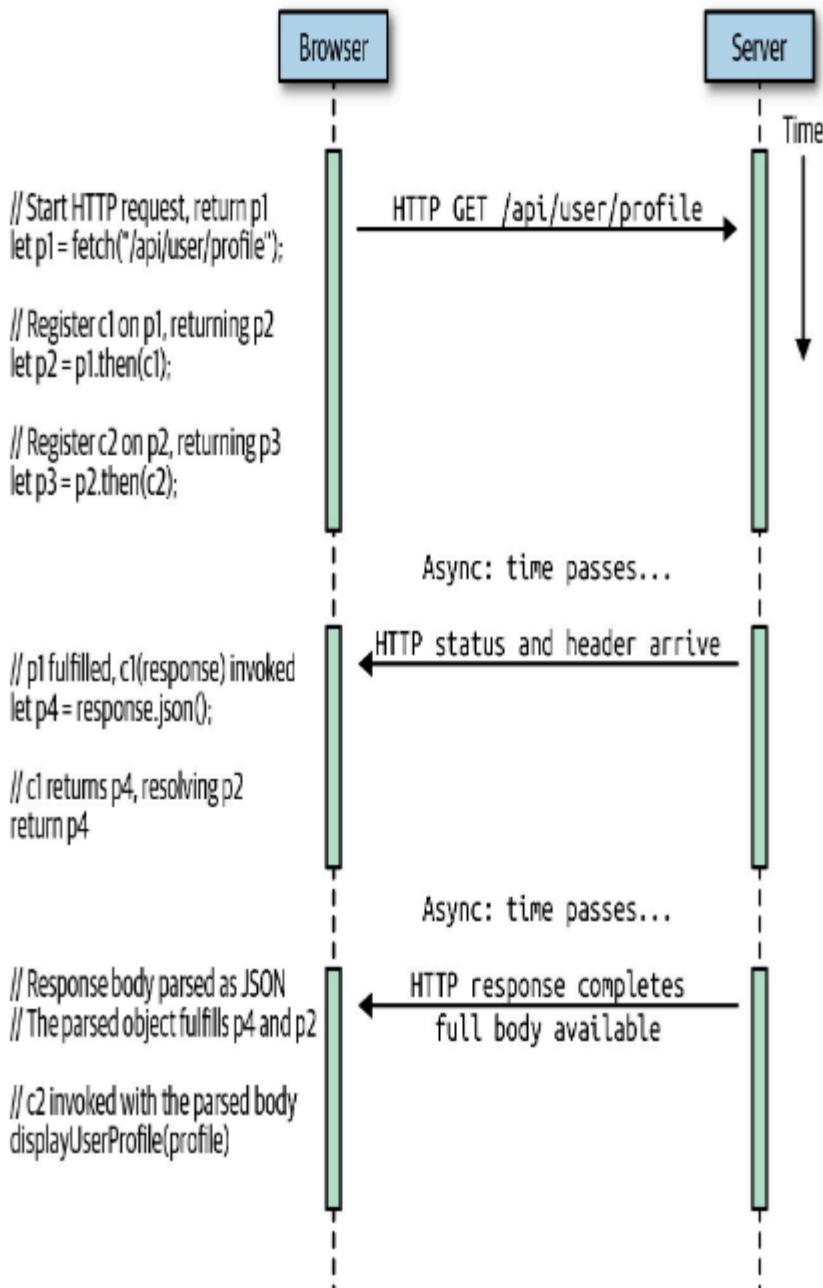


Figura 13-1. Buscando um URL com promessas

13.2.4 Mais sobre promessas e erros

No início do capítulo, vimos que você pode passar uma segunda função de retorno de chamada para o método `.then()` e que essa segunda função será invocada se a Promise for rejeitada. Quando isso acontece, o argumento para essa segunda função de retorno de chamada é um valor — normalmente um objeto `Error` — que representa o motivo da rejeição. Também aprendemos que é incomum (e até não idiomático) passar dois retornos de chamada para o método `a.then()`. Em vez disso, os erros relacionados ao Promise são normalmente tratados adicionando uma invocação do método `.catch()` a um `PromiseChain`. Agora que examinamos as cadeias de promessas, podemos retornar ao tratamento de erros e discuti-lo com mais detalhes. Para prefaciar a discussão, gostaria de enfatizar que o tratamento cuidadoso de erros é realmente importante ao fazer programação assíncrona. Com o código síncrono, se você deixar de fora o código de tratamento de erros, obterá pelo menos uma exceção e um rastreamento de pilha que poderá usar para descobrir o que está errado. Com código assíncrono, exceções sem tratamento geralmente não são relatadas e erros podem ocorrer silenciosamente, tornando-as muito mais difíceis de depurar. A boa notícia é que o método `.catch()` facilita o tratamento de erros ao trabalhar com Promises.

OS MÉTODOS CATCH E FINALLY O método `.catch()` de uma Promise é simplesmente uma maneira abreviada de chamar `.then()` com null como o primeiro argumento e um retorno de chamada de tratamento de erros como o segundo argumento. Dado qualquer Promise p e um callbackc, as duas linhas a seguir são equivalentes:

```
p.then(nulo,  
c); p.catch(c);
```

A abreviação .catch() é preferida porque é mais simples e porque o nome corresponde à cláusula catch em uma instrução try/catchexception-handling. Como discutimos, exceções normais não funcionam com código assíncrono. O método .catch() ofPromises é uma alternativa que funciona para código assíncrono. Quando algo dá errado no código síncrono, podemos falar de uma exceção "borbulhando a pilha de chamadas" até encontrar um bloco catch. Com uma cadeia assíncrona de promessas, a metáfora comparável pode ser de um erro "escorrendo pela cadeia" até encontrar uma invocação .catch().

No ES2018, os objetos Promise também definem um método .finally() cuja finalidade é semelhante à cláusula finally na instrução try/catch/finally. Se você adicionar uma invocação .finally() à sua cadeia de promessas, o retorno de chamada que você passar para .finally() será invocado quando a promessa que você chamou for resolvida. Seu retorno de chamada será invocado se a promessa cumprir ou rejeitar, e não será passado nenhum argumento, então você não pode descobrir se ele foi cumprido ou rejeitado. Mas se você precisar executar algum tipo de código de limpeza (como fechar arquivos abertos ou conexões de rede) em ambos os casos, um retorno de chamada .finally() é a maneira ideal de fazer isso. Como .then() e .catch(), .finally() retorna um novo objeto Promise. O valor de retorno de um retorno de chamada .finally() é geralmente ignorado, e thePromise retornado por .finally() normalmente resolverá ou rejeitará com o mesmo valor que a Promise em que .finally() foi invocada

resolve ou rejeita com. No entanto, se um retorno de chamada .finally() lançar uma exceção, a Promise retornada por .finally() será rejeitada com esse valor.

O código de busca de URL que estudamos nas seções anteriores não fez nenhum tratamento de erros. Vamos corrigir isso agora com uma versão mais realista do código:

```
fetch("/api/usuário/perfil")  Iniciar a solicitação HTTP
    .then(resposta => {          Chame isso quando status e
        Os cabeçalhos estão prontos
        if (!resposta.ok) {       Se tivermos um 404 Not Found ou
            erro semelhante      talvez o usuário esteja desconectado;
            retornar nulo;        Retornar perfil nulo
        }
    }

    Agora verifique os cabeçalhos para garantir que o servidor
    nos enviou JSON.
    Caso contrário, nosso servidor está quebrado e este é um
    erro grave!
    let tipo = resposta.cabeçalhos.get("tipo de
        conteúdo"); if (tipo !== "application/json") {
        throw new TypeError('JSON esperado, tenho
        ${tipo}');
    }

    Se chegarmos aqui, teremos um status 2xx e um
    Tipo de conteúdo JSON
        para que possamos devolver com confiança uma Promessa para o
        resposta
        body como um objeto JSON. return response.json();}.
    then(profile => {// Chamado com o corpo da resposta
        analisada ou null

        if (perfil) {
            displayUserProfile(perfil);
        }
    })
}
```

```
        else { // Se obtivermos um erro 404 acima e retornarmos
    null acabamos aqui
        displayLoggedOutProfilePage();}}).
    catch (e => {
        if (e instânciade NetworkError) {
            fetch() pode falhar dessa maneira se a internet
            a conexão está inativa
            displayErrorMessage("Verifique sua internet
            conexão.");
        }else if (e instanceof TypeError)
        {
            Isso acontece se lançarmos TypeError
            abovedisplayErrorMessage("Algo está errado com nosso
            servidor!");
        }else
        {
            Deve ser algum tipo de erro
            imprevistoconsole.error(e);}});
    
```

Vamos analisar esse código observando o que acontece quando as coisas dão errado. Usaremos o esquema de nomenclatura que usamos antes: p1 é thePromise retornado pela chamada fetch(). p2 é a promessa retornada pela primeira chamada .then() e c1 é o retorno de chamada que passamos para essa chamada .then(). p3 é a promessa retornada pela segunda chamada .then() e c2 é o retorno de chamada que passamos para essa chamada. Finalmente, c3 é o retorno de chamada que passamos para a chamada .catch(). (Essa chamada retorna aPromise, mas não precisamos nos referir a ela pelo nome.)

A primeira coisa que pode falhar é a própria solicitação fetch(). Se a conexão de rede estiver inativa (ou por algum outro motivo uma solicitação HTTP não puder ser feita), a Promessa p1 será rejeitada com um

NetworkError. Não passamos uma função de retorno de chamada de tratamento de erros como o segundo argumento para a chamada .then(), então p2 também rejeita com o mesmo objeto NetworkError. (Se tivéssemos passado um errorhandler para essa primeira chamada .then(), o manipulador de erros seria invocado e, se retornasse normalmente, p2 seria resolvido e/ou preenchido com o valor de retorno desse manipulador.) Sem um manipulador, no entanto, p2 é rejeitado e, em seguida, p3 é rejeitado pelo mesmo motivo. Neste ponto, o retorno de chamada de tratamento de erros c3 é chamado e o código específico de NetworkError dentro dele é executado.

Outra maneira pela qual nosso código pode falhar é se nossa solicitação HTTP retornar um erro 404Not Found ou outro erro HTTP. Essas são respostas HTTP válidas, portanto, a chamada fetch() não as considera erros. fetch() encapsula um objeto 404 Not Found in a Response e preenche p1 com esse objeto, fazendo com que c1 seja invocado. Nosso código em c1 verifica a okpropriedade do objeto Response para detectar que ele não recebeu uma resposta HTTP normal e lida com esse caso simplesmente retornando null. Como esse valor retornado não é uma Promise, ele cumpre p2 imediatamente e c2 é invocado com esse valor. Nosso código em c2 verifica e manipula explicitamente valores falsos, exibindo um resultado diferente para o usuário. Este é um caso em que tratamos uma condição anormal como anonerror e a tratamos sem realmente usar um manipulador de erros.

Um erro mais sério ocorre em c1 se obtivermos um código de resposta HTTP normal, mas o cabeçalho Content-Type não estiver definido adequadamente. Nosso código espera uma resposta formatada em JSON, portanto, se o servidor estiver nos enviando HTML, XML ou texto simples, teremos um problema. c1 inclui código para verificar o cabeçalho Content-Type. Se o

estiver errado, ele trata isso como um problema irrecuperável e lança a TypeError. Quando um retorno de chamada passado para .then() (ou .catch()) lança um valor, a Promise que era o valor de retorno da chamada .then() é rejeitada com esse valor lançado. Nesse caso, o código em c1 que gera um TypeError faz com que p2 seja rejeitado com esse objeto TypeError. Como não especificamos um manipulador de erros para p2, p3 também será rejeitado. c2 não será chamado e o TypeError será passado para c3, que tem código para verificar e lidar explicitamente com esse tipo de erro.

Há algumas coisas que vale a pena observar sobre esse código.

Primeiro, observe que o objeto de erro lançado com uma instrução throw, regular e síncrona, acaba sendo tratado de forma assíncrona com uma invocação do método .catch() em uma cadeia Promise. Isso deve deixar claro por que esse método abreviado é preferível a passar um segundo argumento para .then() e também por que é tão idiomático encerrar cadeias de promessas com uma chamada .catch().

Antes de deixarmos o tópico de tratamento de erros, quero salientar que, embora seja idiomático terminar cada cadeia de promessas com um .catch() para limpar (ou pelo menos registrar) quaisquer erros que ocorreram na cadeia, também é perfeitamente válido usar .catch() em outro lugar em uma cadeia de promessas. Se um dos estágios em sua cadeia de promessas pode falhar com um erro, e se o erro é algum tipo de erro recuperável que não deve impedir o resto da cadeia de execução, então você pode inserir uma chamada .catch() na cadeia, resultando em um código que pode ficar assim:

```
startAsyncOperation()  
  .then(doStageTwo).catch(recover  
    FromStageTwoError)
```

```
.then(doStageThree).then(doStageFour)
    .catch(logStageThreeAndFourErrors);
```

Lembre-se de que o retorno de chamada que você passa para `.catch()` só será invocado se o retorno de chamada em um estágio anterior gerar um erro. Se o retorno de chamada retornar normalmente, o retorno de chamada `.catch()` será ignorado e o valor de retorno do retorno de chamada anterior se tornará a entrada para o próximo retorno de chamada `.then()`. Lembre-se também de que os retornos de chamada `.catch()` não são apenas para relatar erros, mas para lidar e se recuperar de erros. Depois que um erro é passado para um retorno de chamada `.catch()`, ele para de se propagar na cadeia `Promise`. Um retorno de chamada `.catch()` pode gerar um novo erro, mas se retornar normalmente, esse valor de retorno será usado para resolver e/ou cumprir a promessa associada e o erro para de se propagar.

Sejamos concretos sobre isso: no exemplo de código anterior, se `startAsyncOperation()` ou `doStageTwo()` gerar um erro, a função `recoverFromStageTwoError()` será invocada. Se `recoverFromStageTwoError()` retornar normalmente, seu valor de retorno será passado para `doStageThree()` e a operação assíncrona continuará normalmente. Por outro lado, se `recoverFromStageTwoError()` não puder se recuperar, ele próprio lançará um erro (ou lançará novamente o erro de que foi passado). Nesse caso, nem `doStageThree()` nem `doStageFour()` serão invocados, e o erro gerado por `recoverFromStageTwoError()` será passado para `logStageThreeAndFourErrors()`.

Às vezes, em ambientes de rede complexos, os erros podem ocorrer mais ou menos aleatoriamente, e pode ser apropriado lidar com esses erros simplesmente repetindo a solicitação assíncrona. Imagine que você escreveu uma operação baseada em Promise para consultar um banco de dados:

```
queryDatabase()  
  .then(displayTable).catch(di  
splayDatabaseError);
```

Agora suponha que problemas transitórios de carga de rede estejam causando falha em cerca de 1% do tempo. Uma solução simples pode ser repetir a consulta com uma chamada .catch():

```
queryDatabase()  
  .catch(e => wait(500).then(queryDatabase)) Em  
falha, aguarde e tente novamente  
  .then(displayTable).catch(di  
splayDatabaseError);
```

Se as falhas hipotéticas forem realmente aleatórias, adicionar essa linha de código deve reduzir sua taxa de erro de 1% para 0,01%.

RETORNANDO DE UM RETORNO DE CHAMADA DE PROMESSA

Vamos retornar uma última vez ao exemplo anterior de busca de URL e considerar o retorno de chamada c1 que passamos para a primeira invocação .then(). Observe que há três maneiras pelas quais c1 pode terminar. Ele pode retornar normalmente com a Promise retornada pela chamada .json(). Isso faz com que p2 seja resolvido, mas se essa promessa é cumprida ou rejeitada depende do que acontece com a promessa recém-retornada. c1 também pode retornar normalmente com o valor nulo, o que faz com que p2 seja atendido imediatamente. Finalmente, c1 pode terminar lançando um erro, o que faz com que p2 seja rejeitado. Esses são os três resultados possíveis para uma Promise, e o código em c1 demonstra como o retorno de chamada pode causar cada resultado.

Em uma cadeia de promessas, o valor retornado (ou lançado) em um estágio da cadeia torna-se a entrada para o próximo estágio da cadeia, por isso é fundamental acertar. Na prática, esquecer de retornar um valor de uma função de retorno de chamada é, na verdade, uma fonte comum de bugs relacionados ao Promise, e isso é exacerbado pela sintaxe de atalho da função de seta do JavaScript. Considere esta linha de código que vimos anteriormente:

```
.catch(e => wait(500).then(queryDatabase))
```

Lembre-se do Capítulo 8 que as funções de seta permitem muitos atalhos. Como há exatamente um argumento (o valor do erro), podemos omitir os parênteses. Como o corpo da função é uma única expressão, podemos omitir as chaves ao redor do corpo da função e o valor da expressão se torna o valor de retorno da função. Por causa desses atalhos, o código anterior está correto. Mas considere esta mudança aparentemente inócuas:

```
.catch(e => { wait(500).then(queryDatabase) })
```

Ao adicionar as chaves, não obtemos mais o retorno automático. Essa função agora retorna undefined em vez de retornar uma Promise, o que significa que o próximo estágio nessa cadeia de Promise será invocado com undefined como sua entrada, em vez do resultado da consulta repetida. É um erro sutil que pode não ser fácil de depurar.

13.2.5 Promessas em paralelo

Passamos muito tempo falando sobre cadeias de promessas para executar sequencialmente as etapas assíncronas de uma operação assíncrona maior. Às vezes, porém, queremos executar várias operações assíncronas em paralelo. A função Promise.all() pode fazer isso.

Promise.all() recebe uma matriz de objetos Promise como entrada e retorna uma Promise. A promessa retornada será rejeitada se qualquer uma das promessas de entrada for rejeitada. Caso contrário, ele será cumprido com uma matriz dos valores de cumprimento de cada uma das promessas de entrada. Então, por exemplo, se você quiser buscar o conteúdo de texto de vários URLs, você pode usar um código como este:

```
Começamos com um array de URLs
const urls = [ /* zero ou
mais URLs aqui */ ]; // E convertemos em um array de
objetos Promises
promises = urls.map(url =>
fetch(url).then(r => r.text())); // Agora obtenha uma
Promise para executar todas essas Promises em
parallel
Promise.all(promises)
.then(bodies => { /* faça algo com o array of strings
*/ })
```

```
.catch(e => console.error(e));
```

Promise.all() é um pouco mais flexível do que o descrito anteriormente. A matriz de entrada pode conter objetos Promise e valores não Promise. Se um elemento da matriz não for uma Promessa, ele será tratado como se fosse o valor de uma Promessa já cumprida e será simplesmente copiado inalterado para a matriz de saída.

A promessa retornada por Promise.all() é rejeitada quando qualquer uma das promessas de entrada é rejeitada. Isso acontece imediatamente após a primeira rejeição e pode acontecer enquanto outras promessas de entrada ainda estão pendentes. No ES2020, Promise.allSettled() recebe uma matriz de inputPromises e retorna uma Promise, assim como Promise.all(). ButPromise.allSettled() nunca rejeita a promessa retornada e não cumpre essa promessa até que todas as promessas de entrada tenham sido liquidadas. A Promise é resolvida para uma matriz de objetos, com um objeto para cada entrada Promise. Cada um desses objetos retornados tem uma propriedade de status definida como "cumprido" ou "rejeitado". Se o status for "cumprido", o objeto também terá uma propriedade value que fornece o valor de atendimento. Ese o status for "rejeitado", o objeto também terá uma propriedade reasonque fornece o valor de erro ou rejeição da promessa correspondente:

```
Promise.allSettled([Promise.resolve(1),
Promise.reject(2),3]).then(results => {
  Resultados[0] => { status: "cumprido", valor: 1 }
  Resultados[1] => { status: "rejeitado", motivo: 2 }
  Resultados[2] => { status: "cumprido", valor: 3 }
}); 2]
```

Ocasionalmente, você pode querer executar várias promessas de uma só vez, mas pode se preocupar apenas com o valor da primeira a cumprir. Nesse caso, você pode usar `Promise.race()` em vez de `Promise.all()`. Ele retorna uma Promise que é cumprida ou rejeitada quando a primeira das Promises na matriz de entrada é cumprida ou rejeitada. (Ou, se houver algum valor que não seja Promise na matriz de entrada, ele simplesmente retornará o primeiro deles.)

13.2.6 Fazer promessas

Usamos a função de retorno de promessa `fetch()` em muitos dos exemplos anteriores porque é uma das funções mais simples construídas para navegadores da web que retorna uma promise. Nossa discussão sobre Promises também se baseou em funções hipotéticas de retorno de Promise `getJSON()` e `wait()`. As funções escritas para retornar Promises são realmente bastante úteis, e esta seção mostra como você pode criar suas próprias APIs baseadas em Promise. Em particular, mostraremos implementações de `getJSON()` e `wait()`.

PROMESSAS BASEADAS EM OUTRAS PROMESSAS Se é fácil escrever uma função que retorna uma promise se você tiver alguma outra função de retorno de promise para começar. Dada uma promise, você sempre pode criar (e retornar) uma nova chamando `.then()`. Portanto, se usarmos a função `fetch()` existente como ponto de partida, podemos escrever `getJSON()` assim:

```
function getJSON(url) {  
  return fetch(url).then(resposta => resposta.json());}
```

O código é trivial porque o objeto Response da API fetch() tem um método json() predefinido. O método json() retorna aPromise, que retornamos de nosso retorno de chamada (o retorno de chamada é uma arrowfunction com um corpo de expressão única, portanto, o retorno é implícito), então thePromise retornado por getJSON() resolve para a Promise retornada byresponse.json(). Quando essa Promise é cumprida, a Promisereturned by getJSON() é cumprida com o mesmo valor. Observe que não há tratamento de erros nesta implementação getJSON(). Em vez de verificar response.ok e o cabeçalho Content-Type, em vez disso, apenas permitimos que o método json() rejeite a promessa retornada com umSyntaxError se o corpo da resposta não puder ser analisado como JSON.

Vamos escrever outra função de retorno de promessa, desta vez usandogetJSON() como a fonte da promessa inicial:

```
function getHighScore() {  
  return getJSON("/api/usuário/perfil").then(perfil  
    => perfil.highScore);}
```

Estamos assumindo que essa função faz parte de algum tipo de jogo baseado na Web e que a URL "/api/user/profile" retorna uma estrutura de dados formatada em JSON que inclui uma propriedade highScore.

PROMESSAS BASEADAS EM VALORES

SÍNCRONOS **ou** **simplicamente**, pode ser necessário implementar uma API baseada em Promise existente e retornar uma Promise de uma função, mesmo que a computação a ser executada não exija nenhuma operação assíncrona. Nesse caso, os métodos estáticos

Promise.resolve() e Promise.reject() farão o que você quiser. Promise.resolve() recebe um valor como seu argumento único e retorna uma promessa que será imediatamente (massíncrona) cumprida para esse valor. Da mesma forma, Promise.reject() recebe um único argumento e retorna um Promise que será rejeitado com esse valor como motivo. (Para ser claro: as promessas retornadas por esses métodos estáticos ainda não são cumpridas ou rejeitadas quando são retornadas, mas serão cumpridas ou rejeitadas imediatamente após o término da execução do bloco de código síncrono atual. Normalmente, isso acontece em alguns milissegundos, a menos que haja muitas tarefas assíncronas pendentes esperando para serem executadas.)

Lembre-se de §13.2.3 que uma promessa resolvida não é a mesma coisa que uma promessa cumprida. Quando chamamos Promise.resolve(), normalmente passamos o valor de fulfillment para criar um objeto Promise que muito em breve será cumprido para esse valor. No entanto, o método não é namedPromise.fulfill(). Se você passar uma Promise p1 toPromise.resolve(), ela retornará uma nova Promise p2, que é resolvida imediatamente, mas que não será cumprida ou rejeitada até que p1 seja cumprida ou rejeitada.

É possível, mas incomum, escrever uma função baseada em Promise em que thevalue é calculado de forma síncrona e retornado de forma assíncrona withPromise.resolve(). É bastante comum, no entanto, ter casos especiais síncronos dentro de uma função assíncrona, e você pode lidar com esses casos especiais com Promise.resolve() e Promise.reject(). Em particular, se você detectar condições de erro (como valores de argumento incorretos) antes de iniciar um

, você pode relatar esse erro retornando uma Promise criada withPromise.reject(). (Você também pode simplesmente lançar um erro de forma síncrona nesse caso, mas isso é considerado uma forma ruim porque, em seguida, o chamador de sua função precisa escrever uma cláusula synchronouscatch e usar um método assíncrono .catch() para lidar com erros.) Finalmente, Promise.resolve() às vezes é útil para criar a promessa inicial em uma cadeia de promessas. Veremos alguns exemplos que o usam dessa maneira.

PROMESSAS DO ZERO Para getJSON() e getHighScore(), começamos chamando uma função existente para obter uma promessa inicial e criamos e retornamos uma nova promessa chamando o método .then() dessa promessa inicial. Mas que tal escrever uma função de retorno de promessa quando você não pode usar outra função de retorno de promessa como ponto de partida? Nesse caso, você usa o construtor Promise() para criar um objeto newPromise sobre o qual você tem controle total. Veja como funciona: você invoca o construtor Promise() e passa uma função como apenas argumento. A função que você passa deve ser escrita para esperar dois parâmetros, que, por convenção, devem ser nomeados resolver e rejeitar. O construtor chama sua função de forma síncrona argumentos withfunction para os parâmetros resolve e reject. Depois de chamar sua função, o construtor Promise() retorna a promessa recém-criada. Essa promessa retornada está sob o controle da função que você passou para o construtor. Essa função deve executar alguma operação assíncrona e, em seguida, chamar a função de resolução para resolver ou cumprir a promessa retornada ou chamar a função de rejeição para

rejeitar a promessa retornada. Sua função não precisa ser assíncrona: ela pode chamar resolve ou rejeitar de forma síncrona, mas a promisePromise ainda será resolvida, cumprida ou rejeitada de forma assíncrona se você fizer isso.

Pode ser difícil entender as funções passadas para uma função passada para um construtor apenas lendo sobre isso, mas espero que alguns exemplos deixem isso claro. Veja como escrever a função wait() baseada em promessas que usamos em vários exemplos no início do capítulo:

```
função espera (duração) {  
    Criar e retornar uma nova Promisereturn new  
    Promise((resolve, reject) => { // Estes controlam a  
    Promessa  
        Se o argumento for inválido, rejeite o Promiseif  
        (duração < 0) {  
            reject(new Error("Viagem no tempo ainda não  
            implementado"));  
            // Caso contrário, aguarde de forma assíncrona  
            // e, em seguida, resolva  
            a Promessa.  
            setTimeout invocará resolve() sem  
            argumentos, o que significa  
            que a Promessa cumprirá com o indefinido  
            valor.  
            setTimeout(resolve, duration);});}  
}
```

Observe que o par de funções que você usa para controlar o destino de aPromise criado com o construtor Promise() é namedresolve() e reject(), not fulfill() e reject(). Se você passar uma Promise para resolve(), a Promise retornada resolverá essa nova Promise. Muitas vezes, no entanto, você passará uma não-promessa

valor, que cumpre a promessa retornada com esse valor.

O exemplo 13-1 é outro exemplo de uso do construtor `Promise()`. Este implementa nossa função `getJSON()` para uso inNode, onde a API `fetch()` não está integrada. Lembre-se de que começamos este capítulo com uma discussão sobre retornos de chamada e eventos assíncronos. Este exemplo usa retornos de chamada e manipuladores de eventos e é uma boa demonstração, portanto, de como podemos implementar APIs baseadas em promessas em cima de outros estilos de programação assíncrona.

Exemplo 13-1. Uma função `getJSON()` assíncrona

```
const http = require("http");

function getJSON(url) {
    Criar e retornar uma nova Promisereturn
    new Promise((resolve, reject) => {
        Inicie uma solicitação HTTP GET para o URLrequest =
        http.get(url, response => { // chamado quando
A resposta começa
            Rejeite a promessa se o status HTTP estiver
            erradose (response.statusCode !== 200) {
                reject(new Error('Status HTTP
${response.statusCode}'));
                resposta.resume(); para não vazarmos memória
            }// E rejeite se os cabeçalhos de resposta
            estiverem erradoselse if
            (response.headers["content-type"] !==
"aplicativo/json") {
                reject(new Error("Tipo de conteúdo inválido"));
                response.resume();// não vaza memória}else {//
Caso contrário, registra eventos para ler o corpo
da resposta
                seja corpo = "";
                resposta.setEncoding("utf-8");

```

```

        response.on("dados", chunk => { body += chunk;
});

        response.on("end", () => {
    Quando o corpo da resposta estiver concluído, tente
para analisá-lo
    tente {
        let parsed = JSON.parse(body); // Se
        for analisado com sucesso, preencha
a promessa
        resolve(analisado);
    } capture(e) {
        Se a análise falhar, rejeite o
Prometer
        rejeitar(e);});});}); Também rejeitamos a promessa
se a solicitação falhar

antes de nós
até mesmo obter uma resposta (como quando a rede é
para baixo)
request.on("erro", erro => {
rejeitar(erro);});};}

```

13.2.7 Promessas em sequênciа

Promise.all() facilita a execu o de um n mero arbitr rio de Promises em paralelo. E as cadeias de promessas facilitam a expresso de uma sequ ncia de um n mero fixo de promessas. No entanto, executar um n mero arbitr rio de Promises em sequ ncia   mais complicado. Suponha, por exemplo, que voc  tenha uma matriz de URLs para buscar, mas que, para evitar sobrecarregar sua rede, voc  deseja busc -las uma de cada vez. Se o array for de comprimento arbitr rio e conte do desconhecido, voc  n o poder  escrever uma promessa

cadeia com antecedência, então você precisa construir um dinamicamente, com código como este:

```
function fetchSequentially(urls) {  
    Armazenaremos os corpos da URL aqui à medida que  
    os buscamos  
    const bodies = [];  
  
    Aqui está uma função de retorno de promessa que busca  
    onebody  
    function fetchOne(url) {  
        return fetch(url)  
            .then(resposta =>  
                resposta.text().then(corpo => {  
                    Salvamos o corpo na matriz e estamos  
                    intencionalmente  
                    omitindo um valor de retorno aqui (retornando  
                    indefinido)  
                    corpos.push(corpo);})  
                ;  
    }  
}
```

Comece com uma promessa que será cumprida imediatamente
(com valor indefinido)

```
let p = Promise.resolve(indefinido);
```

Agora percorra os URLs desejados, construindo uma
Promisechain
de comprimento arbitrário, buscando um URL em cada estágio
da cadeia

```
for(url de urls) {  
    p = p.then(() => fetchOne(url));}
```

Quando a última promessa dessa cadeia for cumprida, então o
bodies está pronta. Então, vamos devolver uma promessa
para isso
matriz de corpos. Observe que não incluímos nenhum
errorhandler:
queremos permitir que os erros se propaguem para o
chamador.
p.then(() => corpos);}

Com essa função `fetchSequentially()` definida, poderíamos buscar as URLs uma de cada vez com um código muito parecido com o código `fetch-in-parallel` que usamos anteriormente para demonstrar `Promise.all()`:

```
fetchSequentially(urls)
  .then(bodies => { /* faça algo com o array ofstrings
    */ })
  .catch(e => console.error(e));
```

A função `fetchSequentially()` começa criando uma Promise que será cumprida imediatamente após o retorno. Em seguida, ele cria uma cadeia longa e linearPromise a partir dessa Promise inicial e retorna a última Promise na cadeia. É como montar uma fileira de dominós e depois derrubar o primeiro.

Há outra abordagem (possivelmente mais elegante) que podemos adotar. Em vez de criar as promessas com antecedência, podemos fazer com que o retorno de chamada para cada promessa crie e retorne a próxima promessa. Ou seja, em vez de criar e encadear um monte de promessas, criamos promessas que se resolvem em outras promessas. Em vez de criar uma cadeia de promessas semelhante a um dominó, estamos criando uma sequência de promessas aninhadas uma dentro da outra como um conjunto de bonecas matryoshka. Com essa abordagem, nosso código pode retornar a primeira Promise (mais externa), sabendo que ela eventualmente cumprirá (ou rejeitará!) o mesmo valor que a última Promise (mais interna) da sequência. A função `promiseSequence()` a seguir foi escrita para ser genérica e não é específica para busca de URL. Está aqui no final de nossa discussão sobre Promessas porque é complicado. Se você leu este capítulo com atenção, no entanto, espero que seja capaz de entender como ele funciona. Em particular, observe que a função aninhada dentro

`promiseSequence()` parece chamar a si mesmo recursivamente, mas como a chamada "recursiva" é por meio de um método `then()`, não há realmente nenhuma recursão tradicional acontecendo:

Esta função recebe um array de valores de entrada e uma função "promiseMaker".// Para qualquer valor de entrada x no array, promiseMaker(x) deverá retornar um Promise// que será cumprido como um valor de saída. Esta função retorna uma Promise// que cumpre um array dos valores de saída calculados.//// Em vez de criar as Promises de uma só vez e deixá-las rodar em paralelo, no entanto, promiseSequence() executa apenas uma Promise por vez// e não chama promiseMaker() para um valor até que a Promise anterior// tenha cumprido.
`function promiseSequence(inputs, promiseMaker) {`

Faça uma cópia privada do array que podemos modificar
`inputs = [... entradas];`

Aqui está a função que usaremos como um Promise callback
Esta é a magia pseudo-recursiva que faz tudo funcionar.

funcção handleNextInput(saídas) {
 if (entradas.comprimento === 0) {
 Se não houver mais entradas, retorne
A matriz
 de saídas, finalmente cumprindo esta promessa
e todos os
 Promises.return resolvido anterior, mas não
 cumprido else {

Se ainda houver valores de entrada a serem processados,
então vamos
 retorna um objeto Promise, resolvendo o objeto
Prometer
 com o valor futuro de um novo Promise.let
 nextInput = inputs.shift(); Obtenha o próximo
 valor de entrada,

```
        return promiseMaker(nextInput) calcule o
próximo valor de saída,
        Em seguida, crie uma nova matriz de saídas com o
Novo valor de saída
        .then(output => outputs.concat(output))// Em
        seguida, "reurse", passando o novo, mais longo,
matriz de saídas
        .then(handleNextInput);
    }

Comece com uma promessa que atenda a uma matriz vazia e
use
a função acima como seu callback.return
Promise.resolve([]).then(handleNextInput);
```

Essa função promiseSequence() é intencionalmente genérica.
Podemos usá-lo para buscar URLs com código como este:

```
Dada uma URL, retornamos uma Promise que cumpre a função
textfunction do corpo da URL fetchBody(url) { return
fetch(url).then(r =>r.text()); }Use-o para buscar
sequencialmente um monte de URLs
bodiespromiseSequence(urls, fetchBody)

.then(bodies => { /* faça algo com o array ofstrings
*/
})
.catch(console.error);
```

13.3 assíncrono e aguardar

O ES2017 apresenta duas novas palavras-chave (async e await) que representam uma mudança de paradigma na programação JavaScript assíncrona. Essas novas palavras-chave simplificam drasticamente o uso de promessas e nos permitem escrever código assíncrono baseado em promessas que se parece com código síncrono que bloqueia enquanto aguarda respostas da rede ou

outros eventos assíncronos. Embora ainda seja importante entender como as promessas funcionam, grande parte de sua complexidade (e às vezes até mesmo sua própria presença!) desaparece quando você as usa com um assíncrono e espera.

Como discutimos anteriormente no capítulo, o código assíncrono não pode retornar um valor ou lançar uma exceção da maneira que o código síncrono regular pode. E é por isso que as promessas são projetadas da maneira que são. O valor de uma promessa cumprida é como o valor de retorno de uma função síncrona. E o valor de uma promessa rejeitada é como um valor gerado por uma função assíncrona. Esta última semelhança é explicitada pela nomeação do método `.catch()`. `async` e `await` pegam o código eficiente baseado em promessas e ocultam as promessas para que seu código assíncrono possa ser tão fácil de ler e raciocinar quanto o código síncrono ineficiente e bloqueador.

13.3.1 Expressões de espera

A palavra-chave `await` pega uma `Promise` e a transforma de volta em um `returnvalue` ou uma exceção gerada. Dado um objeto `Promise` `p`, a expressão `await p` aguarda até que `p` se estabilize. Se `p` cumprir, então o valor de `await p` é o valor de cumprimento de `p`. Por outro lado, se `p` for rejeitado, a expressão `await p` lança o valor de rejeição de `p`. Normalmente não usamos `await` com uma variável que contém uma promessa; em vez disso, nós o usamos antes da invocação de uma função que retorna uma promessa:

```
let resposta = await fetch("/api/usuário/perfil");
let perfil = await response.json();
```

É fundamental entender imediatamente que a palavra-chave await não faz com que seu programa seja bloqueado e literalmente não faça nada até que o specifiedPromise seja resolvido. O código permanece assíncrono e o await simplesmente disfarça esse fato. Isso significa que qualquer código que use await é assíncrono.

13.3.2 Funções assíncronas

Como qualquer código que usa await é assíncrono, há uma regra crítica: você só pode usar a palavra-chave await em funções que foram declaradas com a palavra-chave async. Aqui está uma versão da função getHighScore() do início do capítulo, reescrita para usar async e await:

```
função assíncrona getHighScore() {  
  let resposta = await fetch("/api/usuário/perfil");  
  let perfil = await response.json(); retornar  
  profile.highScore;}
```

Declarar uma função assíncrona significa que o valor de retorno da função será uma Promise mesmo que nenhum código relacionado a Promise apareça no corpo da função. Se uma função assíncrona parecer retornar normalmente, o objeto Promise que é o valor de retorno real da função será resolvido para esse valor de retorno aparente. E se uma função assíncrona parece lançar uma exceção, o objeto Promise que ela retornará será rejeitado com essa exceção.

A função getHighScore() é declarada assíncrona, portanto, retorna aPromise. E como ele retorna uma promessa, podemos usar o método await

palavra-chave com ele:

```
displayHighScore(agine aguarde getHighScore());
```

Mas lembre-se, essa linha de código só funcionará se estiver dentro de outra função assíncrona! Você pode aninhar expressões await dentro de asyncfunctions tão profundamente quanto desejar. Mas se você estiver no nível superior ou estiver dentro de uma função que não é assíncrona por algum motivo, não poderá usar await e terá que lidar com uma Promise retornada da maneira regular:

```
getHighScore().then(displayHighScore).catch(console.error);
```

Você pode usar a palavra-chave `async` com qualquer tipo de função. Ele funciona com a palavra-chave `function` como uma instrução ou como uma expressão. Funciona com funções de seta e com o método atalho `forma in classes` e literais de objetos. (Veja o Capítulo 8 para mais informações sobre as várias maneiras de escrever funções.)

13.3.3 Aguardando várias promessas

Suponha que escrevemos nossa função `getJSON()` usando `async`:

```
função assíncrona getJSON(url) {  
let resposta = await fetch(url); let  
corpo = aguardar response.json();  
corpo de retorno;}
```

E agora suponha que queremos buscar dois valores JSON com `thisfunction`:

```
let valor1 = await getJSON(url1);
```

```
let valor2 = await getJSON(url2);
```

O problema com esse código é que ele é desnecessariamente sequencial: a busca da segunda URL não começará até que a primeira busca seja concluída. Se a segunda URL não depender do valor obtido da primeira URL, provavelmente devemos tentar buscar os dois valores ao mesmo tempo. Este é um caso em que a natureza baseada em Promise de `asyncfunctions` é mostrada. Para aguardar um conjunto de funções assíncronas executadas simultaneamente, usamos `Promise.all()` da mesma forma que faríamos se estivéssemos trabalhando diretamente com Promises:

```
let [valor1, valor2] = await  
Promise.all([getJSON(url1),getJSON(url2)]);
```

13.3.4 Detalhes da implementação

Finalmente, para entender como funcionam as funções assíncronas, pode ser útil pensar sobre o que está acontecendo nos bastidores.

Suponha que você escreva uma função assíncrona como esta:

```
função assíncrona f(x) { /* corpo */ }
```

Você pode pensar nisso como uma função de retorno de promessa encapsulada no corpo de sua função original:

```
função f(x) {  
    return new Promise(function(resolve, reject) {  
        tente {  
            resolve((function(x) { /* corpo */ })  
(x));}  
        captura(e) {  
  
            rejeitar(e);  
        }  
    });  
}
```

```
});}
```

É mais difícil expressar a palavra-chave `await` em termos de uma transformação de sintaxe como esta. Mas pense na palavra-chave `await` como um marcador que divide um corpo de função em partes síncronas separadas. Um interpretador ES2017 pode dividir o corpo da função em uma sequência de subfunções separadas, cada uma das quais é passada para o método `then()` da `Promise` marcada como espera que a precede.

13.4 Iteração assíncrona

Começamos este capítulo com uma discussão sobre a assincronia baseada em retorno de chamada e evento e, quando introduzimos as Promises, observamos que elas eram úteis para cálculos assíncronos de disparo único, mas não eram adequadas para uso com fontes de eventos assíncronos repetitivos, como `assetInterval()`, o evento "click" em um navegador da Web ou o evento "data" em um fluxo de Node. Como as promessas únicas não funcionam para sequências de eventos assíncronos, também não podemos usar funções assíncronas regulares e as instruções `await` para essas coisas.

No entanto, o ES2018 fornece uma solução. Os iteradores assíncronos são como os iteradores descritos no Capítulo 12, mas são baseados em promessas e devem ser usados com uma nova forma do loop `for/of:for/await`.

13.4.1 O loop `for/await`

O nó 12 torna seus fluxos legíveis de forma assíncrona iterável. Este

significa que você pode ler blocos sucessivos de dados de um fluxo com um loop afor/await como este:

```
const fs = require("fs");

função assíncrona parseFile(nome do arquivo) {
let stream = fs.createReadStream(nome do arquivo, {
encoding:"utf-8"});
    for await (let pedaço de fluxo) {
        parseChunk(pedaço); Suponha que parseChunk() esteja definido
alhures
}}
```

Como uma expressão await regular, o loop for/await é baseado em Promise. Grosso modo, o iterador assíncrono produz aPromise e o loop for/await aguarda que essa Promise seja cumprida, atribui o valor de cumprimento à variável de loop e executa o corpo do loop. E então começa de novo, obtendo outra promessa do iterador e esperando que essa nova promessa seja cumprida.

Suponha que você tenha uma matriz de URLs:

```
const urls = [url1, url2, url3];
```

Você pode chamar fetch() em cada URL para obter uma matriz de promessas:

```
const promessas = urls.map(url => fetch(url));
```

Vimos anteriormente no capítulo que agora poderíamos usar Promise.all() para esperar que todas as promessas na matriz fossem cumpridas. Mas suponha que queremos os resultados da primeira busca assim que eles estiverem disponíveis e não queremos esperar que todos os URLs sejam

Buscado. (Claro, a primeira busca pode demorar mais do que qualquer uma das outras, então isso não é necessariamente mais rápido do que usar `Promise.all()`.) As matrizes são iteráveis, então podemos iterar através da matriz de promessas com um loop `for/of` regular:

```
for(const promessa de promessas) {  
    resposta = aguardar  
    promessa; identificador  
(resposta);}
```

Este código de exemplo usa um loop `for/of` regular com um iterador regular. Mas como esse iterador retorna Promises, também podemos usar o `newfor/await` para um código um pouco mais simples:

```
for await (const resposta de promessas) {  
    identificador  
(resposta);}
```

Nesse caso, o loop `for/await` apenas cria a chamada `await` no loop e torna nosso código um pouco mais compacto, mas os dois exemplos fazem exatamente a mesma coisa. É importante ressaltar que ambos os exemplos só funcionarão se estiverem dentro de funções declaradas assíncronas; Um loop `for/await` não é diferente de uma expressão `await` regular dessa maneira.

É importante perceber, no entanto, que estamos usando `for/await` com um iterador regular neste exemplo. As coisas são mais interessantes com iteradores totalmente assíncronos.

13.4.2 Iteradores assíncronos

Vamos revisar algumas terminologias do Capítulo 12. Um objeto iterável é

um que pode ser usado com um loop for/of. Ele define um método com o nome simbólico `Symbol.iterator`. Esse método retorna um objeto iterator. O objeto iterador tem um método `next()`, que pode ser chamado repetidamente para obter os valores do objeto iterável. O método `next()` do objeto iterador retorna objetos de resultado de iteração. O objeto de resultado de iteração tem uma propriedade `value` e/ou uma propriedade `done`.

Os iteradores assíncronos são bastante semelhantes aos iteradores regulares, mas há duas diferenças importantes. Primeiro, um objeto iterável de forma assíncrona implementa um método com o nome simbólico `Symbol.asyncIterator` em vez de `Symbol.iterator`. (Como vimos anteriormente, `for/await` é compatível com objetos iteráveis regulares, mas prefere objetos iteráveis de forma assíncrona e tenta o método `Symbol.asyncIterator` antes de tentar o método `Symbol.iterator`.) Em segundo lugar, o método `next()` do iterador assíncrono retorna uma promessa que é resolvida para um objeto iteratorresult em vez de retornar um objeto de resultado do iterador diretamente.

NOTA

Na seção anterior, quando usamos `for/await` em uma matriz regular e síncrona iterável de Promises, estávamos trabalhando com objetos de resultado do iterador síncrono nos quais a propriedade `value` era um objeto Promise, mas a propriedade `done` era síncrona. Os iteradores assíncronos verdadeiros retornam Promises para objetos de resultado de iteração, e as propriedades `value` e `done` são assíncronas. A diferença é sutil: com iteradores assíncronos, a escolha sobre quando a iteração termina pode ser feita de forma assíncrona.

13.4.3 Geradores assíncronos

Como vimos no Capítulo 12, a maneira mais fácil de implementar um iterador é muitas vezes usar um gerador. O mesmo vale para iteradores assíncronos, que podemos implementar com funções geradoras que declaramos assíncronas. Um gerador assíncrono tem os recursos das funções assíncronas e os recursos dos geradores: você pode usar await como faria em uma função assíncrona regular e pode usar yield como faria em um gerador regular. Mas os valores que você produz são automaticamente encapsulados em Promises. Até mesmo a sintaxe para geradores assíncronos é uma combinação: função assíncrona e função * combinam em função assíncrona *. Aqui está um exemplo que mostra como você pode usar o gerador assíncrono e um loop for/await para executar repetidamente o código em intervalos fixos usando a sintaxe de loop em vez de uma função de retorno de chamada setInterval():

Um wrapper baseado em Promise em torno de setTimeout() com o qual podemos usar await.// Retorna uma Promise que cumpre o número especificado de milissegundosfunction elapsedTime(ms) {

```
return new Promise(resolve => setTimeout(resolve, ms));}
```

*Uma função geradora assíncrona que incrementa um contador e o produz // um número especificado (ou infinito) de vezes em um intervalo especificado.função assíncrona**

```
clock(interval, max=Infinity) {
```

```
for(let count = 1; count <= max; count++) { // loop regular
    await elapsedTime(interval);           esperar
    contagem de tempo
    para passar yield;
    balcão
}
```

*Uma função de teste que usa o gerador assíncrono
withfor/awaitasync function test() { // Async para que
possamos usar for/await*

```
for await (let tick of clock(300, 100)) { // Loop 100  
vezes a cada 300ms  
console.log(tique);}
```

13.4.4 Implementando iteradores assíncronos

Em vez de usar geradores assíncronos para implementar iteradores assíncronos, também é possível implementá-los diretamente definindo um objeto com um Symbol.método `asyncIterator()` que retorna um objeto com um método `next()` que retorna uma promessa que resolve para o objeto de resultado do aníterator. No código a seguir, reimplementamos a função `clock()` do exemplo anterior para que ela não seja um gerador e, em vez disso, apenas retorne um objeto iterável de forma assíncrona.

Observe que o método `next()` neste exemplo não retorna explicitamente uma promessa; Em vez disso, apenas declaramos `next()` como assíncrono:

```
function clock(intervalo, max=Infinito) {  
Uma versão prometida de setTimeout com a qual podemos  
usar await.  
Observe que isso leva um tempo absoluto em vez de um  
intervalo.  
    função até(tempo) {  
        return new Promise(resolve => setTimeout(resolve,  
        hora - Data.agora()));  
    }  
  
Retornar um objeto iterável de forma  
assíncrona return {  
    startTime: Date.now(), Lembre-se de quando começamos  
    contagem: 1, Lembrar qual iteração  
estamos em
```

```

        assíncrono next() {      O método next() torna
este é um iterador
            if (this.count > max) {      Terminamos?
                return { done: true };  Resultado da iteração
indicando concluído
                }// Descubra quando a próxima iteração deve
começar
            let targetTime = this.startTime + this.count *
intervalo;
            wait until that time, await
            until(targetTime); // e retorna o valor count
em uma iteração
        resultado.
            return { value: this.count++ };},// Este método
significa que este objeto iterador é
também um iterável.
[Symbol.asyncIterator]() { return this; };}

```

Esta versão baseada em iterador da função clock() corrige uma falha na versão baseada em gerador. Observe que, neste código mais recente, visamos o tempo absoluto em que cada iteração deve começar e subtraímos o tempo atual disso para calcular o intervalo que passamos para setTimeout(). Se usarmos clock() com um loop for/await, thisversion executará iterações de loop com mais precisão no intervalo especificado porque leva em conta o tempo necessário para realmente executar o corpo do loop. Mas essa correção não é apenas sobre a precisão do tempo. O for/awaitloop sempre aguarda que a Promise retornada por uma iteração seja cumprida antes de iniciar a próxima iteração. Mas se você usar o iterador assíncrono sem um loop for/await, não há nada que impeça você de chamar o método next() sempre que quiser. Com a versão baseada em gerador de clock(), se você chamar o

`next()` três vezes sequencialmente, você obterá três promessas que serão cumpridas quase exatamente ao mesmo tempo, o que provavelmente não é o que você deseja. A versão baseada em iterador que implementamos aqui não tem esse problema.

O benefício dos iteradores assíncronos é que eles nos permitem representar fluxos de eventos ou dados assíncronos. A função `clock()` discutida anteriormente era bastante simples de escrever porque a fonte da assincronia eram as chamadas `setTimeout()` que estávamos fazendo. Mas quando estamos tentando trabalhar com outras fontes assíncronas, como o acionamento de manipuladores de eventos, torna-se substancialmente mais difícil implementar iteradores assíncronos - normalmente temos uma única função de manipulador de eventos que responde a eventos, mas cada chamada para o método `next()` do iterador deve retornar um objeto Promise distinto, e várias chamadas para `next()` podem ocorrer antes que o `firstPromise` seja resolvido. Isso significa que qualquer método de iterador assíncrono deve ser capaz de manter uma fila interna de promessas que ele resolve na ordem em que os eventos assíncronos estão ocorrendo. Se encapsularmos esse comportamento de enfileiramento de promessas em uma classe `AsyncQueue`, será muito mais fácil escrever iteradores assíncronos com base em `AsyncQueue`.

3

A classe `AsyncQueue` a seguir tem os métodos `enqueue()` e `dequeue()` como seria de esperar para uma classe de fila. O método `dequeue()` retorna uma promessa em vez de um valor real, no entanto, o que significa que não há problema em chamar `dequeue()` antes que `enqueue()` tenha sido chamado. A classe `AsyncQueue` também é um iterador assíncrono e destina-se a ser usada com um `for/awaitloop` cujo corpo é executado uma vez cada vez que um novo valor é assíncrono

Enfileirado. (AsyncQueue tem um método close(). Uma vez chamado, nenhum outro valor pode ser enfileirado. Quando uma fila fechada estiver vazia, o loop for/await interromperá o loop.)

Observe que a implementação de AsyncQueue não usa async orawait e, em vez disso, funciona diretamente com Promises. O código é um pouco complicado e você pode usá-lo para testar sua compreensão do material que abordamos neste longo capítulo. Mesmo que você não entenda completamente a implementação de AsyncQueue, dê uma olhada no exemplo mais curto que a segue: ele implementa um iterador assíncrono simples, mas muito interessante, sobre AsyncQueue.

```
/**  
 * Uma classe de fila iterável de forma assíncrona.  
 Adicionar valores withenqueue()  
 * e remova-os com dequeue(). dequeue() retorna  
 aPromise, que  
 * significa que os valores podem ser removidos da  
 fila antes de serem enfileirados. O  
 * implementa [Symbol.asyncIterator] e next() para que possa  
 * ser usado com o loop for/await (que não terminará até  
 * O método close() é  
 chamado.)*/class AsyncQueue {  
  
    construtor() {  
        Valores que foram enfileirados, mas ainda não removidos da fila  
        são armazenados aqui  
        this.values = [];// Quando as promessas  
        são removidas da fila antes de suas  
        os valores correspondentes são  
        enfileirados, os métodos de resolução para essas promessas são  
        armazenados aqui.  
        this.resolvers = [];// Uma vez fechado, nenhum  
        outro valor pode ser enfileirado, e  
        não mais insatisfeito  
        Promessas  
        retornadas.this.closed = false;
```

```
}
```

```
enqueue(valor) {
    if (this.closed) {
        throw new Error("AsyncQueue
fechado");}if (this.resolvers.length >
0) {
    Se esse valor já tiver sido prometido,
resolver essa promessa
    const resolve = this.resolvers.shift();
    resolve(valor);}else {
```

```
Caso contrário, coloque-o na
fila
upthis.values.push(value);}}
```

```
dequeue() {
    if (this.values.length > 0) {
        Se houver um valor enfileirado, retorne um valor resolvido
Prometa para isso
        const valor = this.values.shift();
        return Promise.resolve(value);}senão
        se (this.closed) {

            Se não houver valores enfileirados e estivermos fechados,
Resolvido
retorne um
Promessa para o marcador de "fim de
fluxo"return
Promise.resolve(AsyncQueue.EOS);}else {

            Caso contrário, retorne uma promessa não
resolvida,// enfileirando a função de resolvedor
para uso posteriorreturn new Promise((resolve) => {
this.resolvers.push(resolve); });

        }}
```

```
fechar() {
    Depois que a fila for fechada, nenhum outro valor será
Enfileirado.
```

Portanto, resolva quaisquer promessas pendentes com o fim de marcador de fluxo

```
        while(this.resolvers.length > 0) {
            this.resolvers.shift()
            (AsyncQueue.EOS);}this.closed = true;}
```

Defina o método que torna essa classe assíncrona iterável

```
[Symbol.asyncIterator]() { return this; }
```

Defina o método que torna isso um iterador assíncrono.

```
O
dequeue() A promessa é resolvida para um valor ou para
o EOSsentinel se estivermos
fechado. Aqui, precisamos retornar uma promessa que
resolve para um
    resultado do iterador
    object.next() {
        return this.dequeue().then(value => (value ===
AsyncQueue.EOS)
                                ? { valor: indefinido,
feito: verdadeiro }
                                : { valor: valor, feito:
falso });
    }
}
```

*Um valor sentinel retorna por dequeue() para marcar
"fim do fluxo" quando closedAsyncQueue.EOS = Symbol("fim
do fluxo");*

Como essa classe AsyncQueue define os princípios básicos da iteração assíncrona, podemos criar nossos próprios iteradores assíncronos mais interessantes simplesmente enfileirando valores de forma assíncrona. Aqui está um exemplo que usa AsyncQueue para produzir um fluxo de eventos do navegador da Web que podem ser manipulados com um loop for/await:

Eventos de push do tipo especificado no especificado

```
document element// em um objeto AsyncQueue e retorna a fila
para uso como um evento streamfunction eventStream(elt,
type) {

    const q = new AsyncQueue(); Crie um
    fila
    elt.addEventListener(tipo, e=>q.enqueue(e)); Enfileirar
    eventos
    return q;}

função assíncrona handleKeys() {
Obtenha um fluxo de eventos de pressionamento de tecla e
faça um loop uma vez para cada um
for await (const evento de
eventStream(document, "keypress")) {
    console.log(event.key);}}
```

13.5 Resumo

Neste capítulo, você aprendeu:

- A maior parte da programação JavaScript do mundo real é assíncrona.
- Tradicionalmente, a assincronia tem sido tratada com eventos e funções de retorno de chamada. Isso pode ficar complicado, no entanto, porque você pode acabar com vários níveis de retornos de chamada aninhados dentro de outros retornos de chamada e porque é difícil fazer tratamento robusto de erros. As promessas fornecem uma nova maneira de estruturar funções de retorno de chamada. Se usadas corretamente (e, infelizmente, as promessas são fáceis de usar incorretamente), elas podem converter código assíncrono que teria sido aninhado em cadeias lineares de chamadas then () em que uma etapa assíncrona de um cálculo segue a outra. Além disso, as promessas permitem que você centralize seu tratamento de erros

- em uma única chamada `catch()` no final de uma cadeia de chamadas `ofthen()`. As palavras-chave `async` e `await` nos permitem escrever um código assíncrono baseado em promessas nos bastidores, mas que se parece com um código síncrono. Isso torna o código mais fácil de entender e raciocinar. Se uma função for `declareasync`, ela retornará implicitamente uma `Promise`. Dentro de uma `asyncfunction`, você pode aguardar uma `Promise` (ou uma função que retorna uma `Promise`) como se o valor `Promise` tivesse sido calculado de forma síncrona. Objetos que são iteráveis de forma assíncrona podem ser usados com o loop `afor/await`. Você pode criar `iterableobjects` de forma assíncrona implementando um método `[Symbol.asyncIterator]()` ou invocando uma função assíncrona `*generator`. Os iteradores assíncronos fornecem uma alternativa aos eventos de "dados" em fluxos no Node e podem ser usados para representar um fluxo de eventos de entrada do usuário no JavaScript do lado do cliente.

1 A classe XMLHttpRequest não tem nada em particular a ver com XML. No JavaScript moderno do lado do cliente, ele foi amplamente substituído pela API `fetch()`, que é abordada em §15.11.1. O exemplo de código mostrado aqui é o último exemplo baseado em XMLHttpRequest restante neste livro.

2 Normalmente, você pode usar `await` no nível superior do console do desenvolvedor de um navegador. E há uma proposta pendente para permitir o `await` de nível superior em uma versão futura do JavaScript.

3 Aprendi sobre essa abordagem de iteração assíncrona no blog do Dr. AxelRauschmayer, <https://2ality.com>.

Capítulo

14. Metaprogramação

Este capítulo aborda uma série de recursos avançados de JavaScript que não são comumente usados na programação do dia-a-dia, mas que podem ser valiosos para programadores que escrevem bibliotecas reutilizáveis e de interesse para qualquer pessoa que queira mexer nos detalhes sobre como os objetos JavaScript se comportam.

Muitos dos recursos descritos aqui podem ser descritos vagamente como "metaprogramação": se a programação regular está escrevendo código para manipular dados, então a metaprogramação está escrevendo código para manipular outro código. Em uma linguagem dinâmica como JavaScript, as linhas entre programação e metaprogramação são borradadas - até mesmo a capacidade simples de iterar sobre as propriedades de um objeto com um loop `for/in` pode ser considerada "meta" por programadores acostumados a linguagens mais estáticas.

Os tópicos de metaprogramação abordados neste capítulo incluem:

- §14.1 Controlando a enumerabilidade, exclusão e configurabilidade das propriedades do objeto
- Controlando a extensibilidade de objetos e criando objetos "selados" e "congelados"
- Consultando e configurando os protótipos de objetos

- §14.4 Ajustando o comportamento de seus tipos com well-knownSymbols
- §14.5 Criando DSLs (linguagens específicas de domínio) com funções de marca de modelo
- §14.6 Sondando objetos com métodos de reflexão
- §14.7 Controlando o comportamento do objeto com Proxy
- -----

14.1 Atributos de propriedade

As propriedades de um objeto JavaScript têm nomes e valores, é claro, mas cada propriedade também tem três atributos associados que especificam como essa propriedade se comporta e o que você pode fazer com ela:

O atributo gravável especifica se o valor de uma propriedade pode ou não ser alterado. O atributo enumerable especifica se a propriedade é enumerada pelo loop for/in e pelo método Object.keys(). O atributo configurável especifica se uma propriedade pode ser excluída e também se os atributos da propriedade podem ser alterados. As propriedades definidas em literais de objeto ou por atribuição comum a um objeto são graváveis, enumeráveis e configuráveis. Mas muitas das propriedades definidas pela biblioteca padrão JavaScript não são.

Esta seção explica a API para consultar e definir propertyattributes. Essa API é particularmente importante para os autores de bibliotecas porque:

- Ele permite que eles adicionem métodos para prototipar objetos e fazer

não enumeráveis, como métodos integrados. Isso permite que eles "bloquem" seus objetos, definindo propriedades que não podem ser alteradas ou excluídas. Lembre-se de §6.10.6 que, enquanto "propriedades de dados" têm um valor, "propriedades de acessador" têm um método getter e/ou setter. Para os fins desta seção, vamos considerar os métodos getter e setter de uma propriedade acessadora como atributos de propriedade. Seguindo essa lógica, diremos até que o valor de uma propriedade de dados também é um atributo. Assim, podemos dizer que uma propriedade tem um nome e quatro atributos. Os quatro atributos de uma propriedade de dados são value, gravável, enumerável e configurável. As propriedades do acessador não têm um atributo value ou um atributo gravável: sua capacidade de escrita é determinada pela presença ou ausência de um setter. Portanto, os quatro atributos de uma propriedade de acessador são get, set, enumerable e configurable.

Os métodos JavaScript para consultar e definir os atributos de uma propriedade usam um objeto chamado descritor de propriedade para representar o conjunto de quatro atributos. Um objeto descritor de propriedade tem propriedades com os mesmos nomes que os atributos da propriedade que ele descreve. Assim, o objeto descritor de propriedade de uma propriedade de dados tem propriedades nomeadas valor, gravável, enumerável e configurável. E o descritor de uma propriedade de acessador tem propriedades get e set em vez de value e writable. As propriedades graváveis, enumeráveis e configuráveis são valores booleanos, e as propriedades get e set são valores de função.

Para obter o descritor de propriedade para uma propriedade nomeada de um objeto especificado, chame Object.getOwnPropertyDescriptor():

```
Retorna {value: 1, writable:true,  
enumerable:true,configurable:true}Object.getOwnPropertyDescriptor({x: 1}, "x");
```

*Aqui está um objeto com um acessador somente leitura
propertyconst random = {*

```
get octet() { return Math.floor(Math.random()*256); },};
```

```
Retorna { get: /*func*/, set:undefined,  
enumerable:true,configurable:true}Object.getOwnPropertyDescriptor(random, "octeto");
```

```
Retorna undefined para propriedades herdadas e propriedades  
que não existem. Object.getOwnPropertyDescriptor({}, "x")//  
=>indefinido; nenhum  
propObject.getOwnPropertyDescriptor({}, "toString") //  
=>undefined; Herdada
```

Como o próprio nome indica,

`Object.getPrototypeOf()` funciona apenas para propriedades próprias. Para consultar os atributos de `inheritedProperties`, você deve percorrer explicitamente a cadeia de protótipos.

(See `Object.getPrototypeOf()` em §14.3); consulte também a função `similarReflect.getPrototypeOf()` em §14.6.)

Para definir os atributos de uma propriedade ou criar uma nova propriedade com os atributos especificados, chame

`Object.defineProperty()`, passando o objeto a ser modificado, o nome da propriedade a ser criada ou alterada e o objeto descritor de propriedade:

```
seja o = {};  
Comece sem nenhuma propriedade  
Adicione uma propriedade de dados não enumerável x  
com o valor 1.Object.defineProperty(o, "x", {  
    valor: 1,gravável:  
    verdadeiro,enumerá  
    vel: falso,
```

```
configurável: true});
```

Verifique se a propriedade está lá, mas não é enumerável.
`x// => 1`
`Object.keys(o) // => []`

Agora modifique a propriedade x para que ela seja somente leitura.
`Object.defineProperty(o, "x", { writable: false });`

Tente alterar o valor do property.
`x = 2; // Falha silenciosamente ou lança TypeError em strictmode.`
`x// => 1`

A propriedade ainda é configurável, então podemos alterar seu valor assim:
`Object.defineProperty(o, "x", { value: 2 });`
`o.x// => 2`

Agora altere x de uma propriedade de dados para um acessador.
`propertyObject.defineProperty(o, "x", { get: function() { return 0; } });`
`o.x// => 0`

O descriptor de propriedade que você passa para `Object.defineProperty()` não precisa incluir todos os quatro atributos. Se você estiver criando uma nova propriedade, os atributos omitidos serão considerados falsos ou indefinidos. Se você estiver modificando uma propriedade existente, os atributos omitidos serão simplesmente deixados inalterados. Observe que esse método altera uma propriedade própria existente ou cria uma nova propriedade própria, mas não alterará uma propriedade herdada. Veja também a função muito semelhante `Reflect.defineProperty()` em §14.6. Se você deseja criar ou modificar mais de uma propriedade por vez, use `Object.defineProperties()`. O primeiro argumento é o objeto

que deve ser modificado. O segundo argumento é um objeto que mapeia os nomes das propriedades a serem criadas ou modificadas para os descritores de propriedade dessas propriedades. Por exemplo:

```
let p = Object.defineProperties({}, {
  x: { valor: 1, gravável: verdadeiro, enumerável:
    verdadeiro, configurável: verdadeiro },
  y: { value: 1, wr
        r: {
          get() { return Math.sqrt(this.x*this.x +
            this.y*this.y); },
          enumerável:
          true,configurable:
          true}}); p.r// =>
Matemática.SQRT2
```

Esse código começa com um objeto vazio e, em seguida, adiciona duas propriedades de dados e uma propriedade de acessador somente leitura a ele. Ele se baseia no fato de queObject.defineProperties() retorna o objeto modificado (assim como Object.defineProperty()).

O método Object.create() foi introduzido no §6.2. Aprendemos que o primeiro argumento para esse método é o objeto protótipo para o objeto recém-criado. Este método também aceita um segundo argumento opcional, que é o mesmo que o segundo argumento toObject.defineProperties(). Se você passar um conjunto de propertydescriptors para Object.create(), eles serão usados para adicionarpropriedades ao objeto recém-criado.

Object.defineProperty() e

`Object.defineProperties()` lançará `TypeError` se a tentativa de criar ou modificar uma propriedade não for permitida. Isso acontece se você tentar adicionar uma nova propriedade a um objeto não extensível (consulte §14.2). Os outros motivos pelos quais esses métodos podem lançar `TypeError` têm a ver com os próprios atributos. O atributo gravável controla as tentativas de alterar o atributo `value`. E o atributo configurável controla tentativas de alterar os outros atributos (e também especifica se uma propriedade pode ser excluída). As regras não são totalmente diretas, no entanto. É possível alterar o valor de uma propriedade não gravável se essa propriedade for configurável, por exemplo. Além disso, é possível alterar uma propriedade de gravável para não gravável, mesmo que essa propriedade não seja configurável. Aqui estão as regras completas. Chamadas para `Object.defineProperty()` ou `Object.defineProperties()` que tentam violá-las geram um `TypeError`:

- Se um objeto não for extensível, você poderá editar suas próprias propriedades existentes, mas não poderá adicionar novas propriedades a ele. Se uma propriedade não for configurável, você não poderá alterar seus atributos
- configuráveis ou enumeráveis. Se uma propriedade de acessador não for configurável, você não poderá alterar seu método `getter` ou `setter` e não poderá alterá-lo para uma `dataproperty`. Se uma propriedade de dados não for configurável, você não poderá alterá-la para uma propriedade acessadora. Se uma propriedade de dados não for configurável, você não poderá alterar seu atributo gravável de `false` para `true`, mas poderá alterá-lo de `true` para `false`.

Se uma propriedade de dados não for configurável e não for gravável, você não poderá alterar seu valor. No entanto, você pode alterar o valor de uma propriedade configurável, mas não gravável (porque isso seria o mesmo que torná-la gravável, alterar o valor e convertê-lo novamente em não gravável). §6.7 descreveu a função `Object.assign()` que copia valores de propriedade de um ou mais objetos de origem para um objeto de destino. `Object.assign()` copia apenas propriedades enumeráveis e valores de propriedade, não atributos de propriedade. Isso normalmente é o que queremos, mas significa, por exemplo, que se um dos objetos de origem tiver uma propriedade acessadora, é o valor retornado pela função getter que é copiado para o objeto de destino, não a própria função getter. O exemplo 14-1 demonstra como podemos usar `Object.getOwnPropertyDescriptor()` e `Object.defineProperty()` para criar uma variante de `Object.assign()` que copia descritores de propriedade inteiros em vez de apenas copiar valores de propriedade.

Exemplo 14-1. Copiando propriedades e seus atributos de um objeto para outro

```
/*
 * Defina uma nova função Object.assignDescriptors() que
funcione como
 * Object.assign(), exceto que copia descritores de
propriedade de
 * objetos de origem no objeto de destino em vez de
apenas copiar
 * valores de propriedade. Esta função copia todas as
próprias propriedades, tanto
 * enumerável e não enumerável. E porque copia
descritores,
 * ele copia funções getter de objetos de origem e
substitui setter
 * no objeto de destino em vez de invocar esses getters e
```

** setters.** Object.assignDescriptors() propaga qualquer TypeErrors lançado por* Object.defineProperty(). Isso pode ocorrer se o objeto alvo estiver selado*

** ou congelado ou se qualquer uma das propriedades de origem tentar alterar um existente*

** propriedade não configurável no objeto de destino.** Observe que a propriedade assignDescriptors é adicionada a Objectwith*

** Object.defineProperty() para que a nova função possa ser criada como*

** uma propriedade não enumerável como Object.assign().*/Object.defineProperty(Object, "assignDescriptors", {*

Corresponde aos atributos de Object.assign()writable:

true,enumerable: false,configurable: true,// A função que é o valor da propriedade assignDescriptors.

```
valor: função(alvo, ... fontes) {
    for(let fonte das fontes) {
        for(let nome de Object.getOwnPropertyNames(fonte))
{
            deixe desc =
Object.getOwnPropertyDescriptor(origem, nome);
            Object.defineProperty(destino, nome, desc);}

            for(let símbolo de
Object.getOwnPropertySymbols(fonte)) {
                deixe desc =
Object.getOwnPropertyDescriptor(origem, símbolo);
                Object.defineProperty(destino, símbolo, desc);}}retornar
alvo;}};
```

```
let o = {c: 1, get count() {return this.c++;}}; Defineobject  
with getterlet p = Object.assign({}, o);// Copia a propriedade  
valueslet q = Object.assignDescriptors({}, o);// Copia a  
propriedade descriptorsp.count// => 1: Agora é apenas uma  
propriedade de dados sop.count// => 1: ... o contador não  
incrementa.q.count// => 2: Incrementado uma vez quando o  
copiamos pela primeira vez,q.count// => 3: ... mas copiamos o  
método getter para que ele seja incrementado.
```

14.2 Extensibilidade de objetos

O atributo extensível de um objeto especifica se novas propriedades podem ser adicionadas ao objeto ou não. Objetos JavaScript comuns são extensíveis por padrão, mas você pode alterar isso com as funções descritas nesta seção.

Para determinar se um objeto é extensível, passe-o para `Object.isExtensible()`. Para tornar um objeto não extensível, passe-o para `Object.preventExtensions()`. Depois de fazer isso, qualquer tentativa de adicionar uma nova propriedade ao objeto lançará um `TypeError` no modo estrito e simplesmente falhará silenciosamente sem um erro no modo não estrito. Além disso, a tentativa de alterar o protótipo (consulte §14.3) de um objeto não extensível sempre lançará um `TypeError`.

Observe que não há como tornar um objeto extensível novamente depois de torná-lo não extensível. Observe também que `callingObject.preventExtensions()` afeta apenas a extensibilidade do próprio objeto. Se novas propriedades forem adicionadas ao protótipo de um

extensível, o objeto não extensível herdará essas novas propriedades.

Duas funções semelhantes, `Reflect.isExtensible()` e `Reflect.preventExtensions()`, são descritas em §14.6.

A finalidade do atributo extensível é ser capaz de "bloquear" objetos em um estado conhecido e evitar adulterações externas. O atributo extensível de objetos é frequentemente usado em conjunto com os atributos configuráveis e graváveis de propriedades, e o JavaScript define funções que facilitam a configuração desses atributos juntos:

`Object.seal()` funciona como `Object.preventExtensions()`, mas além de tornar o objeto não extensível, também torna todas as próprias propriedades desse objeto não configuráveis. Isso significa que `newproperties` não podem ser adicionadas ao objeto e as propriedades existentes não podem ser excluídas ou configuradas. No entanto, as propriedades existentes que são graváveis ainda podem ser definidas. Não há como abrir um objeto lacrado. Você pode usar `Object.isSealed()` para determinar se um objeto está lacrado.

`Object.freeze()` bloqueia os objetos com ainda mais força. Além de tornar o objeto não extensível e suas propriedades não configuráveis, ele também torna todas as propriedades de dados do próprio objeto somente leitura. (Se o objeto tiver `accessorproperties` com métodos `setter`, eles não serão afetados e ainda poderão ser invocados por atribuição à propriedade.) Use `Object.isFrozen()` para determinar se um objeto está congelado. É importante entender que `Object.seal()` e `Object.freeze()` afetam apenas o objeto que são passados: eles têm

nenhum efeito sobre o protótipo desse objeto. Se você deseja bloquear completamente um objeto, provavelmente também precisará selar ou congelar os objetos na cadeia de protótipos.

`Object.preventExtensions()`, `Object.seal()` e `Object.freeze()` retornam o objeto que eles são passados, o que significa que você pode usá-los em invocações de funções aninhadas:

```
Crie um objeto selado com um protótipo congelado e uma  
property let não enumerável o =  
Object.seal(Object.create(Object.freeze({x: 1}),  
{y: {valor: 2, gravável:  
verdadeiro}}));
```

Se você estiver escrevendo uma biblioteca JavaScript que passa objetos para funções de retorno de chamada escritas pelos usuários de sua biblioteca, você pode usar `Object.freeze()` nesses objetos para evitar que o código do usuário os modifique. Isso é fácil e conveniente de fazer, mas há compensações: objetos congelados podem interferir em estratégias comuns de teste de JavaScript, por exemplo.

14.3 O atributo protótipo

O atributo `prototype` de um objeto especifica o objeto do qual ele herda propriedades. (Revise §6.2.3 e §6.3.2 para obter mais informações sobre protótipos e herança de propriedade.) Este é um atributo tão importante que geralmente dizemos simplesmente "o protótipo de o" em vez de "o protótipo de o". Lembre-se também de que quando `prototype` aparece em fonte de código, ele se refere a uma propriedade de objeto comum, não ao atributo `prototype`: O Capítulo 9 explicou que o protótipo

de uma função de construtor especifica o atributo prototypedos objetos criados com esse construtor.

O atributo prototype é definido quando um objeto é criado.

Objetos criados a partir de literais de objeto usam Object.prototype como seu protótipo. Objetos criados com new usam o valor da propriedade prototype de sua função de construtor como seu protótipo. E os objetos criados com Object.create() usam o primeiro argumento para thatfunction (que pode ser nulo) como seu protótipo.

Você pode consultar o protótipo de qualquer objeto passando esse objeto para Object.getPrototypeOf():

```
Object.getPrototypeOf({})      => Object.prototype  
Object.getPrototypeOf([])     => Matriz.prototype  
Object.getPrototypeOf(()=>{}) => Função.prototype
```

Uma função muito semelhante, Reflect.getPrototypeOf(), é descrita em §14.6.

Para determinar se um objeto é o protótipo de (ou faz parte da cadeia de protótipos) de outro objeto, use o método isPrototypeOf():

```
seja p = {x: 1};           Definir um protótipo  
object.let o =  
Object.create(p);          Crie um objeto com  
que  
prototype.p.isPrototypeOf(o) => true: o herda de  
p0bject.prototype.isPrototypeOf(p) // => true: p herda  
fromObject.prototype
```

```
Object.prototype.isPrototypeOf(o) // => true: o também
```

Observe que `isPrototypeOf()` executa uma função semelhante ao operador `instanceof` (consulte §4.9.4).

O atributo `prototype` de um objeto é definido quando o objeto é criado e normalmente permanece fixo. Você pode, no entanto, alterar o protótipo de um objeto com `Object.setPrototypeOf()`:

```
seja o = {x: 1}; seja p = {y: 2}; Object.setPrototypeOf(o, p); Defina o protótipo de o para po.y// => 2: o agora herda a propriedade ylet a = [1, 2, 3]; Object.setPrototypeOf(a, p); Defina o protótipo do array aто pa.join// => undefined: a não tem mais um método join()
```

Geralmente, não há necessidade de usar `Object.setPrototypeOf()`. As implementações de JavaScript podem fazer otimizações agressivas com base na suposição de que o protótipo de um objeto é fixo e imutável. Isso significa que, se você chamar `Object.setPrototypeOf()`, qualquer código que use os objetos alterados poderá ser executado muito mais lentamente do que normalmente.

Uma função semelhante, `Reflect.setPrototypeOf()`, é descrita no §14.6.

Algumas implementações iniciais de JavaScript no navegador expuseram o atributo `prototype` de um objeto por meio da propriedade `__proto__` (escrita com dois sublinhados no início e no final). Isso já faz muito tempo

foi obsoleto, mas o código existente suficiente na web depende `on__proto__` que o padrão ECMAScript o exija para todas as implementações JavaScript executadas em navegadores da web. (O Node também o suporta, embora o padrão não o exija para o Node.) Em modernJavaScript, `__proto__` é legível e gravável, e você pode (embora não deva) usá-lo como uma alternativa para `Object.getPrototypeOf()` e `Object.setPrototypeOf()`. Um uso interessante de `__proto__`, no entanto, é definir o protótipo de um objeto literal:

```
seja p = {z:  
3}; seja o = {  
x: 1,y: 2,__proto__: p}; o.z//  
=> 3: o herda de p
```

14.4 Símbolos conhecidos

O tipo Symbol foi adicionado ao JavaScript no ES6, e uma das principais razões para isso foi adicionar extensões com segurança à linguagem sem quebrar a compatibilidade com o código já implantado na web. Vimos um exemplo disso no Capítulo 12, onde aprendemos que você pode tornar uma classe iterável implementando um método cujo "nome" é o `Symbol.iterator`.

`Symbol.iterator` é o exemplo mais conhecido dos "well-knownSymbols". Estes são um conjunto de valores de símbolo armazenados como propriedades da função de fábrica `theSymbol()` que são usados para permitir que o código JavaScript

controlar determinados comportamentos de baixo nível de objetos e classes. As subseções a seguir descrevem cada um desses símbolos conhecidos e explicam como eles podem ser usados.

14.4.1 Symbol.iterator e Symbol.asyncIterator

Os símbolos `Symbol.iterator` e `Symbol.asyncIterator` permitem que objetos ou classes se tornem iteráveis ou assíncronas. Eles foram abordados em detalhes no Capítulo 12 e §13.4.2, respectivamente, e são mencionados novamente aqui apenas para completar.

14.4.2 Symbol.hasInstance

Quando o operador `instanceof` foi descrito em §4.9.4, dissemos que o lado direito deve ser uma função construtora e que a expressão `o instanceof f` foi avaliada procurando o `valuef.prototype` dentro da cadeia `prototype` de `o`. Isso ainda é verdade, mas no ES6 e além, `Symbol.hasInstance` fornece uma alternativa. No ES6, se o lado direito da instância de `for` qualificar qualquer objeto com um método `[Symbol.hasInstance]`, esse método será invocado com o valor do lado esquerdo como seu argumento, e o valor de retorno do método, convertido em um booleano, torna-se o valor do operador `instanceof`. E, é claro, se o valor no lado direito não tiver um método `[Symbol.hasInstance]`, mas for uma função, o operador `instanceof` se comportará de maneira comum.

`Symbol.hasInstance` significa que podemos usar o `instanceof` operator para fazer verificação de tipo genérico com pseudotypeobjects adequadamente definidos. Por exemplo:

```
Defina um objeto como um "tipo" que podemos usar com  
instanceoflet uint8 = {  
    [Symbol.hasInstance](x) {  
        return Number.isInteger(x) && x >= 0 && x <= 255;}}; 128  
instanceof uint8// => true256 instanceof uint8// =>  
false: muito bigMath.PI instanceof uint8 // => false:  
não é um inteiro
```

Observe que este exemplo é inteligente, mas confuso porque usa um objeto nonclass onde uma classe normalmente seria esperada. Seria igualmente fácil - e mais claro para os leitores do seu código - escrever a função `isUInt8()` em vez de confiar no comportamento `thisSymbol.hasInstance`.

14.4.3 `Symbol.toStringTag`

Se você invocar o método `toString()` de um objeto JavaScript básico, obterá a string "[object Object]":

```
{}.toString() => "[object Object]"
```

Se você invocar essa mesma função `Object.prototype.toString()` como um método de instâncias de tipos internos, obterá alguns resultados interessantes:

```
Object.prototype.toString.call([])      => "[ matriz de objetos ]"  
Object.prototype.toString.call(/./)     => "[objeto  
RegExp]"Object.prototype.toString.call(()=>{}) // =>  
"  
[objectFunction]"Object.prototype.toString.call("")/  
/ => "  
[objectString]"Object.prototype.toString.call(0)//  
=> "[object
```

```
Número]"Object.prototype.toString.ca  
ll(false)                                => "[objeto  
Booleano]"
```

Acontece que você pode usar essa técnica de `Object.prototype.toString().call()` com qualquer valor JavaScript para obter o "atributo de classe" de um objeto que contém informações de tipo que não estão disponíveis de outra forma. A função `followingclassof()` é indiscutivelmente mais útil do que o `typeofoperator`, que não faz distinção entre os tipos de objetos:

```
function classof(o) {  
    return Object.prototype.toString.call(o).slice(8,-1);}  
  
classof(null)      => "Nulo"  
classof(undefined) => "Indefinido"  
classede(1)        => "Número"  
classe de (10n ** 100n) => "BigInt"  
classof(false)// =>  
"Boolean" classof(Symbol())// =>  
"Symbol" classof({})// =>  
"Object" classof([])// =>  
"Array" classof(/.//) // =>  
"RegExp" classof(()=>{})// =>  
"Function" classof(new Map())// =>  
"Map" classof(new Set())// =>  
"Set" classof(new Date()) // =>  
"Data"
```

Antes do ES6, esse comportamento especial do método `Object.prototype.toString()` estava disponível apenas para instâncias de tipos embutidos e, se você chamassem essa função `classof()` em uma instância de uma classe que você mesmo definiu, ela simplesmente retornaria "Object". No ES6, no entanto,

`Object.prototype.toString()` procura uma propriedade com o nome simbólico `Symbol.toStringTag` em seu argumento e, se essa propriedade existir, ele usa o valor da propriedade em sua saída. Isso significa que, se você definir uma classe própria, poderá facilmente fazê-la funcionar com funções como `classof()`:

```
class Intervalo {  
  get [Symbol.toStringTag]() { return "Intervalo"; }  
  // o resto  
  // desta classe é omitido aqui}  
let r = new Range(1, 10);  
Object.prototype.toString.call(r)// => "[Intervalo do  
// objeto]"  
classof(r)// => "Intervalo"
```

14.4.4 Símbolo.especie

Antes do ES6, o JavaScript não fornecia nenhuma maneira real de criar subclasses robustas de classes integradas como `Array`. No ES6, no entanto, você pode estender qualquer classe interna simplesmente usando a classe `extends` keyword. §9.5.2 demonstrou que, com esta subclasse simples de `Array`:

Uma subclasse `Array` trivial que adiciona getters para o primeiro e o último elementos.

```
class EZArray extends Array {  
  get first() { return this[0]; }  
  get last() {  
    return this[this.length-1]; }}
```

```
let e = new EZArray(1,2,3); let f = e.map(x => x * x);  
e.last// => 3: o último elemento do EZArray ef.last// =>  
9: f também é um EZArray com uma propriedade last
```

`Array` define os métodos `concat()`, `filter()`, `map()`, `slice()`,

e splice(), que retornam matrizes. Quando criamos uma subclasse de array como EZArray que herda esses métodos, o inheritedmethod deve retornar instâncias de Array ou instâncias de EZArray? Bons argumentos podem ser feitos para qualquer escolha, mas a especificação ES6 diz que (por padrão) os cinco métodos de retorno de array retornarão instâncias da subclasse.

Veja como funciona:

No ES6 e posterior, o construtor Array() tem uma propriedade com o nome simbólico Symbol.species. (Observe que thisSymbol é usado como o nome de uma propriedade da função construtor. A maioria dos outros símbolos conhecidos descritos aqui são usados como o nome de métodos de um objeto protótipo.) Quando criamos uma subclasse com extends, o construtor resultingsubclass herda propriedades do construtor superclass. (Isso é um acréscimo ao tipo normal de herança, onde as instâncias da subclasse herdam métodos da superclass.) Isso significa que o construtor para cada subclasse de Array também tem uma propriedade herdada com nameSymbol.species. (Ou uma subclasse pode definir sua própria propriedade com esse nome, se desejar.) Métodos como map() e slice() que criam e retornam novas matrizes são ligeiramente ajustados no ES6 e posterior. Em vez de apenas criar um Array regular, eles (na verdade) invocam newthis.constructor[Symbol.species]() para criar o novo array. Agora, aqui está a parte interessante. Suponha que Array[Symbol.species] fosse apenas uma propriedade de dados regular, definida assim:

```
Matriz[Símbolo.espécies] = Matriz;
```

Nesse caso, os construtores de subclasse herdariam o construtor `Array()` como sua "espécie" e invocar `map()` em uma subclasse de matriz retornaria uma instância da superclasse em vez de uma instância da subclasse. Não é assim que o ES6 realmente se comporta, no entanto. O motivo é que `Array[Symbol.species]` é uma propriedade acessadora somente leitura cuja função getter simplesmente retorna `this`. Os construtores de subclasse herdam essa função getter, o que significa que, por padrão, cada construtor de subclasse é sua própria "espécie".

No entanto, às vezes, esse comportamento padrão não é o que você deseja. Se você quiser que os métodos de retorno de array do `EZArray` retornem objetos Array regulares, você só precisa definir `EZArray[Symbol.species]` como `Array`. Mas como a propriedade herdada é um acessador somente leitura, você não pode simplesmente defini-la com um operador de atribuição. Você pode usar `defineProperty()`, no entanto:

```
EZArray[Symbol.species] = Matriz; Falha na tentativa de  
definir uma propriedade somente leitura
```

```
Em vez disso, podemos usar  
defineProperty():Object.defineProperty(EZArray,  
Symbol.species, {value:Array});
```

A opção mais simples é provavelmente definir explicitamente seu getter `ownSymbol.species` ao criar a subclasse em primeiro lugar:

```
class EZArray extends Array {  
    static get [Symbol.species]() { return Array;  
    }get first() { return this[0]; }
```

```
get last() { return this[this.length-1]; }
```

```
let e = new EZArray(1,2,3); seja f = e.map(x => x - 1);
e.last// => 3f.last// => undefined: f é um array regular
sem lastgetter
```

A criação de subclasses úteis de Array foi o principal caso de uso que motivou a introdução de Symbol.species, mas não é o único lugar em que este conhecido Symbol é usado. As classes de matriz tipadas usam o Symbol da mesma forma que a classe Array. Da mesma forma, o método slice() de ArrayBuffer olha para a propriedade Symbol.species de this.constructor em vez de simplesmente criar um novo ArrayBuffer. E os métodos Promise likethen() que retornam novos objetos Promise também criam esses objetos por meio do protocolo thisspecies. Finalmente, se você estiver criando uma subclasse de Map(por exemplo) e definindo métodos que retornam novos objetos Map, talvez queira usar Symbol.species para o benefício de subclasses de sua subclasse.

14.4.5 **Symbol.isConcatSpreadable**

O método Array concat() é um dos métodos descritos na seção anterior que usa Symbol.species para determinar qual construtor usar para o array retornado. Mas concat() também usaSymbol.isConcatSpreadable. Lembre-se de §7.8.3 que o método concat() de uma matriz trata seu valor this e sua matriz argumentos de maneira diferente de seus argumentos não matriz: os argumentos não matriz são simplesmente anexados à nova matriz, mas a matriz this e qualquer

Os argumentos do array são nivelados ou "espalhados" para que os elementos do array sejam concatenados em vez do próprio argumento do array.

Antes do ES6, concat() usava apenas Array.isArray() para determinar se um valor deve ser tratado como um array ou não. No ES6, o algoritmo é ligeiramente alterado: se o argumento (ou o valor this) para concat() for um objeto e tiver uma propriedade com o nome simbólico Symbol.isConcatSpreadable, então o valor booleano dessa propriedade é usado para determinar se o argumento deve ser "espalhado". Se essa propriedade não existir, Array.isArray() será usado como nas versões anteriores da linguagem.

Há dois casos em que você pode querer usar este símbolo:

- Se você criar um objeto semelhante a um Array (veja §7.9) e quiser que ele se comporte como um array real quando passado para concat(), você pode simplesmente adicionar a propriedade simbólica ao seu objeto:

```
let arraylike = {
  comprimento: 1, 0: 1,
  [Symbol.isConcatSpreadable]: true};
[].concat(arraylike)// => [1]: (seria
[[1]] se não se espalhasse)
```

- As subclasses de array podem ser espalhadas por padrão, portanto, se você estiver definindo uma subclasse de array que não deseja que aja como um array quando usado com concat(), poderá adicionar um getter como este à sua subclasse:¹

```
class NonSpreadableArray extends Array {  
    get [Symbol.isConcatSpreadable](){  
        return false;  
    }  
    let a = new  
    NonSpreadableArray(1,2,3);  
    [].concat(a).length // => 1; (teria 3  
    elementos de comprimento se um foi  
    espalhado)
```

14.4.6 Símbolos de correspondência de padrões

§11.3.2 documentou os métodos String que executam operações de correspondência de padrões usando um argumento RegExp. No ES6 e posterior, esses métodos foram generalizados para trabalhar com objetos RegExp ou qualquer objeto que defina o comportamento de correspondência de padrões por meio de propriedades com nomes simbólicos. Para cada um dos métodos de string match(), matchAll(), search(), replace() e split(), há um well-knownSymbol correspondente: Symbol.match, Symbol.search e assim por diante.

RegExps são uma maneira geral e muito poderosa de descrever padrões textuais, mas podem ser complicados e não adequados para correspondência difusa. Com os métodos de string generalizados, você pode definir suas próprias classes de padrão usando os conhecidos métodos Symbol para fornecer correspondência personalizada. Por exemplo, você pode executar comparações de cadeia de caracteres usando Intl.Collator (consulte §11.7.3) para ignorar acentos ao corresponder. Ou você pode definir uma classe de padrão com base no algoritmo Soundex para corresponder palavras com base em seus sons aproximados ou para corresponder vagamente strings até uma determinada distância de Levenshtein.

Em geral, quando você invoca um desses cinco métodos String em um

padrão como este:

```
string.method(padão, arg)
```

Essa invocação se transforma em uma invocação de um NamedMethod simbolicamente em seu objeto Pattern:

```
padão [símbolo] (string, arg)
```

Como exemplo, considere a classe de correspondência de padrões no próximo exemplo, que implementa a correspondência de padrões usando os caracteres curinga * e ?simples com os quais você provavelmente está familiarizado nos sistemas de arquivos. Esse estilo de correspondência de padrões remonta aos primeiros dias do sistema operacional Unix, e os padrões são freqüentemente chamados de globs:

```
class Glob {  
    construtor(glob) {  
        this.glob = glob;  
  
        Implementamos a correspondência glob usando RegExp internamente.  
        // ? corresponde a qualquer caractere, exceto /, e *  
        // corresponde a zero ou mais  
        // desses personagens. Usamos grupos de captura  
        // em torno de cada um.  
        let regexpText = glob.replace(/\?/, "  
([/])").replace(/\*/g, "([/]*");
```

Usamos o sinalizador g para obter correspondência com reconhecimento de Unicode.// Globs destinam-se a corresponder strings inteiras, então nós usamos ^ e \$
âncoras e não implementam search() ou
matchAll() já que eles
não são úteis com padrões como este.
`this.regexp = new RegExp(`^${regexpText}$`, "u");}`

```
toString() { return this.glob; }
```

```

[Symbol.search](s) { return s.search(this.regexp); }
[Símbolo.correspondência](s){ return s.match(this.regexp);
 }[Symbol.replace](s, substituição) {
return s.replace(this.regexp, replacement);}
}

let padrão = new Glob("docs/*.txt");
"docs/js.txt".search(pattern)// => 0: corresponde a
character0"docs/js.htm".search(pattern)// => -1: não
corresponde a match = "docs/js.txt".match(pattern);
match[0]// => "docs/js.txt"match[1]// => "js"match.index// 
=> 0"docs/js.txt".replace(pattern, "web/$1.htm")//
=>"web/js.htm"

```

14.4.7 Symbol.toPrimitive

§3.9.3 explicou que o JavaScript tem três algoritmos ligeiramente diferentes para converter objetos em valores primitivos. Falando livremente, para conversões em que um valor de string é esperado ou preferido, o JavaScript invoca o método `toString()` de um objeto primeiro e recorre ao método `valueOf()` se `toString()` não estiver definido ou não retornar um valor primitivo. Para conversões em que um valor numérico é preferido, o JavaScript tenta o método `valueOf()` primeiro e recorre a `String()` se `valueOf()` não estiver definido ou se não retornar um valor primitivo. E, finalmente, nos casos em que não há preferência, permite que a classe decida como fazer a conversão. Objetos de data convertusing `toString()` primeiro, e todos os outros tipos tentam `valueOf()` primeiro.

No ES6, o conhecido `Symbol.toPrimitive` permite que você

para substituir esse comportamento padrão de objeto para primitivo e dá a você controle total sobre como as instâncias de suas próprias classes serão convertidas em valores primitivos. Para fazer isso, defina um método com este nome simbólico. O método deve retornar um valor primitivo que de alguma forma representa o objeto. O método definido será invocado com um único argumento de cadeia de caracteres que informa que tipo de conversão JavaScript está tentando fazer em seu objeto:

Se o argumento for "string", significa que o JavaScript está fazendo a conversão em um contexto em que esperaria ou preferiria (mas não exigiria) uma string. Isso acontece quando você interpola o objeto em um literal de modelo, por exemplo. Se o argumento for "número", significa que o JavaScript está fazendo a conversão em um contexto em que esperaria ou preferiria (mas não exigiria) um valor numérico. Isso acontece quando você usa o objeto com um operador < ou > ou com operadores aritméticos como - e *. Se o argumento for "padrão", significa que o JavaScript está convertendo seu objeto em um contexto em que um valor numérico ou de string pode funcionar. Isso acontece com os operadores +, == e !=. Muitas classes podem ignorar o argumento e simplesmente retornar o mesmo valor primitivo em todos os casos. Se você deseja que as instâncias de sua classe sejam comparáveis e classificáveis com < e >, esse é um bom motivo para definir um método [Symbol.toPrimitive].

14.4.8 Símbolo.unscopables

O último símbolo conhecido que abordaremos aqui é um obscuro que foi introduzido como uma solução alternativa para problemas de compatibilidade causados por

a instrução `with` obsoleta. Lembre-se de que a instrução `with` pega um objeto e executa o corpo da instrução como se estivesse em um escopo em que as propriedades desse objeto fossem variáveis. Isso causou problemas de compatibilidade quando novos métodos foram adicionados à classe `Array` e quebrou algum código existente. `Symbol.unscopables` é o resultado. InES6 e posterior, a instrução `with` foi ligeiramente modificada. Quando usado com um objeto `o`, uma instrução `with` computa `Object.keys(o[Symbol.unscopables]||{})` e ignora propriedades cujos nomes estão na matriz resultante ao criar o escopo simulado no qual executar seu corpo. O ES6 usa isso para adicionar newmethods ao `Array.prototype` sem quebrar o código existente na web. Isso significa que você pode encontrar uma lista dos métodos `Array` mais recentes avaliando:

```
let newArrayMethods  
=Object.keys(Array.prototype[Symbol.unscopables]);
```

14.5 Tags de modelo

As strings dentro de acentos graves são conhecidas como "literais de modelo" e recuperamos [em §3.3.4](#). Quando uma expressão cujo valor é uma função é seguida por um literal de modelo, ela se transforma em uma invocação de função e a chamamos de "literal de modelo marcado". Definir uma nova função de tag para uso com literais de modelo marcados pode ser pensado como metaprogramação, porque os modelos marcados são frequentemente usados para definir DSLs - linguagens específicas de domínio - e definir uma nova função de tag é como adicionar uma nova sintaxe ao JavaScript. Literais de modelo marcados foram adotados por vários pacotes JavaScript de front-end. A linguagem de consulta TheGraphQL usa uma função de tag `gql`" para permitir consultas

a ser incorporado ao código JavaScript. E a biblioteca Emotion usa a função de tag acss" para permitir que os estilos CSS sejam incorporados ao JavaScript. Esta seção demonstra como escrever suas próprias tagfunções como essas.

Não há nada de especial nas funções de tag: elas são funções JavaScript comuns e nenhuma sintaxe especial é necessária para defini-las. Quando uma expressão de função é seguida por um literal de modelo, a função é invocada. O primeiro argumento é uma matriz de strings, e isso é seguido por zero ou mais argumentos adicionais, que podem ter valores de qualquer tipo.

O número de argumentos depende do número de valores que são interpolados no literal de modelo. Se o literal de modelo for simplesmente uma string constante sem interpolações, a função de tag será chamada com uma matriz dessa string e sem argumentos adicionais. Se o literal de modelo incluir um valor interpolado, a tagfunction será chamada com dois argumentos. O primeiro é uma matriz de duas strings e o segundo é o valor interpolado. As strings nessa matriz inicial são a string à esquerda do valor interpolado e a string à sua direita, e qualquer uma delas pode ser a string vazia. Se o literal de modelo incluir dois valores interpolados, a função de tag será invocada com três argumentos: uma matriz de três cadeias de caracteres e os dois valores interpolados. As três cadeias de caracteres (qualquer uma ou todas as quais podem estar vazias) são o texto à esquerda do primeiro valor, o texto entre os dois valores e o texto à direita do segundo valor. No caso geral, se o literal de modelo tiver n valores interpolados, a função de tag será invocada com $n + 1$ argumentos. O primeiro argumento será um


```
e</b>"  
  
let kind = "jogo", nome = "D&D";html'<div  
class="${kind}">${nome}</div>' //  
=>'<divclass="jogo">D&D</div>'
```

Para obter um exemplo de uma função de marca que não retorna uma cadeia de caracteres, mas uma representação analisada de uma cadeia de caracteres, pense na classe `Globpattern` definida em §14.4.6. Como o construtor `Glob()` recebe um único argumento de string, podemos definir uma função de tag para criar objetos `newGlob`:

```
function glob(strings, ... valores) {  
    Monte as strings e os valores em uma única stringlet s  
    = strings[0]; for(let i = 0; i < valores.comprimento;  
    i++) {  
        s += valores[i] + strings[i+1];}Retorna uma  
    representação analisada dessa stringretorna novo(s)  
    Glob(s);}  
  
let root = "/tmp"; let padrão_de_arquivo = glob`${raiz}/*.html';  
A  
RegExpalternative"/tmp/test.html".match(filePattern)  
[1]// => "test"
```

Um dos recursos mencionados na passagem em §3.3.4 é a função de tag `theString.raw` que retorna uma string em sua forma "bruta" sem interpretar nenhuma das sequências de escape de barra invertida. Isso é implementado usando um recurso de invocação de função de tag que ainda não discutimos. Quando uma função de tag é invocada, vimos que seu primeiro argumento é uma matriz de strings. Mas essa matriz também tem uma propriedade chamada `raw`, e o valor dessa propriedade é outra matriz de strings, com o mesmo número de elementos. A matriz de argumentos inclui cadeias de caracteres

que tiveram sequências de escape interpretadas como de costume. E a matriz bruta inclui strings nas quais as sequências de escape não são interpretadas. Esse recurso obscuro é importante se você deseja definir uma DSL com uma gramática que usa barras invertidas. Por exemplo, se quiséssemos que a função de tag ourglob " suportasse a correspondência de padrões em caminhos de estilo Windows (que usam barras invertidas em vez de barras) e não quiséssemos que os usuários da tag tivessem que dobrar cada barra invertida, poderíamos reescrever essa função para usar strings.raw[] em vez de strings []. A desvantagem, é claro, seria que não poderíamos mais usar escapes como \u em nossos literais globais.

14.6 A API Reflect

O objeto Reflect não é uma classe; como o objeto Math, suas propriedades simplesmente definem uma coleção de funções relacionadas. Essas funções, adicionadas no ES6, definem uma API para "refletir sobre" objetos e suas propriedades. Há pouca funcionalidade nova aqui: o objeto Reflect define um conjunto conveniente de funções, todas em um único namespace, que imitam o comportamento da sintaxe da linguagem principal e duplicam os recursos de várias funções Object pré-existentes.

Embora as funções Reflect não forneçam novos recursos, elas agrupam os recursos em uma API conveniente. E, mais importante, o conjunto de funções Reflect mapeia um a um com o conjunto de métodos de manipulador de proxy sobre os quais aprenderemos em §14.7.

A API Reflect consiste nas seguintes funções:

Reflect.apply(f, o, args)

Esta função invoca a função f como um método de o (ou a invoca como uma função sem este valor se o for nulo) e passa os valores na matriz args como argumentos. É equivalente a f.apply(o, args).

Reflect.construct(c, args, newTarget)

Essa função invoca o construtor c como se a palavra-chave new tivesse sido usada e passa os elementos dos argumentos da matriz como argumentos. Se o argumento opcional newTarget for especificado, ele será usado como o valor de new.target na chamada do construtor. Se não for especificado, o valor new.target será c.

Reflect.defineProperty(o, nome, descriptor)

Esta função define uma propriedade no objeto o, usando name (astring ou symbol) como o nome da propriedade. O Descriptor deve definir o valor (ou getter e/ou setter) e os atributos da propriedade. Reflect.defineProperty() é muito semelhante a Object.defineProperty(), mas retorna true em caso de sucesso e false em caso de falhas. (Object.defineProperty() retorna false em caso de sucesso e gera TypeError em caso de falha.)

Reflect.deleteProperty(o, nome)

Essa função exclui a propriedade com a cadeia de caracteres especificada ou o nome simbólico do objeto o, retornando true se for bem-sucedido (ou se essa propriedade não existir) e false se a propriedade não puder ser excluída. Chamar essa função é semelhante a escrever deleteo[name].

Reflect.get(o, nome, receptor)

Esta função retorna o valor da propriedade de o com o nome especificado (uma string ou símbolo). Se a propriedade for um acessador

com um getter, e se o argumento receiver opcional for especificado, a função getter será chamada como um método ofreceivever em vez de como um método de o. Chamar essa função é semelhante a avaliar o[name].

Reflect.getOwnPropertyDescriptor(o, nome)

Essa função retorna um objeto descritor de propriedade que descreve os atributos da propriedade chamada nome do objeto o, ou returnsundefined se tal propriedade não existir. Essa função é quase idêntica a Object.getOwnPropertyDescriptor(), exceto que a versão da API Reflect da função requer que o primeiro argumento seja um objeto e gere TypeError se não for.

Reflect.getPrototypeOf(o)

Esta função retorna o protótipo do objeto o ou null se o objeto não tiver protótipo. Ele lançará um TypeError se o for um primitivevalue em vez de um objeto. Esta função é quase idêntica a Object.getPrototypeOf(), exceto que Object.getPrototypeOf() apenas lança um TypeError para argumentos nulos e indefinidos e coage outros valores primitivos para seus objetos wrapper.

Reflect.has(o, nome)

Essa função retornará true se o objeto o tiver uma propriedade com o nome especificado (que deve ser uma string ou um símbolo). Chamar essa função é semelhante a avaliar o nome em o.

Reflect.isExtensible(o)

Essa função retornará true se o objeto o for extensível (§14.2) e false se não for. Ele lança um TypeError se o não for um objeto. Object.isExtensible() é semelhante, mas simplesmente retorna false quando passado um argumento que não é um objeto.

Reflect.ownKeys(o)

Essa função retorna uma matriz dos nomes das propriedades do objeto o ou lança um TypeError se o não for um objeto. Os nomes na matriz retornada serão cadeias de caracteres e/ou símbolos. Chamar thisfunction é semelhante a callingObject.getOwnPropertyNames() eObject.getOwnPropertySymbols() e combinar seus resultados.

Reflect.preventExtensions(o)

Essa função define o atributo extensível (§14.2) do objeto o tofalse e retorna true para indicar êxito. Ele lança aTypeError se o não for um objeto. Object.preventExtensions() tem o mesmo efeito, mas retorna o em vez de true e não lança TypeError para argumentos que não são objetos.

Reflect.set(o, nome, valor, receptor)

Essa função define a propriedade com o nome especificado do objeto o para o valor especificado. Ele retorna true em caso de êxito e false em caso de falha (o que pode acontecer se a propriedade for somente leitura). Ele lança TypeError se o não for um objeto. Se a propriedade especificada for uma propriedade acessadora com uma função setter e se o argumento optionalreceiver for passado, o setter será invocado como um método de receiver em vez de ser invocado como um método de o. Chamar essa função geralmente é o mesmo que avaliar o[name] =value.

Reflect.setPrototypeOf(o, p)

Esta função define o protótipo do objeto o como p, retornando true em caso de sucesso e false em caso de falha (o que pode ocorrer se o não for extensível ou se a operação causar um protótipo circular

cadeia). Ele lança um TypeError se o não for um objeto ou se p não for um objeto nem nulo. Object.setPrototypeOf() é semelhante, mas retorna o em caso de sucesso e gera TypeError em caso de falha. Lembre-se de que chamar qualquer uma dessas funções provavelmente tornará seu código mais lento, interrompendo as otimizações do interpretador JavaScript.

14.7 Objetos proxy

A classe Proxy, disponível no ES6 e posterior, é o recurso de metaprogramação mais poderoso do JavaScript. Ele nos permite escrever código que altera o comportamento fundamental dos objetos JavaScript. A API Reflect descrita em §14.6 é um conjunto de funções que nos dá acesso direto a umconjunto de operações fundamentais em objetos JavaScript. O que a classe Proxy faz é nos permitir uma maneira de implementar essas operações fundamentais por conta própria e criar objetos que se comportam de maneiras que não são possíveis para objetos comuns.

Quando criamos um objeto Proxy, especificamos dois outros objetos, o objeto alvo e o objeto manipulador:

```
let proxy = new Proxy(destino, manipuladores);
```

O objeto Proxy resultante não tem estado ou comportamento próprio. Sempre que você executa uma operação nele (ler uma propriedade, escrever uma propriedade, definir uma nova propriedade, procurar o protótipo, invocá-lo como uma função), ele despacha essas operações para o objeto handlers ou para o objeto target.

As operações suportadas por objetos Proxy são as mesmas que aquelas

definido pela API Reflect. Suponha que p seja um objeto Proxy e você escreva delete p.x. A função Reflect.deleteProperty() tem o mesmo comportamento que o operador delete. E quando você usa o operador delete para excluir uma propriedade de um objeto Proxy, ele procura o método deleteProperty() no objeto handlers. Se tal método existir, ele o invocará. E se esse método não existir, então o Objeto proxy executa a exclusão de propriedade no objeto de destino em vez disso.

Os proxies funcionam dessa maneira para todas as operações fundamentais: se existir um método apropriado no objeto handlers, ele invocará esse método para executar a operação. (Os nomes e assinaturas dos métodos são os mesmos das funções Reflect abordadas no §14.6.) E se esse método não existir no objeto handlers, o Proxy executará a operação fundamental no objeto de destino. Isso significa que um Proxy pode obter seu comportamento do objeto de destino ou do objeto handlers. Se o objeto handlers estiver vazio, o proxy será essencialmente um wrapper transparente em torno do objeto de destino:

```
let t = { x: 1, y: 2 }; let p = new Proxy(t, {}); p.x// =>
 1delete p.y// => true: exclui a propriedade y do proxyt.y// => undefined: isso a exclui no destino, toop.z = 3// Definindo uma nova propriedade no proxyt.z// => 3: define a propriedade no destino
```

Esse tipo de proxy wrapper transparente é essencialmente equivalente ao objeto de destino subjacente, o que significa que realmente não há uma razão para usá-lo em vez do objeto encapsulado. Os invólucros transparentes podem ser

útil, no entanto, quando criado como "proxies revogáveis". Em vez de criar um Proxy com o construtor Proxy(), você pode usar a função de fábrica Proxy.revocable(). Essa função retorna um objeto que inclui um objeto Proxy e também uma função revoke(). Depois de chamar a função revoke(), o proxy para de funcionar imediatamente:

```
function accessTheDatabase() { /* implementação omitida */  
  return 42; }let {proxy, revoke} =  
Proxy.revocable(accessTheDatabase, {});  
  
proxy() => 42: O proxy dá acesso ao  
função de destino revogar(); Mas esse acesso pode ser  
desativado sempre que quisermos proxy()// ! TypeError: não  
podemos mais chamar esta função
```

Observe que, além de demonstrar proxies revogáveis, o código anterior também demonstra que os proxies podem funcionar com funções de destino, bem como objetos de destino. Mas o ponto principal aqui é que proxies revogáveis são um bloco de construção para um tipo de isolamento de código, e você pode usá-los ao lidar com bibliotecas de terceiros não confiáveis, por exemplo. Se você tiver que passar uma função para uma biblioteca que não controla, poderá passar um proxy revogável e revogar o proxy quando terminar a biblioteca. Isso impede que a biblioteca mantenha uma referência à sua função e a chame em momentos inesperados. Esse tipo de programação defensiva não é típico em programas JavaScript, mas a classe Proxy pelo menos torna isso possível.

Se passarmos um objeto manipulador não vazio para o construtor Proxy(), não estaremos mais definindo um objeto wrapper transparente e estaremos

em vez disso, implementando um comportamento personalizado para nosso proxy. Com o conjunto certo de manipuladores, o objeto de destino subjacente torna-se essencialmente irrelevante.

No código a seguir, por exemplo, é como poderíamos implementar um objeto que parece ter um número infinito de propriedades somente leitura, onde o valor de cada propriedade é o mesmo que o nome da propriedade:

Usamos um Proxy para criar um objeto que parece ter todas// propriedades possíveis, com o valor de cada propriedade igual ao seu nomelet identity = new Proxy({}, {

Cada propriedade tem seu próprio nome como seu valueget(o, name, target) { return name; },// Cada nome de propriedade é definedhas(o, name) { return true; },// Há muitas propriedades para enumerar, então nós apenas jogamos

```
ownKeys(o) { throw new RangeError("Número infinito de propriedades"); },
```

Todas as propriedades existem e não são graváveis, configuráveis ou enumeráveis.

```
    getOwnPropertyDescriptor(o, nome) {
```

```
        retornar {
```

```
            value: nome,enumerable: false,writable: false,configurable: false};},// Todas as propriedades são somente leitura, portanto, não podem ser setset(o, name, value, target) { return false; },// Todas as propriedades não são configuráveis, portanto, não podem ser excluídas
```

```
deleteProperty(o, name) { return false; },// Todas as propriedades existem e não são configuráveis, portanto, não podemos definir mais
```

```
    defineProperty(o, name, desc) { return false; },
```

```

Com efeito, isso significa que o objeto não é extensível
isExtensible(o) { return false; },// Todas as propriedades já estão definidas neste objeto, portanto, não foi possível
inherit anything, mesmo que tenha um prototypeobject.

getPrototypeOf(o) { return null; },// O objeto não é extensível, então não podemos alterar o protótipo

setPrototypeOf(o, proto) { return false; },});

identidade.x          => "x"
identidade.comString  => "toString"
identidade[0]          => "0"
identidade.x = 1;      Definir propriedades não tem efeito
identidade.x          => "x"
excluir identidade.x  => false: não é possível excluir
Propriedades
EitherIdentity.x       => "x"
Object.keys(identidade); // ! RangeError: não é possível listar todos os
keysfor(let p de
identidade) ;           // ! Erro de intervalo

```

Os objetos proxy podem derivar seu comportamento do objeto de destino e do objeto handlers, e os exemplos que vimos até agora usaram um objeto ou outro. Mas normalmente é mais útil definir proxiesque usam ambos os objetos.

O código a seguir, por exemplo, usa Proxy para criar um wrapper somente leitura para um objeto de destino. Quando o código tenta ler valores do objeto, essas leituras são encaminhadas para o objeto de destino normalmente. Mas se qualquer código tentar modificar o objeto ou suas propriedades, os métodos do objeto manipulador lançam um TypeError. Um proxy como este pode ser útil para escrever testes: suponha que você tenha escrito uma função que recebe um argumento de objeto e deseja garantir que sua função não faça nenhum

tentativa de modificar o argumento de entrada. Se o teste for aprovado em um objeto wrapper somente leitura, todas as gravações lançarão exceções que farão com que o teste falhe:

```
function readOnlyProxy(o) {  
    function somente_leitura() { throw new  
        TypeError("Somente leitura"); } return new Proxy(o, {  
    set: readonly,defineProperty:  
    readonly,deleteProperty:  
    readonly,setPrototypeOf:  
    readonly,});}
```

Objeto que não pode ser gravado
*p = readOnlyProxy(o); // Versão somente leitura do itp.x// => 1: propriedades de leitura
worksp.x = 2;// ! TypeError: não é possível
alterarpropriedadesexcluir p.y;// ! TypeError: não é
possível excluirpropriedadesp.z = 3;// ! TypeError: não é
possível addpropertiesp.__proto__ = {};// ! TypeError: não
é possível alterar o protótipo*

Outra técnica ao escrever proxies é definir métodos de manipulador que interceptam operações em um objeto, mas ainda delegam as operações ao objeto de destino. As funções da API Reflect ([§14.6](#)) têm exatamente as mesmas assinaturas que os métodos do manipulador, portanto, facilitam esse tipo de delegação.

Aqui, por exemplo, está um proxy que delega todas as operações ao targetobject, mas usa métodos de manipulador para registrar as operações:

```
/*
```

** Retorna um objeto Proxy que envolve o, delegando
alloperations para
esse objeto após registrar cada operação. objname é
uma string que
aparecerá nas mensagens de log para identificar o objeto.
Ifo tem próprio
cujos valores são objetos ou funções, então sevocê
consultar
o valor dessas propriedades, você obterá um
loggingProxyback, de modo que
o comportamento de registro deste proxy é
"contagioso".*/function loggingProxy(o, objname)
{
Defina manipuladores para nosso objeto Proxy de registro.//
Cada manipulador registra uma mensagem e, em seguida,
delega para o objeto de destino.
manipuladores const = {
Este manipulador é um caso especial porque para o próprio
Propriedades
cujo valor é um objeto ou função, ele retorna um
proxy em vez de
do que retornar o valor
itself.get(target, property, receiver) {
Registre o get
operationconsole.log('Handler
get(\${objname},\${property.toString()}');

Use a API Reflect para obter a propriedade
valuelet value = Reflect.get(target, property,
receptor);

Se a propriedade for uma propriedade própria do
alvo e
o valor é um objeto ou função e, em seguida, retorna
um proxy para ele.
if (Reflect.ownKeys(target).includes(property) &&
(typeof value === "object" || typeof value
== "função")) {
return loggingProxy(valor,
'\${objname}.\${property.toString()}');
}

Caso contrário, retorne o valor
unmodified.return value;*

,

*Não há nada de especial nos três seguintes
Métodos:*

*eles registram a operação e delegam ao destino
objeto.*

*Eles são um caso especial simplesmente para que possamos evitar
registrando o
objeto receptor que pode causar infinitos*

Recursão.

```
set(alvo, prop, valor, receptor) {  
    console.log('Manipulador  
set(${objname},${prop.toString()},${value})');  
    return Reflect.set(destino, prop, valor,  
receptor);  
},apply(destino, receptor,  
args) {  
    console.log('Manipulador ${objname}(${args})');  
    return Reflect.apply(destino, receptor,  
argumentos);},construct(destino, argumentos,  
receptor) {  
    console.log('Manipulador ${objname}(${args})'); return  
Reflect.construct(alvo, args, receptor);};
```

*Podemos gerar automaticamente o resto dos
manipuladores.*

Metaprogramação FTW!

```
Reflect.ownKeys(Reflect).forEach(handlerName => {  
    se (!( handlerName em manipuladores)) {  
        manipuladores[handlerName] = function(destino, ... args)  
{  
            Registre o operationconsole.log('Handler  
${handlerName}  
(${absurdo},${args})');  
            Delegar a operaçãoreturn Reflect[handlerName](target,  
... args);};}});
```

```
Retornar um proxy para o objeto usando esses  
logginghandlers  
return new Proxy(o,  
manipuladores);}
```

A função loggingProxy() definida anteriormente cria proxies que registram todas as maneiras como são usados. Se você está tentando entender como uma função não documentada usa os objetos que você passa, usar um loggingproxy pode ajudar.

Considere os exemplos a seguir, que resultam em alguns insights genuínos sobre a iteração de array:

```
Defina uma matriz de dados e um objeto com uma  
função propertylet data = [10,20]; let métodos = {  
quadrado: x => x*x };
```

```
Crie proxies de log para o array e o objectlet  
proxyData = loggingProxy(data, "data"); let  
proxyMethods = loggingProxy(métodos, "métodos");
```

```
Suponha que queremos entender como o Array.map()  
methodworksdata.map(methods.square)// => [100, 400]
```

```
Primeiro, vamos tentar com um Proxy de registro  
arrayproxyData.map(methods.square)// => [100,  
400]// Ele produz este output:// Manipulador  
get(data,map)// Manipulador get(data,length)//  
Manipulador get(data,constructor)// Manipulador  
has(data,0)// Manipulador get(data,0)//  
Manipulador has(data,1)// Manipulador  
get(data,1)
```

```
Agora vamos tentar com um método proxy  
objectdata.map(proxyMethods.square)// => [100,  
400]// Saída do log:
```

```
Manipulador get(methods,square)//
Manipulador methods.square(10,0,10,20)//
Manipulador methods.square(20,1,10,20)
```

```
Finalmente, vamos usar um proxy de registro para
aprender sobre o protocolo de iteração para(let x de
proxyData) console.log("Datum", x); // Log output://
Manipulador get(data,Symbol(Symbol.iterator))//
Manipulador get(data,length)// Manipulador
get(data,0)// Datum 10// Manipulador get(data,length)//
Manipulador get(data,1)// Datum 20// Manipulador
get(data,length)
```

A partir da primeira parte da saída do log, aprendemos que o método Array.map() verifica explicitamente a existência de cada elemento array (fazendo com que o manipulador has() seja invocado) antes de realmente ler o valor do elemento (que aciona o manipulador get()). Presumivelmente, isso é para que ele possa distinguir elementos de matriz inexistentes de elementos que existem, mas são indefinidos.

A segunda parte da saída de log pode nos lembrar que a função que passamos para Array.map() é invocada com três argumentos: o valor do elemento, o índice do elemento e o próprio array. (Há um problema em nossa saída de log: o método Array.toString() não inclui colchetes em sua saída, e as mensagens de log seriam mais claras se fossem incluídas na lista de argumentos (10,0,[10,20]).)

A terceira parte da saída de log nos mostra que o loop for/of funciona procurando um método com nome simbólico

[Symbol.iterator]. Ele também demonstra que a implementação da classe Array desse método iterador tem o cuidado de verificar o comprimento da matriz em cada iteração e não pressupõe que o comprimento da matriz permaneça constante durante a iteração.

14.7.1 Invariantes de proxy

A função `readOnlyProxy()` definida anteriormente cria `Proxyobjects` que são efetivamente congelados: qualquer tentativa de alterar um valor de propriedade ou atributo de propriedade ou de adicionar ou remover propriedades lançará uma exceção. Mas, desde que o objeto de destino não esteja congelado, descobriremos que, se pudermos consultar o proxy com `Reflect.isExtensible()` e `Reflect.getOwnPropertyDescriptor()`, ele nos dirá que devemos ser capazes de definir, adicionar e excluir propriedades. Só `readOnlyProxy()` cria objetos em um estado inconsistente. Poderíamos corrigir isso adicionando os manipuladores `isExtensible()` e `getOwnPropertyDescriptor()`, ou podemos simplesmente conviver com esse tipo de pequena inconsistência.

A API do manipulador de proxy nos permite definir objetos com grandes inconsistências e, neste caso, a própria classe `Proxy` nos impedirá de criar objetos `Proxy` que sejam inconsistentes de maneira inadequada. No início desta seção, descrevemos proxies como objetos sem comportamento próprio porque eles simplesmente encaminham todas as operações para o objeto manipuladores e o objeto de destino. Mas isso não é totalmente verdade: depois de encaminhar uma operação, a classe `Proxy` executa algumas verificações de sanidade no resultado para garantir que invariáveis importantes do JavaScript não sejam violadas. Se detectar uma violação, o proxy lançará `a TypeError` em vez de permitir que a operação continue.

Por exemplo, se você criar um proxy para um objeto não extensível, o proxy lançará um TypeError se o manipulador isExtensible() ever returns true:

```
let alvo = Object.preventExtensions({}); let proxy = new
Proxy(destino, { isExtensible() { return true;}});
Reflect.isExtensible(proxy); // ! TypeError: violação
invariável
```

Da mesma forma, objetos proxy para destinos não extensíveis podem não ter o manipulador agetPrototypeOf() que retorna nada além do objeto protótipo real do destino. Além disso, se o objeto de destino tiver propriedades não graváveis e não configuráveis, a classe Proxy lançará um TypeError se o manipulador get() retornar algo diferente do valor real:

```
let alvo = Object.freeze({x: 1}); let proxy = new
Proxy(destino, { get() { return 99; }}); proxy.x; // !
TypeError: o valor retornado por get() não corresponde
ao destino
```

O proxy impõe uma série de invariáveis adicionais, quase todas relacionadas a objetos de destino não extensíveis e propriedades não configuráveis no objeto de destino.

14.8 Resumo

Neste capítulo, você aprendeu:

- Os objetos JavaScript têm um atributo extensível e as propriedades de objeto têm gravável, enumerável e configurável

- atributos, bem como um valor e um atributo getter e/ou setter. Você pode usar esses atributos para "bloquear" seus objetos de várias maneiras, incluindo a criação de objetos "selados" e "congelados". JavaScript define funções que permitem percorrer a cadeia de protótipos de um objeto e até mesmo alterar o protótipo de um objeto (embora isso possa tornar seu código mais lento). As propriedades do objeto Symbol têm valores que são "símbolos conhecidos", que você pode usar como nomes de propriedade ou método para os objetos e classes que você define. Isso permite que você controle como seu objeto interage com recursos da linguagem JavaScript e com a biblioteca principal. Por exemplo, símbolos conhecidos permitem que você torne suas classes iteráveis e controle a string que é exibida quando uma instância é passada para Object.prototype.toString(). Antes do ES6, esse tipo de personalização estava disponível apenas para as classes nativas que foram incorporadas a uma implementação. Os literais de modelo marcados são uma sintaxe de invocação de função e definir uma nova função de tag é como adicionar uma nova sintaxe literal à linguagem. Definir uma função de tag que analisa seu argumento de string de modelo permite que você incorpore DSLs no código JavaScript. As funções de tag também fornecem acesso a uma forma bruta e sem escape de literais de cadeia de caracteres em que as barras invertidas não têm nenhum significado especial. A classe Proxy e a API Reflect relacionada permitem controle de baixo nível sobre os comportamentos fundamentais dos objetos JavaScript. Os objetos proxy podem ser usados como wrappers opcionalmente revogáveis para melhorar o encapsulamento de código e também podem ser usados para implementar comportamentos de objeto não padrão (como algumas das APIs de casos especiais definidas pelos primeiros navegadores da web).

1 Um bug no mecanismo JavaScript V8 significa que esse código não funciona corretamente no Node13.

Capítulo 15. JavaScript em navegadores da Web

A linguagem JavaScript foi criada em 1994 com o propósito expresso de permitir o comportamento dinâmico nos documentos exibidos pelos navegadores da web. A linguagem evoluiu significativamente desde então e, ao mesmo tempo, o escopo e os recursos da plataforma web cresceram explosivamente. Hoje, os programadores de JavaScript podem pensar na web como uma plataforma completa para o desenvolvimento de aplicativos. Os navegadores da Web são especializados na exibição de texto e imagens formatados, mas, como os sistemas operacionais nativos, os navegadores também fornecem outros serviços, incluindo gráficos, vídeo, áudio, rede, armazenamento e threading. JavaScript é a linguagem que permite que os aplicativos da Web usem os serviços fornecidos pela plataforma da Web, e este capítulo demonstra como você pode usar o mais importante desses serviços.

O capítulo começa com o modelo de programação da plataforma web, explicando como os scripts são incorporados nas páginas HTML (§15.1) e como o código JavaScript é acionado de forma assíncrona por eventos (§15.2). As seções que seguem este material introdutório documentam as APIs coreJavaScript que permitem que seus aplicativos Web:

- Controlar o conteúdo do documento (§15.3) e o estilo
 - (§15.4) Determinar a posição na tela dos elementos do documento (§15.5)
-

- Criar componentes de interface do usuário
- reutilizáveis ([§15.6](#))
- Desenhar gráficos ([§15.7](#) e [§15.8](#))
- Reproduzir e gerar sons ([§15.9](#))
- Gerenciar navegação e histórico do navegador ([§15.10](#))
- Trocar dados pela rede ([§15.11](#))
- Armazenar dados no computador do usuário ([§15.12](#))
- Executar computação simultânea com threads ([§15.13](#))

JAVASCRIPT DO LADO DO CLIENTE

Neste livro e na Web, você verá o termo "JavaScript do lado do cliente". O termo é simplesmente um sinônimo de JavaScript escrito para ser executado em um navegador da web e contrasta com o código do "lado do servidor", que é executado em servidores da web.

Os dois "lados" referem-se às duas extremidades da conexão de rede que separam o servidor web e o navegador web, e o desenvolvimento de software para a web normalmente requer que o código seja escrito em ambos os "lados". O lado do cliente e o lado do servidor também são frequentemente chamados de "front-end" e "back-end".

As edições anteriores deste livro tentaram cobrir de forma abrangente todas as APIs JavaScript definidas por navegadores da web e, como resultado, este livro era muito longo há uma década. O número e a complexidade das APIs da web continuaram a crescer, e não acho mais que faça sentido tentar cobri-las todas em um livro. A partir da sétima edição, meu objetivo é cobrir a linguagem JavaScript definitivamente e fornecer uma introdução aprofundada ao uso da linguagem com o Node e com navegadores da web. Este capítulo não pode cobrir todas as APIs da web, mas apresenta as mais importantes com detalhes suficientes para que você possa começar a usá-las imediatamente. E, tendo aprendido sobre as APIs principais abordadas aqui, você deve ser capaz de escolher novas APIs (como as resumidas em §15.15) quando e se precisar delas.

O Node tem uma única implementação e uma única fonte autorizada para documentação. As APIs da Web, por outro lado, são definidas por consenso entre os principais fornecedores de navegadores da Web, e a documentação oficial assume a forma de uma especificação destinada aos programadores de C++ que implementam a API, não aos programadores de JavaScript que a usarão. Felizmente, o projeto "MDN web docs" da Mozilla é uma fonte confiável e abrangente para documentação de API da web.

APIS LEGADAS

Nos 25 anos desde que o JavaScript foi lançado pela primeira vez, os fornecedores de navegadores adicionaram recursos e APIs para os programadores usarem. Muitas dessas APIs agora estão obsoletas. Eles incluem:

- APIs proprietárias que nunca foram padronizadas e/ou implementadas por outros fornecedores de navegadores. O Internet Explorer da Microsoft definiu muitas dessas APIs. Alguns (como a propriedade `innerHTML`) se mostraram úteis e acabaram sendo padronizados. Outros (como o método `attachEvent()`) estão obsoletos há anos.
- APIs ineficientes (como o método `document.write()`) que têm um impacto tão severo no desempenho que seu uso não é mais considerado aceitável.
- APIs desatualizadas que há muito foram substituídas por novas APIs para alcançar a mesma coisa. Um exemplo é `document.bgColor`, que foi definido para permitir que o JavaScript defina a cor de fundo de um documento. Com o advento do CSS, `document.bgColor` tornou-se um caso especial sem propósito real.
- APIs mal projetadas que foram substituídas por outras melhores. Nos primórdios da web, os comitês de padrões definiram a API chave do Document Object Model de maneira independente de linguagem para que a mesma API pudesse ser usada em programas Java para trabalhar com documentos XML e em programas JavaScript para trabalhar com documentos HTML. Isso resultou em uma API que não era adequada para a linguagem JavaScript e que tinha recursos com os quais os programadores da web não se importavam particularmente. Demorou décadas para se recuperar desses primeiros erros de design, mas os navegadores da Web de hoje oferecem suporte a um Document Object Model muito aprimorado.

Os fornecedores de navegadores podem precisar oferecer suporte a essas APIs herdadas no futuro próximo para garantir a compatibilidade com versões anteriores, mas não há mais necessidade de este livro documentá-las ou de você aprender sobre elas. A plataforma web amadureceu e se estabilizou, e se você é um desenvolvedor web experiente que se lembra da quarta ou quinta edição deste livro, então você pode ter tanto conhecimento desatualizado para esquecer quanto tem novo material para aprender.

15.1 Noções básicas de programação web

Esta seção explica como os programas JavaScript para a web são estruturados, como eles são carregados em um navegador da web, como eles obtêm entrada, como eles produzem saída e como eles são executados de forma assíncrona respondendo a eventos.

15.1.1 JavaScript em tags HTML <script>

Os navegadores da Web exibem documentos HTML. Se você deseja que um navegador da Web execute código JavaScript, você deve incluir (ou referenciar) esse código de um documento HTML, e é isso que a <script> tag HTML faz.

O código JavaScript pode aparecer embutido em um arquivo HTML entre <script> e </script> tags. Aqui, por exemplo, está um arquivo HTML que inclui uma tag de script com código JavaScript que atualiza dinamicamente um elemento do documento para fazê-lo se comportar como um relógio digital:

```
<!DOCTYPE html>                                <!-- Este é um arquivo HTML5 -->
<html>                                         <!-- O elemento raiz -->
  >>                                         <!-- Título, scripts e estilos
  <head>                                         pode ir aqui -->
    <title>Relógio</title>                         /* Uma folha de estilo CSS para o
    <style>Digital                                relógio
      relógio                                         /* Estilos se aplicam ao elemento
      *#clock {                                     com id="relógio" */
        fonte: Negrito 24px sem serifa             /* Use uma fonte grande em negrito */
        fundo: #ddf;                               /* em um cinza-azulado claro
        fundo.                                         */
      /*preenchimento: 15px;                      /* Cerque-o com alguns
        espaço*/                                     espaços*/
```

```

        Borda: Preto Sólido 2px;      /* e uma borda preta sólida
*/
        raio da borda: 10px;          /* com cantos arredondados. */
</style>
>
</head>
<body>           <!-- O corpo contém o conteúdo de
o documento. -->
<h1>Relógio digital</h1> <!-- Exibir um título. -->
<span id="relógio"></span> <!-- Vamos inserir o tempo em
este elemento. --><script> Define uma função
para exibir a função timefunction displayTime()
{



let clock = document.querySelector("#clock"); Getcom
id="clock"
    let now = new Data();                      Obter
Hora atual
clock.textContent = now.toLocaleTimeString(); Displaytime
in the clock}displayTime()// Exibe a hora
rightawaysetInterval(displayTime, 1000); // E atualize-a a
cada segundo.</script></body></html>

```

Embora o código JavaScript possa ser incorporado diretamente em uma `<script>` tag, é mais comum usar o atributo `src` da `<script>` tag para especificar a URL (uma URL absoluta ou uma URL relativa à URL do arquivo HTML que está sendo exibido) de um arquivo que contém código JavaScript. Se retirássemos o código JavaScript deste arquivo HTML e o armazenássemos em seu próprio arquivo `scripts/digital_clock.js`, a `<script>` tag poderia fazer referência a esse arquivo de código assim:

```
<script src="scripts/digital_clock.js"></script>
```

Um arquivo JavaScript contém JavaScript puro, sem <script> tags ou qualquer outro HTML. Por convenção, os arquivos de código JavaScript têm nomes que terminam com .js.

Uma <script> tag com o atributo a src se comporta exatamente como se o conteúdo do arquivo JavaScript especificado aparecesse diretamente entre as tags <script> e </script>. Observe que a tag de fechamento </script> é necessária em documentos HTML mesmo quando o atributo src é especificado: HTML não suporta uma <script/> tag.

Há uma série de vantagens em usar o atributo src:

- Ele simplifica seus arquivos HTML, permitindo que você remova grandes blocos de código JavaScript deles, ou seja, ajuda a manter o conteúdo e o comportamento separados. Quando várias páginas da Web compartilham o mesmo código JavaScript, o uso do atributo src permite que você mantenha apenas uma única cópia desse código, em vez de ter que editar cada arquivo HTML quando o código for alterado. Se um arquivo de código JavaScript for compartilhado por mais de uma página, ele só precisará ser baixado uma vez, pela primeira página que o usar - as páginas subsequentes podem recuperá-lo do cache do navegador. Como o atributo src usa uma URL arbitrária como seu valor, um programa JavaScript ou página da Web de um servidor da Web pode empregar código exportado por outros servidores da Web. Muita publicidade na Internet depende desse fato.

MODULES 10.3 documenta os módulos JavaScript e cobre sua importação e

diretivas de exportação. Se você escreveu seu programa JavaScript usando módulos (e não usou uma ferramenta de empacotamento de código para combinar todos os seus módulos em um único arquivo não modular de JavaScript), então você deve carregar o módulo de nível superior do seu programa com uma `<script>` tag que tem um atributo `type="module"`. Se você fizer isso, o módulo especificado será carregado e todos os módulos importados serão carregados e (recursivamente) todos os módulos importados serão carregados. Consulte §10.3.5 para obter detalhes completos.

ESPECIFICANDO O TIPO DE SCRIPT Nos primórdios da web, pensava-se que os navegadores poderiam um dia implementar outras linguagens além do JavaScript, e os programadores adicionaram atributos como `language="javascript"` e `type="application/javascript"` às suas `<script>` tags. Isso é completamente desnecessário. JavaScript é a linguagem padrão (e única) da web. O atributo `language` foi preterido e há apenas dois motivos para usar um atributo `type` em uma `<script>` tag:

- Para especificar que o script é um módulo
- Para incorporar dados em uma página da Web sem exibi-los (consulte §15.3.4)

QUANDO OS SCRIPTS SÃO EXECUTADOS: ASSÍNCRONO E DIFERIDO Quando o JavaScript foi adicionado pela primeira vez aos navegadores da web, não havia API para percorrer e manipular a estrutura e o conteúdo de um documento já renderizado. A única maneira de o código JavaScript afetar o conteúdo de um documento era gerar esse conteúdo em tempo real enquanto o documento estava em processo de carregamento. Ele fez isso usando o comando

`document.write()` para injetar texto HTML no documento no local do script.

O uso de `document.write()` não é mais considerado um bom estilo, mas o fato de ser possível significa que quando o analisador HTML encontra um `<script>` elemento, ele deve, por padrão, executar o script apenas para ter certeza de que ele não produz nenhum HTML antes de poder retomar a análise e renderização do documento. Isso pode diminuir drasticamente a análise e a renderização da página da Web.

Felizmente, esse modo de execução de script síncrono ou de bloqueio padrão não é a única opção. A `<script>` tag pode ter atributos `defer` e `async`, que fazem com que os scripts sejam executados de forma diferente. Esses são atributos booleanos - eles não têm um valor; eles só precisam estar presentes na `<script>` etiqueta. Observe que esses atributos só são significativos quando usados em conjunto com o atributo `src`:

```
<script defer src="deferred.js"></script>
<script async src="async.js"></script>
```

Os atributos `defer` e `async` são maneiras de informar ao navegador que o script vinculado não usa `document.write()` para gerar saída HTML e que o navegador, portanto, pode continuar a analisar e renderizar o documento enquanto baixa o script. O atributo `defer` faz com que o navegador adie a execução do script até que o documento tenha sido totalmente carregado e analisado e esteja pronto para ser manipulado. O atributo `async` faz com que o navegador execute o script o mais rápido possível, mas não bloqueia a análise de documentos enquanto o script está sendo baixado. Se uma `<script>` tag tiver ambos os atributos, o

`async` tem precedência.

Observe que os scripts adiados são executados na ordem em que aparecem no documento. Os scripts assíncronos são executados à medida que são carregados, o que significa que eles podem ser executados fora de ordem.

Os scripts com o atributo `type="module"` são, por padrão, executados após o carregamento do documento, como se tivessem um atributo `defer`. Você pode substituir esse padrão pelo atributo `async`, o que fará com que o código seja executado assim que o módulo e todas as suas dependências forem carregados.

Uma alternativa simples para os atributos `async` e `defer` — especialmente para o código incluído diretamente no HTML — é simplesmente colocar seus scripts no final do arquivo HTML. Dessa forma, o script pode ser executado sabendo que o conteúdo do documento antes dele foi analisado e está pronto para ser manipulado.

CARREGANDO SCRIPTS SOB DEMANDA Som vezes, você pode ter um código JavaScript que não é usado quando um documento é carregado pela primeira vez e só é necessário se o usuário realizar alguma ação, como clicar em um botão ou abrir um menu. Se você estiver desenvolvendo seu código usando módulos, poderá carregar um módulo sob demanda com `import()`, conforme descrito em §10.3.6.

Se você não estiver usando módulos, poderá carregar um arquivo de JavaScript sob demanda simplesmente adicionando uma `<script>` tag ao seu documento quando quiser que o script seja carregado:

Carregar e executar de forma assíncrona um script a partir de um URL especificado// Retorna uma promessa que é resolvida quando o script é carregado.function

```
importScript(url) {  
  
    return new Promise((resolve, reject) => {  
        let s = document.createElement("script"); Crie um  
<script> elemento  
        s.onload = () => { resolve(); }; Resolver  
        promessa quando carregada  
        s.onerror = (e) => { reject(e); }; Rejeitar  
        em caso de falha  
        Definir URL do script  
  
        document.head.append(s); Adicionar  
        <script> para documentar  
    });}
```

Essa função importScript() usa APIs DOM (§15.3) para criar uma nova `<script>` tag e adicioná-la ao documento `<head>`. Ele usa manipuladores de eventos (§15.2) para determinar quando o script foi carregado com êxito ou quando o carregamento falhou.

15.1.2 O modelo de objeto do documento

Um dos objetos mais importantes na programação JavaScript do lado do cliente é o objeto Document, que representa o documento HTML exibido em uma janela ou guia do navegador. A API para trabalhar com documentos HTML é conhecida como Document Object Model, ou DOM, e é abordada em detalhes no §15.3. Mas o DOM é tão central para a programação JavaScript do lado do cliente que merece ser introduzido aqui.

Os documentos HTML contêm elementos HTML aninhados uns nos outros,

formando uma árvore. Considere o seguinte documento HTML simples:

```
<html>
<head>
  <title>Exemplo de
  documento</title></head><body>

<h1>Um documento</h1><p>HTML Este é um
documento <i>simples</i>.</p></body></html>
```

A marca de nível superior `<html>` contém `<head>` marcas e `<body>`. A `<head>` tag contém uma `<title>` tag. E a `<body>` tag contém `<h1>` e `<p>` tags. As `<title>` `<h1>` tags and contêm strings detext, e a `<p>` tag contém duas strings de texto com uma `<i>` tag entre elas.

A API DOM espelha a estrutura em árvore de um documento HTML. Para cada tag HTML no documento, há um objeto `JavaScriptElement` correspondente e, para cada execução de texto no documento, há um objeto `Text` correspondente. As classes `Element` e `Text`, bem como a própria classe `Document`, são todas subclasses da classe `Node` mais geral, e os objetos `Node` são organizados em uma estrutura de árvore que o JavaScript pode consultar e percorrer usando a API DOM. A representação DOM deste documento é a árvore representada na Figura 15-1.

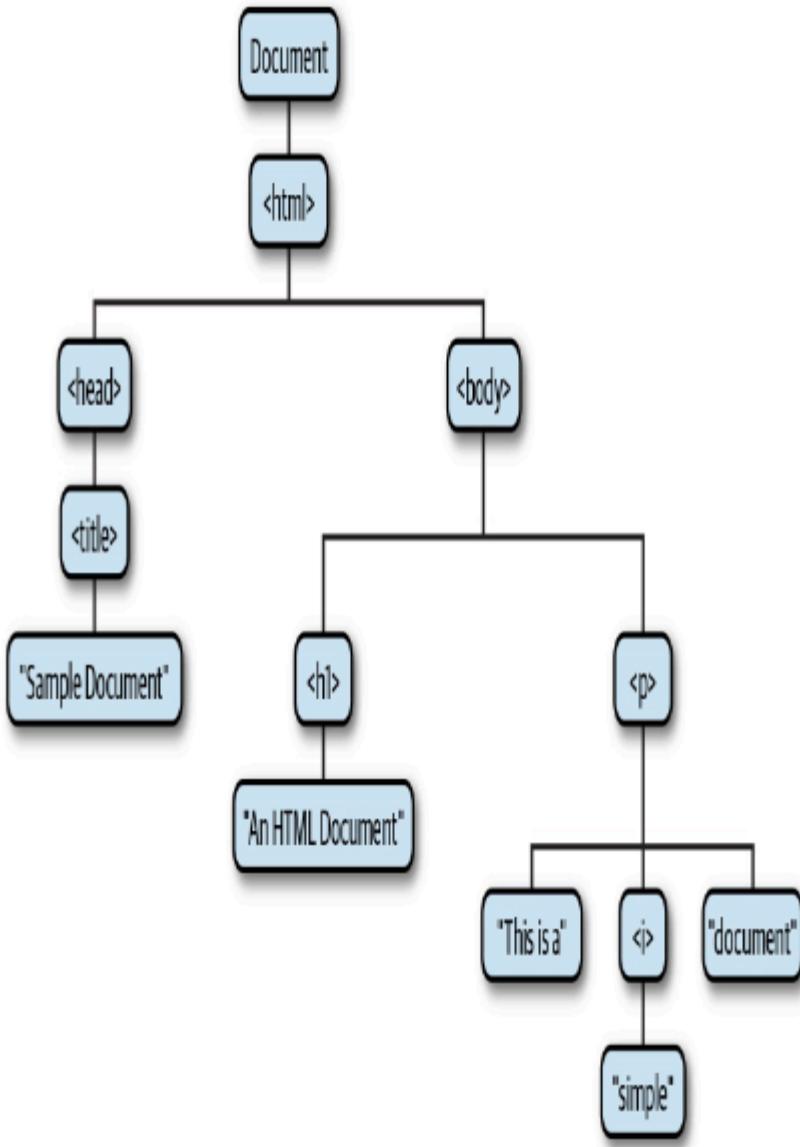


Figura 15-1. A representação em árvore de um documento HTML

Se você ainda não estiver familiarizado com estruturas de árvore no computador

programação, é útil saber que eles emprestam terminologia de árvores genealógicas. O nó diretamente acima de um nó é o pai desse nó. Os nós de um nível diretamente abaixo de outro nó são filhos desse nó. Os nós no mesmo nível e com o mesmo pai são irmãos. O conjunto de nós de qualquer número de níveis abaixo de outro nó são os descendentes desse nó. E o pai, o avô e todos os outros nós acima de um nó são os ancestrais desse nó.

A API DOM inclui métodos para criar novos Element e Textnodes e para inseri-los no documento como filhos de objetos otherElement. Existem também métodos para mover elementos dentro do documento e removê-los completamente. Enquanto um aplicativo do lado do servidor pode produzir saída de texto sem formatação escrevendo strings with console.log(), um aplicativo JavaScript do lado do cliente pode produzir saída HTML formatada criando ou manipulando o documento da árvore de documentos usando a API DOM.

Há uma classe JavaScript correspondente a cada tipo de tag HTML, e cada ocorrência da tag em um documento é representada por uma instância da classe. A <body> tag, por exemplo, é representada por uma instância de HTMLBodyElement, e uma <table> tag é representada por uma instância de HTMLTableElement. Os objetos de elemento JavaScript têm propriedades que correspondem aos atributos HTML das tags. Por exemplo, instâncias de HTMLImageElement, que representam tags, têm uma propriedade src que corresponde ao atributo src da tag. O valor inicial da propriedade src é o valor do atributo que aparece na tag HTML, e definir essa propriedade com JavaScript altera o valor do atributo HTML (e faz com que o navegador

carregar e exibir uma nova imagem). A maioria das classes de elementos JavaScript apenas espelha os atributos de uma tag HTML, mas algumas definem métodos adicionais. As classes `HTMLAudioElement` e `HTMLVideoElement`, por exemplo, definem métodos como `play()` e `pause()` para controlar a reprodução de arquivos de áudio e vídeo.

15.1.3 O objeto global em navegadores da Web

Há um objeto global por janela ou guia do navegador (§3.7). Todo o código JavaScript (exceto o código em execução em threads de trabalho; consulte §15.13) em execução nessa janela compartilha esse único objeto global. Isso é verdade independentemente de quantos scripts ou módulos estejam no documento: todos os scripts e módulos de um documento compartilham um único objeto global; Se o OneScript definir uma propriedade nesse objeto, essa propriedade também ficará visível para todos os outros scripts.

O objeto global é onde a biblioteca padrão do JavaScript é definida - a função `parseInt()`, o objeto `Math`, a classe `Set` e assim por diante. Em navegadores da web, o objeto global também contém os principais pontos de entrada devárias APIs da web. Por exemplo, a propriedade `document` representa o documento exibido no momento, o método `fetch()` faz solicitações de rede HTTP e o construtor `Audio()` permite que os programas JavaScript reproduzam sons.

Nos navegadores da web, o objeto global tem dupla função: além de definir tipos e funções integrados, ele também representa a janela atual do navegador da web e define propriedades como `history` (§15.10.2), que representa o histórico de navegação da janela, e `innerWidth`, que contém a largura da janela em pixels. Uma das propriedades deste

global é chamado de window e seu valor é o próprio objeto global. Isso significa que você pode simplesmente digitar window para se referir ao objeto global em seu código do lado do cliente. Ao usar recursos específicos da janela, geralmente é uma boa ideia incluir uma janela. prefix>window.innerWidth é mais claro que innerWidth, por exemplo.

15.1.4 Scripts compartilham um namespace

Com módulos, as constantes, variáveis, funções e classes definidas no nível superior (ou seja, fora de qualquer função ou definição de classe) do módulo são privadas para o módulo, a menos que sejam explicitamente exportadas, caso em que podem ser importadas seletivamente por outros módulos. (Observe que essa propriedade dos módulos também é respeitada pelas ferramentas de agrupamento de código.)

Com scripts que não são módulos, no entanto, a situação é completamente diferente. Se o código de nível superior em um script definir uma constante, variável, função ou classe, essa declaração ficará visível para todos os outros scripts no mesmo documento. Se um script define uma função f() e outro script define uma classe c, então um terceiro script pode invocar a função e instanciar a classe sem ter que executar nenhuma ação para importá-los. Portanto, se você não estiver usando módulos, os scripts independentes em seu documento compartilham um único namespace e se comportam como se todos fizessem parte de um único script maior. Isso pode ser conveniente para programas pequenos, mas a necessidade de evitar conflitos de nomenclatura pode se tornar problemática para programas maiores, especialmente quando alguns dos scripts são bibliotecas de terceiros.

Existem algumas peculiaridades históricas sobre como esse namespace compartilhado funciona. var e declarações de função no nível superior criar

no objeto global compartilhado. Se um script define uma função de nível superior `f()`, outro script no mesmo documento pode invocar essa função como `f()` ou como `window.f()`. Por outro lado, as declarações ES6 `const`, `let` e `class`, quando usadas no nível superior, não criam propriedades no objeto global. Eles ainda são definidos em um namespace compartilhado, no entanto: se um script define uma classe `C`, outros scripts serão capazes de criar instâncias dessa classe com `new C()`, mas não com `new window.C()`.

Para resumir: nos módulos, as declarações de nível superior têm como escopo o módulo e podem ser exportadas explicitamente. Em scripts não módulo, no entanto, as declarações de nível superior têm como escopo o documento que o contém e as declarações são compartilhadas por todos os scripts no documento. As declarações `var` e `function` mais antigas são compartilhadas por meio de propriedades do objeto global. As declarações `const`, `let` e `class` mais recentes também são compartilhadas e têm o mesmo escopo de documento, mas não existem como propriedades de nenhum objeto ao qual o código JavaScript tenha acesso.

15.1.5 Execução de programas JavaScript

Não existe uma definição formal de um programa em JavaScript do lado do cliente, mas podemos dizer que um programa JavaScript consiste em todo o código JavaScript em um documento ou referenciado a partir dele. Esses bits separados de codeshare compartilham um único objeto `Window` global, o que lhes dá acesso ao mesmo objeto `Document` subjacente que representa o documento HTML. Os scripts que não são módulos também compartilham um namespace de nível superior.

Se uma página da Web incluir um quadro incorporado (usando o `<iframe>` elemento), o código JavaScript no documento incorporado terá um

objeto global e objeto Document diferentes do código nodocumento de incorporação, e pode ser considerado um JavaScriptprograma separado. Lembre-se, porém, de que não há uma definição formal de quais são os limites de um programa JavaScript. Se o documento de contêiner e o documento contido forem carregados do mesmo servidor, o código em um documento poderá interagir com o código no outro e você poderá tratá-los como duas partes interagindo de um único programa, se desejar. §15.13.6 explica como um programa JavaScript pode enviar e receber mensagens de e para o código JavaScript em execução em um <iframe>.

Você pode pensar na execução do programa JavaScript como ocorrendo em duas fases. Na primeira fase, o conteúdo do documento é carregado e o código dos <script> elementos (scripts embutidos e externos) é executado. Os scripts geralmente são executados na ordem em que aparecem no documento, embora essa ordem padrão possa ser modificada pelos atributos `async` e `defer` que descrevemos. O código JavaScript dentro de qualquer script é executado de cima para baixo, sujeito, é claro, às condicionais, loops e outras instruções de controle do JavaScript. Alguns scripts realmente não fazem nada durante esta primeira fase e, em vez disso, apenas definem funções e classes para uso na segunda fase. Outros scripts podem fazer um trabalho significativo durante a primeira fase e depois não fazer nada na segunda. Imagine um script no final de um documento que localiza todas <h1> as marcas no <h2> documento e modifica o documento gerando e inserindo um sumário no início do documento. Isso poderia ser feito inteiramente na primeira fase. (Consulte §15.3.6 para obter um exemplo que faz exatamente isso.)

Depois que o documento é carregado e todos os scripts são executados, o JavaScript

A execução entra em sua segunda fase. Esta fase é assíncrona e orientada por eventos. Se um script vai participar dessa segunda fase, uma das coisas que ele deve ter feito durante a primeira fase é registrar pelo menos um manipulador de eventos ou outra função de retorno de chamada que será invocada de forma assíncrona. Durante essa segunda fase controlada por eventos, o navegador da Web invoca funções de manipulador de eventos e outros retornos de chamada em resposta a eventos que ocorrem de forma assíncrona. Os manipuladores de eventos são mais comumente invocados em resposta à entrada do usuário (cliques do mouse, pressionamentos de tecla, etc.), mas também podem ser acionados por atividade de rede, carregamento de documentos e recursos, tempo decorrido ou erros no código JavaScript. Eventos e manipuladores de eventos são descritos em detalhes em §15.2.

Alguns dos primeiros eventos a ocorrer durante a fase orientada a eventos são os eventos "DOMContentLoaded" e "load". "DOMContentLoaded" é acionado quando o documento HTML foi completamente carregado e analisado. O evento "load" é acionado quando todos os recursos externos do documento, como imagens, também são totalmente carregados. JavaScript geralmente usam um desses eventos como um gatilho ou sinal de partida. É comum ver programas cujos scripts definem funções, mas não tomam nenhuma ação além de registrar uma função de manipulador de eventos a ser acionada pelo evento "load" no início da fase de execução orientada a eventos. É esse manipulador de eventos de "carregamento" que manipula o documento e faz o que quer que o programa deva fazer. Observe que é comum na programação JavaScript que uma função de manipulador de eventos, como o manipulador de eventos "load" descrito aqui, registre outros manipuladores de eventos.

A fase de carregamento de um programa JavaScript é relativamente curta: idealmente menos de um segundo. Depois que o documento é carregado, o comando orientado a eventos

dura enquanto o documento é exibido pelo navegador da web. Como essa fase é assíncrona e orientada a eventos, pode haver longos períodos de inatividade em que nenhum JavaScript é executado, pontuados por rajadas de atividade acionadas por eventos de usuário ou rede. Abordaremos essas duas fases com mais detalhes a seguir.

MODELO DE THREADING JAVASCRIPT DO LADO DO CLIENTE JavaScript é uma linguagem de thread único e execução de thread único torna a programação muito mais simples: você pode escrever código com a garantia de que dois manipuladores de eventos nunca serão executados ao mesmo tempo. Você pode manipular o conteúdo do documento sabendo que nenhum outro thread está tentando modificá-lo ao mesmo tempo, e você nunca precisa se preocupar com bloqueios, deadlock ou condições de corrida ao escrever código JavaScript.

A execução de thread único significa que os navegadores da Web param de responder à entrada do usuário enquanto os scripts e manipuladores de eventos estão em execução. Isso sobrecarrega os programadores de JavaScript: significa que os scripts JavaScript e os manipuladores de eventos não devem ser executados por muito tempo. Se um script executar uma tarefa computacionalmente intensiva, ele introduzirá um atraso no carregamento do documento e o usuário não verá o conteúdo do documento até que o script seja concluído. Se um manipulador de eventos executar uma tarefa computacionalmente intensiva, o navegador poderá parar de responder, possivelmente fazendo com que o usuário pense que ele travou.

A plataforma da web define uma forma controlada de simultaneidade chamada "web worker". Um web worker é um thread em segundo plano para executar tarefas computacionalmente intensivas sem congelar a interface do usuário. O código executado em um thread de trabalho da Web não tem acesso a

conteúdo do documento, não compartilha nenhum estado com o thread principal ou com outros trabalhadores e só pode se comunicar com o thread principal e outros trabalhadores por meio de eventos de mensagem assíncrona, portanto, a simultaneidade não é detectável para o thread principal e os trabalhos da Web não alteram o modelo básico de execução de thread único de programas JavaScript. Consulte §15.13 para obter detalhes completos sobre o mecanismo de threading seguro da Web.

LINHA DO TEMPO DO JAVASCRIPT DO LADO DO CLIENTE Você já viu que os programas JavaScript começam em uma fase de execução de script e, em seguida, fazem a transição para uma fase de manipulação de eventos. Estas duas fases podem ser divididas nas seguintes etapas:

1. O navegador da Web cria um objeto Document e começa a analisar a página da Web, adicionando objetos Element e nós Text ao documento à medida que analisa elementos HTML e seu conteúdo textual. A propriedade document.readyState tem o valor "loading" neste estágio.
2. Quando o analisador HTML encontra uma <script> tag que não tem nenhum dos atributos async, defer, outype="module", ele adiciona essa tag de script ao documento e executa o script. O script é executado de forma síncrona e o analisador HTML pausa enquanto o script é baixado (se necessário) e executado. Um script como este pode usar document.write() para inserir texto no fluxo de entrada, e esse texto se tornará parte do documento quando o analisador for retomado. Um script como esse geralmente simplesmente define funções e registra manipuladores de eventos para uso posterior, mas pode atravessar e manipular a árvore de documentos como ela existe naquele momento. Ou seja, scripts que não são de módulo que não têm um atributo defer assíncrono podem ver sua própria <script> tag e

documento o conteúdo que vem antes dele.

3. Quando o analisador encontra um <script> elemento que tem o atributo assíncrono definido, ele começa a baixar o texto do script (e se o script for um módulo, ele também baixa recursivamente todas as dependências do script) e continua analisando o documento. O script será executado o mais rápido possível após o download, mas o analisador não para e espera que ele seja baixado. Os scripts assíncronos não devem usar o método document.write(). Eles podem ver sua própria <script> tag e todo o conteúdo do documento que vem antes dela, e podem ou não ter acesso ao conteúdo adicional do documento.

4. Quando o documento é completamente analisado, a propriedade document.readyState muda para "interativo".

5. Todos os scripts que tinham o atributo defer definido (junto com qualquer script de módulo que não tenha um atributo assíncrono) são executados na ordem em que apareceram no documento. Scripts assíncronos também podem ser executados neste momento. Os scripts adiados têm acesso ao documento completo e não devem usar o método document.write().

6. O navegador dispara um evento "DOMContentLoaded" no objeto Document. Isso marca a transição da fase de execução de script síncrono para a fase assíncrona e orientada a eventos da execução do programa. Observe, no entanto, que ainda pode haver scripts assíncronos que ainda não foram executados neste momento.

7. O documento é completamente analisado neste momento, mas o navegador ainda pode estar aguardando o carregamento de conteúdo adicional, como imagens. Quando todo esse conteúdo terminar de carregar e quando todos os scripts assíncronos tiverem sido carregados e executados, a propriedade document.readyState muda para "complete" e o navegador da Web dispara um evento "load" na janela

objeto.

8. Deste ponto em diante, os manipuladores de eventos são invocados de forma assíncrona em resposta a eventos de entrada do usuário, eventos de rede, timerexpirations e assim por diante.

15.1.6 Entrada e saída do programa

Como qualquer programa, os programas JavaScript do lado do cliente processam dados de entrada para produzir dados de saída. Há uma variedade de entradas disponíveis:

- O conteúdo do próprio documento, que o código JavaScript pode acessar com a API DOM (§15.3). A entrada do usuário, na forma de eventos, como cliques do mouse (ou toques na tela sensível ao toque) em <button> elementos HTML ou texto inserido em <textarea> elementos HTML, por exemplo. §15.2 demonstra como os programas JavaScript podem responder a eventos de usuário como esses. A URL do documento que está sendo exibido está disponível para JavaScript do lado do cliente como documento.URL. Se você passar essa string para o construtor URL() (§11.9), poderá acessar facilmente as seções de caminho, consulta e fragmento da URL. O conteúdo do cabeçalho da solicitação HTTP "Cookie" está disponível para o código do lado do cliente como document.cookie. Os cookies são geralmente usados pelo código do lado do servidor para manter as sessões do usuário, mas o código do lado do cliente também pode lê-los (e gravá-los), se necessário. Consulte §15.12.2 para obter mais detalhes. A propriedade global navigator fornece acesso a informações sobre o navegador da Web, o sistema operacional sobre o qual ele está sendo executado e os recursos de cada um. Por exemplo, navigator.userAgent é uma string que identifica o navegador da web, navigator.language é o preferido do usuário

`language` e `navigator.hardwareConcurrency` retorna o número de CPUs lógicas disponíveis para o navegador da Web. Da mesma forma, a propriedade global `screen` fornece acesso ao tamanho de exibição do usuário por meio das propriedades `screen.width` e `screen.height`. De certa forma, esses objetos `navigator` and `screen` são para os navegadores da web o que as variáveis de ambiente são para os programas Node. O JavaScript do lado do cliente normalmente produz saída, quando necessário, manipulando o documento HTML com a API DOM (§15.3) ou usando uma estrutura de nível superior, como React ou Angular, para manipular o documento. O código do lado do cliente também pode usar `console.log()` e métodos relacionados (§11.8) para produzir saída. Mas essa saída só é visível no console do desenvolvedor da Web, por isso é útil durante a depuração, mas não para a saída visível ao usuário.

15.1.7 Erros de programa

Ao contrário dos aplicativos (como aplicativos Node) que são executados diretamente no sistema operacional, os programas JavaScript em um navegador da Web não podem realmente "travar". Se ocorrer uma exceção enquanto o programa JavaScript estiver em execução e se você não tiver uma instrução `catch` para tratá-la, uma mensagem de erro será exibida no console do desenvolvedor, mas todos os manipuladores de eventos registrados continuarão em execução e respondendo aos eventos.

Se você quiser definir um manipulador de erros de último recurso a ser invocado quando esse tipo de exceção não detectada ocorrer, defina a propriedade `onerror` do objeto Window como uma função de manipulador de erros. Quando uma exceção não capturada se propaga por toda a pilha de chamadas e uma mensagem de erro está prestes a ser exibida no console do desenvolvedor, o

window.onerror será invocada com três stringarguments. O primeiro argumento para window.onerror é uma mensagem que descreve o erro. O segundo argumento é uma string que contém oURL do código JavaScript que causou o erro. O terceiro argumento é o número da linha dentro do documento em que ocorreu o erro. Se o manipulador de erros retornar true, ele informará ao navegador que o manipulador tratou o erro e que nenhuma ação adicional é necessária - em outras palavras, o navegador não deve exibir sua própria mensagem de erro.

Quando uma promessa é rejeitada e não há nenhuma função .catch() para lidar com ela, essa é uma situação muito parecida com uma exceção não tratada: um erro imprevisto ou um erro lógico em seu programa. Você pode detectar isso definindo uma função window.onunhandledrejection ou usando window.addEventListener() para registrar um manipulador para eventos "unhandledrejection". O objeto de evento passado para este manipulador terá uma propriedade promise cujo valor é o objeto Promise thatrejected e uma propriedade reason cujo valor é o que teria sido passado para uma função .catch(). Assim como acontece com os manipuladores de erros descritos anteriormente, se você chamar preventDefault() no objeto de evento de rejeição não tratada, ele será considerado manipulado e não causará uma mensagem de erro no console do desenvolvedor.

Muitas vezes não é necessário definir onerror ouonunhandledmanipuladores de rejeição, mas pode ser bastante útil como ummechanismo de telemetria se você quiser relatar erros do lado do cliente para oservidor (usando a função fetch() para fazer uma solicitação HTTP POST,por exemplo) para que você possa obter informações sobre erros inesperadosque acontecem nos navegadores de seus usuários.

15.1.8 O modelo de segurança da Web

O fato de as páginas da web poderem executar código JavaScript arbitrário em seu dispositivo pessoal tem implicações claras de segurança, e os fornecedores de navegadores trabalharam duro para equilibrar dois objetivos concorrentes:

Definindo APIs poderosas do lado do cliente para habilitar aplicativos da Web úteis
Impedindo que códigos mal-intencionados leiam ou alterem seus dados, comprometam sua privacidade, enganem você ou desperdicem seu tempo
As subseções a seguir fornecem uma visão geral rápida das restrições e problemas de segurança dos quais você, como programador JavaScript, deve estar ciente.

O QUE O JAVASCRIPT NÃO PODE FAZER A PRIMEIRA LINHA DE DEFESA DOS NAVEGADORES CONTRA CÓDIGOS MALICIOSOS É QUE ELES SIMPLESMENTE NÃO SUPORTAM DETERMINADOS RECURSOS. Por exemplo, o JavaScript do lado do cliente não fornece nenhuma maneira de gravar ou excluir arquivos arbitrários ou listar diretórios arbitrários no computador cliente. Isso significa que um programa JavaScript não pode excluir dados ou vírus de plantas. Da mesma forma, o JavaScript do lado do cliente não tem recursos de rede de uso geral. Um programa JavaScript do lado do cliente pode fazer solicitações HTTP (§15.11.1). E outro padrão, conhecido como WebSockets (§15.11.3), define uma API semelhante a um soquete para comunicação com servidores especializados. Mas nenhuma dessas APIs permite acesso não mediado à rede mais ampla. Clientes e servidores de Internet de uso geral não podem ser escritos em JavaScript do lado do cliente.

A POLÍTICA DE MESMA ORIGEM A POLÍTICA DE MESMA ORIGEM É UMA RESTRIÇÃO DE SEGURANÇA ABRANGENTE SOBRE COM O QUE O CÓDIGO JavaScript do conteúdo da Web pode interagir. Normalmente entra em jogo quando uma página da web inclui <iframe> elementos. Nesse caso, a política de mesma origem rege as interações do código JavaScript em um quadro com o conteúdo de outros quadros. Especificamente, um script pode ler somente as propriedades de janelas e documentos que têm a mesma origem que o documento que contém o script.

A origem de um documento é definida como o protocolo, o host e a porta da URL da qual o documento foi carregado. Os documentos carregados de diferentes servidores da Web têm origens diferentes. Documentos carregados por diferentes portas do mesmo host têm origens diferentes. E um documento carregado com o protocolo http: tem uma origem diferente do que aquele carregado com o protocolo https:, mesmo que venham do mesmo servidor web. Os navegadores normalmente tratam cada arquivo: URL como uma origem separada, o que significa que, se você estiver trabalhando em um programa que exibe mais de um documento do mesmo servidor, talvez não seja capaz de testá-lo localmente usando URLs file: e terá que executar um servidor web estático durante o desenvolvimento.

É importante entender que a origem do script em si não é relevante para a política de mesma origem: o que importa é a origem do documento no qual o script está incorporado. Suponha, por exemplo, que um script hospedado pelo host A seja incluído (usando a propriedade src de um <script> elemento) em uma página da Web servida pelo host B. A origem desse script é o host B, e o script tem acesso total ao conteúdo do documento que o contém. Se o documento contiver <iframe> um que

contém um segundo documento do host B, então o script também tem acesso total ao conteúdo desse segundo documento. Mas se o documento de nível superior contiver outro `<iframe>` que exiba um documento do host C (ou mesmo um do host A), a política de mesma origem entrará em vigor e impedirá que o script acesse esse documento aninhado.

A política de mesma origem também se aplica a solicitações HTTP com script (consulte §15.11.1). O código JavaScript pode fazer solicitações HTTP arbitrárias para o servidor web do qual o documento que o contém foi carregado, mas não permite que os scripts se comuniquem com outros servidores web (a menos que esses servidores web optem pelo CORS, como descrevemos a seguir).

A política de mesma origem apresenta problemas para sites grandes que usam vários subdomínios. Por exemplo, scripts com `originorders.example.com` podem precisar ler propriedades de documentos `onexample.com`. Para oferecer suporte a sites de vários domínios desse tipo, os scripts canalizam sua origem definindo `document.domain` como um sufixo de domínio. Portanto, um script com origem `https://orders.example.com` pode alterar sua origem para `https://example.com` definindo `document.domain` como "example.com". Mas esse script não pode definir `document.domain` como "orders.example", "ample.com" ou "com".

A segunda técnica para relaxar a política de mesma origem é o Cross-Origin Resource Sharing, ou CORS, que permite que os servidores decidam quais origens estão dispostos a atender. O CORS estende o HTTP com um novo cabeçalho de solicitação `Origin`: e um novo cabeçalho de resposta `Access-Control-Allow-Origin`. Ele permite que os servidores usem um cabeçalho para listar explicitamente as origens que podem solicitar um arquivo ou usar um curinga e

permitir que um arquivo seja solicitado por qualquer site. Os navegadores respeitam esses cabeçalhos CORSe não relaxam as restrições de mesma origem, a menos que eles representem.

SCRIPT ENTRE SITES Cross-site scripting, ou XSS, é um termo para uma categoria de problemas de segurança em que um invasor injeta tags ou scripts HTML em um site de destino. Os programadores JavaScript do lado do cliente devem estar cientes e se defender contra scripts entre sites.

Uma página da Web é vulnerável a scripts entre sites se gerar dinamicamente o conteúdo do documento e basear esse conteúdo em dados enviados pelo usuário sem primeiro "higienizar" esses dados removendo todas as tags HTML incorporadas dele. Como um exemplo trivial, considere a seguinte página da Web que usa JavaScript para cumprimentar o usuário pelo nome:

```
<script>let nome = novo URL(documento.  
URL).searchParams.get("nome");  
document.querySelector('h1').innerHTML = "Olá " + nome;  
</script>
```

Esse script de duas linhas extrai a entrada do parâmetro de consulta "name" da URL do documento. Em seguida, ele usa a API DOM para injetar uma string HTML na primeira <h1> tag do documento. Esta página deve ser invocada com uma URL como esta:

```
http://www.example.com/greet.html?name=David
```

Quando usado assim, ele exibe o texto "Olá David". Mas considere o que acontece quando ele é invocado com este parâmetro de consulta:

```
name=%3Cimg%20src=%22x.png%22%20onload=%22alert(%27hacked%27)%22/%3E
```

Quando os parâmetros de escape de URL são decodificados, esse URL faz com que o seguinte HTML seja injetado no documento:

```
Olá 
```

Depois que a imagem é carregada, a string de JavaScript no atributo onload é executada. A função global alert() exibe uma caixa de diálogo modal. Uma única caixa de diálogo é relativamente benigna, mas demonstra que a execução de código arbitrário é possível neste site porque exibe HTML deshigienizado.

Os ataques de script entre sites são assim chamados porque mais de um site está envolvido. O site B inclui um link especialmente criado (como o do exemplo anterior) para o site A. Se o site B conseguir convencer os usuários a clicar no link, eles serão levados ao site A, mas esse site agora estará executando o código do site B. Esse código pode desconfigurar a página ou causar mau funcionamento. Mais perigosamente, o código malicioso pode ler cookies armazenados pelo site A (talvez números de conta ou outras informações de identificação pessoal) e enviar esses dados de volta ao site B. O código injetado pode até rastrear as teclas digitadas pelo usuário e enviar esses dados de volta para o site B.

Em geral, a maneira de evitar ataques XSS é remover tags HTML de quaisquer dados não confiáveis antes de usá-los para criar conteúdo de documento dinâmico. Você pode corrigir o arquivo greet.html mostrado anteriormente substituindo caracteres HTML especiais na string de entrada não confiável por suas entidades HTML equivalentes:

```
nome = nome
    .replace(/&/g,
    '"&quot;').replace(/</g,
    '&lt;&quot;').replace(/>/g,
    '&quot;&gt;').replace(/"/g,
    '&quot;&quot;').replace(/\//g,
    '&quot;\//&quot;')
```

Outra abordagem para o problema do XSS é estruturar seus aplicativos da Web para que o conteúdo não confiável seja sempre exibido em um<iframe> com o atributo sandbox definido para desabilitar scripts e outros recursos.

O cross-site scripting é uma vulnerabilidade perniciosa cujas raízes se aprofundam na arquitetura da web. Vale a pena entender essa vulnerabilidade em profundidade, mas uma discussão mais aprofundada está além do escopo deste livro. Existem muitos recursos online para ajudá-lo a se defender contra scripts entre sites.

15.2 Eventos

Os programas JavaScript do lado do cliente usam um modelo assíncrono de programação orientada a eventos. Nesse estilo de programação, o navegador da web gera um evento sempre que algo interessante acontece com o documento ou navegador ou com algum elemento ou objeto associado a ele. Por exemplo, o navegador da Web gera um evento quando termina de carregar um documento, quando o usuário move o mouse sobre um hiperlink ou quando o usuário pressiona uma tecla no teclado. Se um aplicativo JavaScript se preocupa com um tipo específico de evento, ele pode registrar uma ou mais funções a serem invocadas quando eventos desse tipo ocorrerem. Observe que isso não é exclusivo da programação web: todos os aplicativos com gráficos

As interfaces de usuário são projetadas dessa maneira - elas ficam esperando para interagir (ou seja, esperam que os eventos ocorram) e então respondem.

No JavaScript do lado do cliente, os eventos podem ocorrer em qualquer elemento dentro de um documento HTML, e esse fato torna o modelo de evento dos navegadores da Web significativamente mais complexo do que o modelo de evento do Node. Começamos esta seção com algumas definições importantes que ajudam a explicar esse modelo de evento:

Tipo de evento

Essa cadeia de caracteres especifica que tipo de evento ocorreu. O tipo "mousemove", por exemplo, significa que o usuário moveu o mouse. O tipo "keydown" significa que o usuário pressionou uma tecla no teclado para baixo. E o tipo "load" significa que um documento (ou algum outro recurso) terminou de ser carregado da rede. Como o tipo de um evento é apenas uma cadeia de caracteres, às vezes é chamado de nome de evento e, de fato, usamos esse nome para identificar o tipo de evento de que estamos falando.

Destino do evento

Este é o objeto no qual o evento ocorreu ou com o qual o evento está associado. Quando falamos de um evento, devemos especificar tanto o tipo quanto o alvo. Um evento de carregamento em uma janela, por exemplo, ou um evento de clique em um <button> elemento. Os objetos Window, Document e Element são os destinos de eventos mais comuns em aplicativos JavaScript do lado do cliente, mas alguns eventos são acionados em outros tipos de objetos. Por exemplo, um objeto Worker (um tipo de thread, coberto §15.13) é um destino para eventos de "mensagem" que ocorrem quando o thread de trabalho envia uma mensagem para o thread principal.

manipulador de eventos ou ouvinte de eventos

Essa função manipula ou responde a um evento. A ²Aplicativos registre suas funções de manipulador de eventos com o navegador da Web, especificando um tipo de evento e um destino de evento. Quando um evento do tipo especificado ocorre no destino especificado, o navegador invoca a função de manipulador. Quando os manipuladores de eventos são invocados para um objeto, dizemos que o navegador "disparou", "disparou" ou "despachou" o evento. Há várias maneiras de registrar manipuladores de eventos, e os detalhes do registro e invocação do manipulador são explicados em §15.2.2 e §15.2.3.

objeto de evento

Esse objeto está associado a um evento específico e contém detalhes sobre esse evento. Os objetos de evento são passados como um argumento para a função de manipulador de eventos. Todos os objetos de evento têm uma propriedade `type` que especifica o tipo de evento e uma propriedade `target` que especifica o destino do evento. Cada tipo de evento define um conjunto de propriedades para seu objeto de evento associado. O objeto associado a um evento de mouse inclui as coordenadas do ponteiro do mouse, por exemplo, e o objeto associado a um evento de teclado contém detalhes sobre a tecla que foi pressionada e as teclas modificadoras que foram pressionadas. Muitos tipos de eventos definem apenas algumas propriedades padrão — como `type` e `target` — e não carregam muitas outras informações úteis. Para esses eventos, é a simples ocorrência do evento, não os detalhes do evento, que importa.

Propagação de eventos

Esse é o processo pelo qual o navegador decide em quais objetos acionar os manipuladores de eventos. Para eventos específicos de um único objeto, como o evento "load" no objeto `Window` ou um evento "message" em um objeto `Worker`, nenhuma propagação é necessária. Mas quando certos tipos de eventos ocorrem em elementos dentro do documento HTML, no entanto, eles se propagam ou "borbulham" na árvore de documentos. Se o usuário mover o mouse sobre um hiperlink, o evento `mousemove` será acionado primeiro no `<a>` elemento que define isso

link. Em seguida, ele é disparado nos elementos que o contêm: talvez um `<p>` elemento `<section>`, um elemento e o próprio objeto `Document`. Às vezes, é mais conveniente registrar um único manipulador de eventos em um documento ou outro elemento de contêiner do que registrar manipuladores em cada elemento individual em que você está interessado. Um manipulador de eventos pode interromper a propagação de um evento para que ele não continue a bolha e não acione manipuladores nos elementos que o contêm. Os manipuladores fazem isso invocando um método do objeto de evento. Noutra forma de propagação de eventos, conhecida como captura de eventos, os manipuladores especialmente registados em elementos de contentores têm a oportunidade de interceptar (ou "capturar") eventos antes de estes serem entregues ao seu alvo real. Borbulhamento e captura de eventos são recuperados em detalhes em §15.2.4.

Alguns eventos têm ações padrão associadas a eles. Quando um evento de clique ocorre em um hiperlink, por exemplo, a ação padrão é que o navegador siga o link e carregue uma nova página. Os manipuladores de eventos podem impedir essa ação padrão invocando um método do objeto de evento. Isso às vezes é chamado de "cancelamento" do evento e é abordado no §15.2.5.

15.2.1 Categorias de Eventos

O JavaScript do lado do cliente suporta um número tão grande de tipos de eventos que não há como este capítulo cobrir todos eles. No entanto, pode ser útil agrupar os eventos em algumas categorias gerais, para ilustrar o âmbito e a grande variedade de eventos suportados:

Eventos de entrada dependentes do dispositivo

Esses eventos estão diretamente ligados a um dispositivo de entrada específico, como o mouse ou o teclado. Eles incluem tipos de eventos como

"mousedown", "mousemove", "mouseup", "touchstart",
"touchmove", "touchend", "keydown" e "keyup".

Eventos de entrada independentes do dispositivo

Esses eventos de entrada não estão diretamente vinculados a um dispositivo de entrada específico. O evento "click", por exemplo, indica que um link ou botão (ou outro elemento do documento) foi ativado. Isso geralmente é feito por meio de um clique do mouse, mas também pode ser feito pelo teclado ou (em dispositivos sensíveis ao toque) com um toque. O evento "input" é uma alternativa independente de dispositivo ao evento "keydown" e suporta entrada de teclado, bem como alternativas como recortar e colar e métodos de entrada usados para scripts ideográficos. Os tipos de evento "pointerdown", "pointermove" e "pointerup" são alternativas independentes de dispositivo para eventos de mouse e toque. Eles funcionam para ponteiros do tipo mouse, para telas sensíveis ao toque e também para entrada no estilo caneta ou caneta.

Eventos da interface do usuário

Os eventos de interface do usuário são eventos de nível superior, geralmente em elementos de formulário HTML que definem uma interface do usuário para um aplicativo da web. Eles incluem o evento "focus" (quando um campo de entrada de texto ganha o foco do teclado), o evento "change" (quando o usuário altera o valor exibido por um elemento de formulário) e o evento "submit" (quando o usuário clica em um botão Submit em um formulário).

Eventos de mudança de estado

Alguns eventos não são acionados diretamente pela atividade do usuário, mas pela atividade da rede ou do navegador, e indicam algum tipo de ciclo de vida ou mudança relacionada ao estado. Os eventos "load" e "DOMContentLoaded" — disparados nos objetos Window e Document, respectivamente, no final do carregamento do documento — são provavelmente os mais usados desses eventos (consulte "Linha do tempo do JavaScript do lado do cliente"). Browsersdisparar eventos "online" e "offline" no objeto Window quando

Alterações na conectividade de rede. O mecanismo de gerenciamento de histórico do navegador (§15.10.4) dispara o evento "popstate" em resposta ao botão Voltar do navegador.

Eventos específicos da API

Várias APIs da Web definidas por HTML e related specifications incluem seus próprios tipos de evento. O HTML <video> e <audio> os elementos definem uma longa lista de tipos de eventos associados, como "aguardando", "reproduzindo", "buscando", "mudança de volume" e em breve, e você pode usá-los para personalizar a reprodução de mídia. De um modo geral, as APIs da plataforma da Web que são assíncronas e foram desenvolvidas antes de Promises serem adicionadas ao JavaScript são baseadas em eventos e definem eventos específicos da API. A API IndexedDB, por exemplo (§15.12.3), dispara eventos de "sucesso" e "erro" quando as solicitações de banco de dados são bem-sucedidas ou falham. E embora a nova API fetch() (§15.11.1) para fazer solicitações HTTP seja baseada em promessas, a API XMLHttpRequest que ela substitui define vários tipos de eventos específicos da API.

15.2.2 Registrando manipuladores de eventos

Há duas maneiras básicas de registrar manipuladores de eventos. A primeira, desde os primeiros dias da web, é definir uma propriedade no objeto ou elemento de documento que é o destino do evento. A segunda técnica (mais recente e mais geral) é passar o manipulador para o método addEventListener() do objeto ou elemento.

DEFININDO PROPRIEDADES DO MANIPULADOR DE EVENTOS A MANEIRA MAIS SIMPLES DE REGISTRAR UM MANIPULADOR DE EVENTOS É DEFININDO UMA PROPRIEDADE DO DESTINO DO EVENTO PARA A FUNÇÃO DE MANIPULADOR DE EVENTOS DESEJADA. Por convenção, as propriedades do manipulador de eventos têm nomes que consistem na palavra "on"

seguido pelo nome do evento: onclick, onchange, onload, onmouseover e assim por diante. Observe que esses nomes de propriedade diferenciam maiúsculas de minúsculas e são escritos em letras minúsculas, mesmo quando o tipo de evento (como "mousedown") consiste em várias palavras. O código a seguir inclui dois registros de manipulador de eventos desse tipo:

Defina a propriedade onload do objeto Window como uma função.// A função é o manipulador de eventos: ela é invocada quando o documento carrega.window.onload = function() {

Procure um <form> elementlet form = document.querySelector("form#shipping");// Registre uma função de manipulador de eventos no formulário que será invocado antes do envio do formulário. Suponha que isFormValid() esteja definido em outro lugar.
form.onsubmit = function(event) { // Quando o usuário envia o formulário
 if (!isFormValid(this)) { *verificar se o formulário As entradas são válidas*
 event.preventDefault(); *e se não, evitar envio de formulários.*
 };};;

A desvantagem das propriedades do manipulador de eventos é que elas são projetadas em torno da suposição de que os destinos de eventos terão no máximo um manipulador para cada tipo de evento. Geralmente, é melhor registrar manipuladores de eventos usandoaddEventListener() porque essa técnica não substitui nenhum manipulador registrado anteriormente.

CONFIGURAÇÃO DE ATRIBUTOS DO MANIPULADOR DE EVENTOS

As propriedades do manipulador de eventos dos elementos do documento também podem ser definidas

diretamente no arquivo HTML como atributos na tag HTML correspondente. (Os manipuladores que seriam registrados no elemento Window withJavaScript podem ser definidos com atributos na <body> marca inHTML.) Essa técnica geralmente é desaprovada no desenvolvimento web moderno, mas é possível e está documentada aqui porque você ainda pode vê-la no código existente.

Ao definir um manipulador de eventos como um atributo HTML, o attributevalue deve ser uma string de código JavaScript. Esse código deve ser o corpo da função do manipulador de eventos, não uma declaração de função completa. Ou seja, o código do manipulador de eventos HTML não deve ser cercado por chaves e prefixado com a palavra-chave function. Por exemplo:

```
<button onclick="console.log('Obrigado');">Por  
favorClique</button>
```

Se um atributo do manipulador de eventos HTML contiver várias instruções JavaScript, lembre-se de separar essas instruções com ponto-e-vírgula ou quebrar o valor do atributo em várias linhas.

Quando você especifica uma string de código JavaScript como o valor de um atributo de manipulador de eventos HTML, o navegador converte sua string em uma função que funciona mais ou menos assim:

```
função(evento) {  
    with(documento) {  
        with(this.form || {}) {  
            com(isso) {  
                /* seu código aqui */}}}
```

}

O argumento event significa que o código do manipulador pode se referir ao objeto de evento atual como evento. As instruções with significam que thecode do seu manipulador pode se referir às propriedades do objeto de destino, thecontaining <form> (se houver) e o objeto Document recipienting diretamente, como se fossem variáveis no escopo. A instrução with é proibida no modo estrito (§5.6.3), mas o código JavaScript em atributos HTML nunca é estrito. Os manipuladores de eventos definidos dessa maneira são executados em um ambiente no qual variáveis inesperadas são definidas. Isso pode ser uma fonte de bugs confusos e é um bom motivo para evitar escrever manipuladores de eventos em HTML.

ADDEVENTLISTENER() Qualquer objeto que possa ser um destino de evento, incluindo os objetos Window e Document e todos os elementos do documento, define um método chamado addEventListener() que você pode usar para registrar um manipulador de eventos para esse destino. addEventListener() recebe três argumentos. O primeiro é o tipo de evento para o qual o manipulador está sendo registrado. O tipo de evento (ou nome) é uma cadeia de caracteres que não inclui o prefixo "on" usado ao definir as propriedades do manipulador de eventos. O segundo argumento to addEventListener() é a função que deve ser invocada quando o tipo especificado de evento ocorre. O terceiro argumento é opcional e é explicado abaixo.

O código a seguir registra dois manipuladores para o evento "click" em um <button> elemento. Observe as diferenças entre as duas técnicas usadas:

```
<button id="mybutton">Clique em mim</button><script>let  
b = document.querySelector("#mybutton"); b.onclick =  
function() { console.log("Obrigado por clicar em mim!");}  
; b.addEventListener("clique", () => {  
console.log("Obrigado novamente!"); });</script>
```

Chamar addEventListener() com "click" como seu primeiro argumento não afeta o valor da propriedade onclick. Neste código, um clique de botão registrará duas mensagens no console do desenvolvedor. E se chamássemos addEventListener() primeiro e depois definissemos onclick, ainda registráramos duas mensagens, apenas na ordem oposta. Mais importante, você pode chamar addEventListener() várias vezes para registrar mais de uma função de manipulador para o mesmo tipo de evento no mesmo objeto. Quando um evento ocorre em um objeto, todos os manipuladores registrados para esse tipo de evento são invocados na ordem em que foram registrados. Invocar addEventListener() mais de uma vez no mesmo objeto com os mesmos argumentos não tem efeito — a função handler permanece registrada apenas uma vez e a invocação repetida não altera a ordem na qual os handlers são invocados.

addEventListener() é emparelhado com um método removeEventListener() que espera os mesmos dois argumentos (mais um terceiro opcional), mas remove uma função de manipulador de eventos de um objeto em vez de adicioná-la. Muitas vezes, é útil registrar temporariamente um manipulador de eventos e removê-lo logo em seguida. Por exemplo, quando você obtém um evento "mousedown", você pode registrar manipuladores de eventos temporários para eventos "mousemove" e "mouseup" para que você possa ver se o usuário arrasta o mouse.

Em seguida, você cancelaria o registro desses manipuladores quando o evento "mouseup" chegasse. Em tal situação, o código de remoção do manipulador de eventos pode ter esta aparência:

```
document.removeEventListener("mousemove", handleMouseMove);
document.removeEventListener("mouseup", handleMouseUp);
```

O terceiro argumento opcional para addEventListener() é um valor ou objeto booleano. Se você passar true, sua função de manipulador será registrada como um manipulador de eventos de captura e será invocada em uma fase diferente do envio de eventos. Abordaremos a captura de eventos em §15.2.4. Se você passar um terceiro argumento de true ao registrar um ouvinte de eventos, também deverá passar true como o terceiro argumento toremoveEventListener() se quiser remover o manipulador.

Registrar um manipulador de eventos de captura é apenas uma das três opções que addEventListener() suporta e, em vez de passar um único valor booleano, você também pode passar um objeto que especifica explicitamente as opções desejadas:

```
document.addEventListener("click", handleClick, {
  captura: verdadeiro, uma vez:
  verdadeiro, passivo:
  verdadeiro});
```

Se o objeto Options tiver uma propriedade de captura definida como true, o manipulador de eventos será registrado como um manipulador de captura. Se essa propriedade for falsa ou omitida, o manipulador não será capturado.

Se o objeto Options tiver uma propriedade once definida como true, o evento

será removido automaticamente depois de ser acionado uma vez. Se thisproperty for false ou for omitido, o manipulador nunca será removido automaticamente.

Se o objeto Options tiver uma propriedade passiva definida como true, isso indicará que o manipulador de eventos nunca chamará preventDefault() para cancelar a ação padrão ([consulte §15.2.5](#)). Isso é particularmente importante para eventos de toque em dispositivos móveis — se os manipuladores de eventos para eventos "touchmove" puderem impedir a ação de rolagem padrão do navegador, o navegador não poderá implementar a rolagem suave. Essa propriedade passiva fornece uma maneira de registrar um manipulador de eventos potencialmente disruptivo desse tipo, mas permite que o navegador da Web saiba que ele pode iniciar com segurança seu comportamento padrão, como rolagem, enquanto o manipulador de eventos está em execução. A rolagem suave é tão importante para uma boa experiência do usuário que o Firefox e Chrome tornam os eventos "touchmove" e "roda do mouse" passivos por padrão. Portanto, se você realmente deseja registrar um manipulador que chama preventDefault() para um desses eventos, defina explicitamente a propriedade passiva como false.

Você também pode passar um objeto Options para removeEventListener(), mas a propriedade capture é a única que é relevante. Não há necessidade de especificar uma vez ou passivo ao remover um ouvinte, essas propriedades são ignoradas.

15.2.3 Invocação do manipulador de eventos

Depois de registrar um manipulador de eventos, o navegador da Web o invocará automaticamente quando um evento do tipo especificado ocorrer no objeto especificado. Esta seção descreve a invocação do manipulador de eventos em

detail, explicando os argumentos do manipulador de eventos, o contexto de invocação (o valor this) e o significado do valor retornado de um manipulador de eventos.

ARGUMENTO DO MANIPULADOR DE EVENTOS

manipuladores de ventilação são invocados com um objeto Event como seu argumento único. As propriedades do objeto Event fornecem detalhes sobre o evento:

tipo

O tipo do evento que ocorreu.

alvo

O objeto no qual o evento ocorreu.

alvo atual

Para eventos que se propagam, essa propriedade é o objeto no qual o manipulador de eventos atual foi registrado.

timestamp

Um carimbo de data/hora (em milissegundos) que representa quando o evento ocorreu, mas que não representa um tempo absoluto. Você pode determinar o tempo decorrido entre dois eventos subtraindo o carimbo de data/hora do primeiro evento do carimbo de data/hora do segundo.

é confiável

Essa propriedade será true se o evento tiver sido despachado pelo próprio navegador da Web e false se o evento tiver sido despachado pelo JavaScriptcode.

Tipos específicos de eventos têm propriedades adicionais. Mouse e ponteiro

eventos, por exemplo, têm propriedades clientX e clientY que especificam as coordenadas da janela nas quais o evento ocorreu.

CONTEXTO DO MANIPULADOR DE EVENTOS Quando você registra um manipulador de eventos definindo uma propriedade, parece que você está definindo um novo método no objeto de destino:

```
target.onclick = function() { /* código do manipulador */ };
```

Não é surpreendente, portanto, que os manipuladores de eventos sejam invocados como métodos do objeto no qual eles são definidos. Ou seja, dentro do corpo de um manipulador de eventos, a palavra-chave this refere-se ao objeto no qual o manipulador de eventos foi registrado.

Os manipuladores são invocados com o destino como seu valor this, mesmo quando registrados usando addEventListener(). No entanto, isso não funciona para manipuladores definidos como funções de seta: as funções de seta sempre têm o mesmo valor que o escopo no qual são definidas.

VALOR DE RETORNO DO MANIPULADOR No JavaScript moderno, os manipuladores de eventos não devem retornar nada. Você pode ver manipuladores de eventos que retornam valores em código mais antigo, e o valor de retorno normalmente é um sinal para o navegador de que ele não deve executar a ação padrão associada ao evento. Se o manipulador onclick de um botão Submit em um formulário retornar false, por exemplo, o navegador da Web não enviará o formulário (geralmente porque o manipulador de eventos determinou que a entrada do usuário falha na validação do lado do cliente).

A maneira padrão e preferida de evitar que o navegador

executar uma ação padrão é chamar o método preventDefault()(§15.2.5) no objeto Event.

ORDEM DE INVOCAÇÃO Um destino de evento pode ter mais de um manipulador de eventos registrado para um tipo específico de evento. Quando ocorre um evento desse tipo, o navegador invoca todos os manipuladores na ordem em que foram registrados. Curiosamente, isso é verdade mesmo se você misturar manipuladores de eventos registrados com um manipulador de eventos registrado em uma propriedade de objeto como onclick.

15.2.4 Propagação de eventos

Quando o destino de um evento é o objeto Window ou algum outro objeto autônomo, o navegador responde a um evento simplesmente invocando os manipuladores apropriados nesse objeto. Quando o destino do evento é um documento ou elemento de documento, no entanto, a situação é mais complicada.

Depois que os manipuladores de eventos registrados no elemento de destino são invocados, a maioria dos eventos "borbulha" na árvore DOM. Os manipuladores de eventos do pai do destino são chamados. Em seguida, os manipuladores registrados no avô do alvo são invocados. Isso continua até o objeto Document e, em seguida, além do objeto Window. A propagação de eventos fornece uma alternativa para registrar manipuladores em muitos elementos de documento individuais: em vez disso, você pode registrar um único manipulador em um elemento commonancestor e manipular eventos lá. Você pode registrar um manipulador de "alteração" em um <form> elemento, por exemplo, em vez de registrar um manipulador de "alteração" para cada elemento no formulário.

A maioria dos eventos que ocorrem em elementos de documento borbulham. Exceções notáveis são os eventos "foco", "desfoque" e "rolagem". O evento "load" nos elementos do documento borbulha, mas para de borbulhar no objeto Document e não se propaga para o objeto Window. (Os manipuladores de eventos "load" do objeto Window são acionados somente quando todo o documento é carregado.)

O borbulhar de eventos é a terceira "fase" da propagação de eventos. A invocação dos manipuladores de eventos do próprio objeto de destino é a segunda fase. A primeira fase, que ocorre antes mesmo de os manipuladores de destino serem invocados, é chamada de fase de "captura". Lembre-se de queaddEventListen()r() recebe um terceiro argumento opcional. Se esse argumento for true, ou {capture:true}, o manipulador de eventos será registrado como um manipulador de eventos de captura para invocação durante essa primeira fase da propagação do evento. A fase de captura da propagação do evento é como a fase borbulhante ao contrário. Os manipuladores de captura do objeto theWindow são invocados primeiro, depois os manipuladores de captura do objeto theDocument, depois do objeto body e assim por diante na DOMtree até que os manipuladores de eventos de captura do pai do destino do evento sejam invocados. Os manipuladores de eventos de captura registrados no próprio destino do evento não são invocados.

A captura de eventos oferece uma oportunidade de espiar os eventos antes que eles sejam entregues ao seu destino. Um manipulador de eventos de captura pode ser usado para depuração ou pode ser usado junto com a técnica de cancelamento de eventos descrita na próxima seção para filtrar eventos para que os manipuladores de eventos de destino nunca sejam realmente invocados. Um uso comum para captura de eventos é lidar com arrastar o mouse, em que os eventos de movimento do mouse precisam ser manipulados pelo objeto que está sendo arrastado, não pelos elementos do documento

sobre o qual é arrastado.

15.2.5 Cancelamento de Evento

Os navegadores respondem a muitos eventos do usuário, mesmo que seu código não: quando o usuário clica com o mouse em um hiperlink, o navegador segue o link. Se um elemento de entrada de texto HTML tiver o foco do teclado e o usuário digitar uma tecla, o navegador inserirá a entrada do usuário. Se o usuário mover o dedo por um dispositivo com tela sensível ao toque, o navegador rolará. Se você registrar um manipulador de eventos para eventos como esses, poderá impedir que o navegador execute sua ação padrão invocando o método preventDefault() do objeto de evento. (A menos que você tenha registrado o manipulador com a opção passiva, o que torna preventDefault() ineficaz.)

Cancelar a ação padrão associada a um evento é apenas um tipo de cancelamento de evento. Também podemos cancelar a propagação de eventos chamando o método stopPropagation() do objeto de evento. Se houver outros manipuladores definidos no mesmo objeto, o restante desses manipuladores ainda será invocado, mas nenhum manipulador de eventos em qualquer outro objeto será invocado depois que stopPropagation() for chamado. stopPropagation() funciona durante a fase de captura, no próprio destino do evento e durante a fase de borbulhamento. stopImmediatePropagation() funciona como stopPropagation(), mas também impede a invocação de quaisquer manipuladores de eventos subsequentes registrados no mesmo objeto.

15.2.6 Despachando eventos personalizados

A API de eventos do JavaScript do lado do cliente é relativamente poderosa e você pode usá-la para definir e despachar seus próprios eventos. Suponha, por exemplo, que seu programa precise executar periodicamente um cálculo longo ou fazer uma solicitação de rede e que, enquanto essa operação estiver gastando, outras operações não serão possíveis. Você deseja que o usuário saiba sobre isso exibindo "spinners" para indicar que o aplicativo está ocupado. Mas o módulo que está ocupado não deve precisar saber onde os spinners devem ser exibidos. Em vez disso, esse módulo pode apenas despachar um evento para anunciar que está ocupado e, em seguida, despachar outro evento quando não estiver mais ocupado. Em seguida, o módulo de interface do usuário pode registrar manipuladores de eventos para esses eventos e executar as ações de interface do usuário apropriadas para notificar o usuário.

Se um objeto JavaScript tiver um método `addEventListener()`, ele será um "destino de evento" e isso significa que ele também terá um método `dispatchEvent()`. Você pode criar seu próprio objeto de evento com o construtor `CustomEvent()` e passá-lo para `dispatchEvent()`. O primeiro argumento para `CustomEvent()` é uma string que especifica o tipo do seu evento, e o segundo argumento é um objeto que especifica as propriedades do objeto de evento. Defina a propriedade `detail` deste objeto como uma cadeia de caracteres, objeto ou outro valor que represente o conteúdo do seu evento. Se você planeja despachar seu evento em um elemento de documento e deseja que ele borbulhe na árvore de documentos, adicione `bubbles:true` ao segundo argumento:

Despache um evento personalizado para que a interface do usuário saiba que estamos ocupados
`document.dispatchEvent(new CustomEvent("busy", { detail: true}));`

Executar uma operação de rede

```
fetch(url)
  .then(handleNetworkResponse)
  .catch(handleNetworkError)
  .finally(() => {
  Depois que a solicitação de rede for bem-sucedida ou
falhar, despache
  outro evento para que a interface do usuário saiba que
não estamos mais ocupados.
  document.dispatchEvent(new CustomEvent("ocupado",
  {detalhe: falso }));
})

Em outro lugar, em seu programa, você pode registrar um
manipulador para eventos "ocupados" // e usá-lo para
mostrar ou ocultar o controle giratório para permitir que o
usuário saiba.document.addEventListener("ocupado", (e) => {

  if (e.detalhe) {
    showSpinner();
  } else {
    hideSpinner();
  }
});
```

15.3 Documentos de script

O JavaScript do lado do cliente existe para transformar documentos HTML estáticos em aplicativos da Web interativos. Portanto, criar scripts para o conteúdo das páginas da web é realmente o objetivo central do JavaScript.

Cada objeto Window tem uma propriedade de documento que se refere a um objeto Document. O objeto Document representa o conteúdo da janela e é o assunto desta seção. No entanto, o objeto Document não está sozinho. É o objeto central no DOM para representar e manipular o conteúdo do documento.

O DOM foi introduzido em §15.1.2. Esta seção explica a API em detalhes. Abrange:

- Como consultar ou selecionar elementos individuais de um documento. Como percorrer um documento e como localizar os ancestrais, irmãos e descendentes de qualquer elemento do documento.
- Como consultar e definir os atributos dos elementos do documento. Como consultar, definir e modificar o conteúdo de um documento. Como modificar a estrutura de um documento criando, inserindo e excluindo nós.

15.3.1 Selecionando elementos do documento

Os programas JavaScript do lado do cliente geralmente precisam manipular um ou mais elementos dentro do documento. A propriedade global document refere-se ao objeto Document, e o objeto Document tem propriedades head e body que se referem aos objetos Element para as <head> tags e ,<body> respectivamente. Mas um programa que deseja manipular um elemento embutido mais profundamente no documento deve de alguma forma obter ou selecionar os objetos Element que se referem a esses elementos do documento.

SELECIONANDO ELEMENTOS COM SELETORES CSS AS folhas de estilo CSS têm uma sintaxe muito poderosa, conhecida como seletores, para descrever elementos ou conjuntos de elementos dentro de um documento. Os métodos DOMs querySelector() e querySelectorAll() nos permitem encontrar o elemento ou elementos dentro de um documento que correspondem a um seletor CSS especificado. Antes de abordarmos os métodos, começaremos com um tutorial rápido sobre a sintaxe do seletor CSS.

Os seletores CSS podem descrever elementos pelo nome da tag, o valor de seu atributo `id`, ou as palavras em seu atributo de classe:

<code>Div</code>	Qualquer <div> elemento
<code>#nav</code>	O elemento com <code>id="nav"</code>
<code>.aviso</code>	Qualquer elemento com "warning" em seu
<code>atributo class</code>	

O caractere `#` é usado para corresponder com base no atributo `id` e o `.` é usado para corresponder com base no atributo `class`. Os elementos também podem ser selecionados com base em valores de atributo mais gerais:

<code>p[lang="fr"]</code>	Um parágrafo escrito em francês: <p
<code>lang="fr">*</code>	
<code>[nome="x"]</code>	Qualquer elemento com um <code>name="x"</code>

Observe que esses exemplos combinam um seletor de nome de tag (ou o curinga `*` tagname) com um seletor de atributo. Combinações mais complexas também são possíveis:

<code>span.fatal.error</code>	Qualquer um com "fatal" e
<code>"error" em seu classspan[lang="fr"].warning</code>	// Qualquer em francês com class "warning"

Os seletores também podem especificar a estrutura do documento:

<code>#log</code>	vão	Qualquer descendente do
elemento com		
<code>id="log"</code>	<code>#log>span</code>	Qualquer filho do elemento
<code>body>h1:first-child</code>		
<code>img + p.caption</code>		O primeiro <h1> filho do <body>
		A <p> com a classe "legenda"
		imediatamente após um h2 ~ p// Qualquer <p> um que
		siga um <h2> andis um irmão dele

Se dois seletores estiverem separados por uma vírgula, isso significa que selecionaremos os elementos que correspondem a qualquer um dos seletores:

```
button, input[type="button"] // Todos <button> e  
<input type="button"> elementos
```

Como você pode ver, os seletores CSS nos permitem nos referir a elementos dentro de um documento por tipo, ID, classe, atributos e posição dentro do documento. O método querySelector() usa uma string de seletor CSS como argumento e retorna o primeiro elemento correspondente no documento que encontra ou retorna null se nenhum corresponder:

```
Encontre o elemento de documento para a tag HTML com  
attribute id="spinner" let spinner =  
document.querySelector("#spinner");
```

querySelectorAll() é semelhante, mas retorna todos os elementos correspondentes no documento em vez de apenas retornar o primeiro:

```
Localize todos os objetos Element para <h1>, <h2>, e <h3>  
tags let titles = document.querySelectorAll("h1, h2, h3");
```

O valor de retorno de querySelectorAll() não é um array de objetos Element. Em vez disso, é um objeto semelhante a uma matriz conhecido como a NodeList. Os objetos NodeList têm uma propriedade length e podem ser indexados como matrizes, para que você possa fazer um loop sobre eles com um for loop tradicional. NodeLists também são iteráveis, então você também pode usá-los com for/of loops. Se você deseja converter um NodeList em um array verdadeiro, basta passá-lo para Array.from().

O NodeList retornado por querySelectorAll() terá um

`length` definida como 0 se não houver nenhum elemento no documento que corresponda ao seletor especificado.

`querySelector()` e `querySelectorAll()` são implementados pela classe `Element`, bem como pela classe `Document`. Quando invocado em um elemento, esses métodos retornarão apenas elementos que são descendentes desse elemento.

Observe que o CSS define `::first-line` e `::first-letter` ~~pseudoelements~~. Em CSS, eles correspondem a partes de nós de texto em vez de elementos reais. Eles não corresponderão se usados com `querySelectorAll()` ou `querySelector()`. Além disso, muitos navegadores se recusarão a retornar correspondências para as pseudoclasses `:link` e `:visited`, pois isso pode expor informações sobre o histórico de navegação do usuário.

Outro método de seleção de elementos baseado em CSS é `closest()`. Esse método é definido pela classe `Element` e usa um seletor como seu único argumento. Se o seletor corresponder ao elemento em que é invocado, ele retornará esse elemento. Caso contrário, ele retornará o elemento ancestral mais próximo que o seletor corresponder, ou retornará nulo se nenhum corresponder. Em certo sentido, `closest()` é o oposto de `querySelector()`: `closest()` começa em um elemento e procura uma correspondência acima dele na árvore, enquanto `querySelector()` começa com um elemento e procura uma correspondência abaixo dele na árvore. `closest()` pode ser útil quando você registrou um manipulador de eventos em um nível alto na árvore de documentos. Se você estiver manipulando um evento de "clique", por exemplo, talvez queira saber se é um clique em um hiperlink. O objeto de evento informará o que o objeto

target era, mas esse target pode ser o texto dentro de um link, em vez da própria tag do hiperlink `<a>`. Seu manipulador de eventos pode procurar o hiperlink que contém mais próximo como este:

```
Encontre a tag delimitadora mais próxima <a> que  
tenha um hrefattribute.let hyperlink =  
event.target.closest("a[href]");
```

Aqui está outra maneira de usar closest():

```
Retorna true se o elemento e estiver dentro de uma  
função de elemento de lista HTML insideList(e) {  
  
return e.closest("ul,ol,dl") !== null;}
```

O método relacionado matches() não retorna ancestrais ou descendentes: ele simplesmente testa se um elemento é correspondido por um CSSselector e retorna true se for o caso e false caso contrário:

```
Retorne true se e for um elemento de  
cabeçalho HTMLFUNCTION isHeading(e) {  
return e.matches("h1,h2,h3,h4,h5,h6");}
```

OUTROS MÉTODOS DE SELEÇÃO DE ELEMENTOS Em adição a `querySelector()` e `querySelectorAll()`, o DOM também define vários métodos de seleção de elementos mais antigos que estão mais ou menos obsoletos agora. No entanto, você ainda pode ver alguns desses métodos (especialmente `getElementById()`) em uso:

```
Procure um elemento por id. O argumento é apenas o  
id,without// o prefixo do seletor CSS #. Semelhante a
```

```
document.querySelector("#sect1")let sect1 =  
document.getElementById("sect1");
```

Procure todos os elementos (como caixas de seleção de formulário) que tenham aname="color"// atributo. Semelhante a document.querySelectorAll('[name="color"]'); let cores = document.getElementsByName("cor");*

Procure todos os <h1> elementos no documento.// Semelhante a document.querySelectorAll("h1")let headings = document.getElementsByTagName("h1");

getElementsByTagName() também é definido em elements.// Obtém todos os <h2> elementos dentro do elemento sect1 element.let subheads = sect1.getElementsByTagName("h2");

Pesquise todos os elementos que têm a classe "dica de ferramenta". Semelhante a document.querySelectorAll(".tooltip")let tooltips = document.getElementsByClassName("tooltip");

Procure todos os descendentes de sect1 que tenham a classe "sidebar"// Semelhante a sect1.querySelectorAll(".sidebar")let sidebars = sect1.getElementsByClassName("sidebar");

Como querySelectorAll(), os métodos neste código retornam aNodeList (exceto getElementById(), que retorna um objeto singleElement). Ao contrário de querySelectorAll(), no entanto, as NodeLists retornadas por esses métodos de seleção mais antigos são "dinâmicas", o que significa que o comprimento e o conteúdo da lista podem ser alterados se o conteúdo ou a estrutura do documento forem alterados.

ELEMENTOS PRÉ-SELECIONADOSPara motivos históricos, a classe Document define propriedades de atalho para acessar certos tipos de nós. As propriedades images, forms e links, por exemplo, fornecem acesso fácil aos <form>elementos , ,and <a> (mas apenas <a> tags que têm um atributo href) de um

documento. Essas propriedades referem-se a objetos `HTMLCollection`, que são muito parecidos com objetos `NodeList`, mas também podem ser indexados por ID ou nome do elemento. Com a propriedade `document.forms`, por exemplo, você pode acessar a tag `<form id="address">` como:

```
documento.formulários.endereço;
```

Uma API ainda mais desatualizada para selecionar elementos é a propriedade `document.all`, que é como uma `HTMLCollection` para todos os elementos do documento. `document.all` está obsoleto e você não deve mais usá-lo.

15.3.2 Estrutura e Travessia do Documento

Depois de selecionar um elemento de um documento, às vezes você precisa encontrar partes estruturalmente relacionadas (pai, irmãos, filhos) do documento. Quando estamos interessados principalmente nos elementos de um documento em vez do texto dentro deles (e no espaço em branco entre eles, que também é texto), há uma API transversal que nos permite tratar um documento como uma árvore de objetos `Element`, ignorando os nós de texto que também fazem parte do documento. Essa API transversal não envolve nenhum método; é simplesmente um conjunto de propriedades em objetos `Element` que nos permite nos referir ao pai, filhos e irmãos de um determinado elemento:

Pai

Essa propriedade de um elemento refere-se ao pai do elemento, que será outro objeto `Element` ou `Document`.

crianças

Essa `NodeList` contém os filhos `Element` de um elemento, mas

exclui filhos que não são de elemento, como nós de texto (e nós de comentário).

childElementCount

O número de filhos do Element. Retorna o mesmo valor que children.length.

firstElementChild, lastElementChild

Essas propriedades referem-se ao primeiro e ao último filho Element de um elemento. Eles serão nulos se o Elemento não tiver filhos de Elemento.

próximoElementSibling, anteriorElementSibling

Essas propriedades referem-se aos elementos irmãos imediatamente antes ou imediatamente após um elemento, ou nulos se não houver tal irmão.

Usando essas propriedades de elemento, o segundo elemento filho do elemento firstchild do documento pode ser referenciado com qualquer uma destas expressões:

```
document.children[0].children[1]document.firstChild.firstChild.nextElementSibling
```

(Em um documento HTML padrão, ambas as expressões se referem à<body> marca do documento.)

Aqui estão duas funções que demonstram como você pode usar essas propriedades para fazer recursivamente uma passagem em profundidade de um documento invocando uma função especificada para cada elemento no documento:

Percorra recursivamente o Documento ou Elemento e, invocando

```

a função// f em e e em cada um de seus
descendentesfunction traverse(e, f) {

    f(e);                                Invocar f() em e
    for(let filho de e.children) {          Iterar sobre o
        crianças
        travessia(criança, f);             E recursa em cada um
    }  

    Um
}

função traverse2(e, f) {
    f(e);                                Invocar f() em e
    let criança = e.firstElementChild;     Iterar os filhos
    estilo de lista vinculada
    while(filho !== null) {
        travessia2(criança, f);           E recursar
        criança = criança.nextElementSibling;}}

```

DOCUMENTOS COMO ÁRVORES DE NÓS Se você quiser percorrer um documento ou alguma parte de um documento e não quiser ignorar os nós de texto, poderá usar um conjunto diferente de propriedades definidas em todos os objetos de nó. Isso permitirá que você veja Elementos, Nós de texto e até Nós de comentário (que representam comentários HTML no documento).

Todos os objetos Node definem as seguintes propriedades:

Pai

O nó que é o pai deste ou nulo para nós como o objeto Document que não têm pai.

nós filhos

Um NodeList somente leitura que contém todos os filhos (não apenas filhos Element) do nó.

primeiroFilho, últimoFilho

O primeiro e o último nó filho de um nó ou null se o nó tiver nochildren.

próximoIrmão, anteriorIrmão

Os nós irmãos seguintes e anteriores de um nó. Essas propriedades conectam nós em uma lista duplamente vinculada.

tipo de nó

Um número que especifica que tipo de nó é esse. Os nós do documento têm valor 9. Os nós de elemento têm valor 1. Os nós de texto têm valor 3. Os nós de comentário têm o valor 8.

valor do nó

O conteúdo textual de um nó Texto ou Comentário.

nome do nó

O nome da marca HTML de um elemento, convertido em maiúsculas.

Usando essas propriedades Node, o segundo nó filho do primeiro filho do Documento pode ser referenciado com expressões como estas:

```
document.childNodes[0].childNodes[1]document.firstChild.firstChild.nextSibling
```

Suponha que o documento em questão seja o seguinte:

```
<html><head><title>Teste</title></head><body>Hello  
World!</body></html>
```

Então, o segundo filho do primeiro filho é o <body> elemento. Ele tem anodeType de 1 e um nodeName de "BODY".

Observe, no entanto, que essa API é extremamente sensível a variações no texto do documento. Se o documento for modificado inserindo uma única nova linha entre o <html> e a <head> tag, por exemplo, o nó Text que representa essa nova linha se tornará o primeiro filho do primeiro filho, e o segundo filho será o <head> elemento em vez do <body> elemento.

Para demonstrar essa API de passagem baseada em nó, aqui está uma função que retorna todo o texto dentro de um elemento ou documento:

Retorna o conteúdo de texto simples do elemento e, recursando os elementos enchild.// Este método funciona como a propriedade textContentfunction textContent(e) {

```
    seja s = "";                      Acumule o texto
    aqui
    for(let criança = e.primeiraCriança; criança !== null;
        criança = criança.próximoIrmão) {
        let tipo = child.nodeType;
        if (tipo === 3) {              Se for um texto
            nodo
            s += filho.nodeValor;     Adicione o texto
            conteúdo à nossa string.
        } else if (tipo === 1) {        E se for um
            Nó de elemento
            s += textContent(filho);  então recursa.
        }
    } returnar
    s;}
```

Essa função é apenas uma demonstração - na prática, você simplesmente

escreva e.textContent para obter o conteúdo textual do elemento e.

15.3.3 Atributos

Os elementos HTML consistem em um nome de tag e um conjunto de pares de nome/valor conhecidos como atributos. O <a> elemento que define um hiperlink, por exemplo, usa o valor de seu atributo href como o destino do link.

A classe Element define os métodos gerais getAttribute(), setAttribute(), hasAttribute() e removeAttribute() para consultar, definir, testar e remover os atributos de um elemento. Mas os valores de atributo dos elementos HTML (para todos os atributos padrão dos elementos HTML padrão) estão disponíveis como propriedades dos objetos HTMLElement que representam esses elementos, e geralmente é muito mais fácil trabalhar com eles como propriedades JavaScript do que chamar getAttribute() e métodos relacionados.

ATRIBUTOS HTML COMO PROPRIEDADES DE ELEMENTOS objetos de elemento que representam os elementos de um documento HTML geralmente definem propriedades de leitura/gravação que espelham os atributos HTML dos elementos. O elemento define propriedades para os atributos HTML universais, como id, title, lang e dir, e as propriedades do manipulador de eventos, como onclick. Os subtipos específicos do elemento definem atributos específicos para esses elementos. Para consultar a URL de uma imagem, por exemplo, você pode usar a propriedade src do HTMLElement que representa o elemento:

```
let imagem = document.querySelector("#main_image"); let url  
= image.src;// O atributo src é a URL do imageimage.id ===  
"main_image");// => true; nós olhamos para cima a imagem por id
```

Da mesma forma, você pode definir os atributos de envio de formulário de um <form>elemento com código como este:

```
Reimagine o elemento que você está tratando formulário);  
"https://www.example.com/submit"; // Defina a URL para  
enviá-la.f.method = "POST"; // Defina o tipo de solicitação  
HTTP.
```

Para alguns elementos, como o <input> elemento, alguns nomes de atributos HTML são mapeados para propriedades com nomes diferentes. O atributo HTML value de um <input>, por exemplo, é espelhado pela propriedade JavaScript defaultValue. A propriedade de valor JavaScript do <input> elemento contém a entrada atual do usuário, mas as alterações na propriedade value não afetam a propriedade defaultValue nem o atributo value.

Os atributos HTML não diferenciam maiúsculas de minúsculas, mas os nomes de propriedades JavaScript são. Para converter um nome de atributo na propriedade JavaScript, escreva-o em minúsculas. Se o atributo tiver mais de uma palavra, no entanto, coloque a primeira letra de cada palavra após a primeira em maiúsculas: defaultChecked e tabIndex, por exemplo. No entanto, as propriedades do manipulador de eventos, como onclick, são uma exceção e são escritas em minúsculas.

Alguns nomes de atributos HTML são palavras reservadas em JavaScript. Para estes, a regra geral é prefixar o nome da propriedade com "html". O atributo HTML for (do <label> elemento), por exemplo, torna-se a propriedade htmlFor do JavaScript. "class" é uma palavra reservada em JavaScript, e o atributo de classe HTML muito importante é uma exceção à regra: ele se torna className no código JavaScript.

As propriedades que representam atributos HTML geralmente têm stringvalues. Mas quando o atributo é um valor booleano ou numérico (os atributos defaultChecked e maxLength de um <input>elemento, por exemplo), as propriedades são booleanos ou números em vez de cadeias de caracteres. Os atributos do manipulador de eventos sempre têm funções (ou nulos) como seus valores.

Observe que essa API baseada em propriedade para obter e definir attributevalues não define nenhuma maneira de remover um atributo de um elemento. Em particular, o operador delete não pode ser usado para essa finalidade. Se você precisar excluir um atributo, use o método removeAttribute().

O ATRIBUTO DE CLASSE O atributo de classe de um elemento HTML é particularmente importante. Seu valor é uma lista separada por espaços de classes CSS que se aplicam ao elemento e afetam a forma como ele é estilizado com CSS. Como class é uma palavra reservada em JavaScript, o valor desse atributo está disponível por meio da propriedade className em objetos Element. A propriedade className pode definir e retornar o valor do atributo class como uma cadeia de caracteres. Mas o atributo class é mal nomeado: seu valor

é uma lista de classes CSS, não uma única classe, e é comum na programação JavaScript do lado do cliente querer adicionar e remover nomes de classes individuais dessa lista em vez de trabalhar com a lista como uma única string.

Por esse motivo, os objetos Element definem uma propriedade classList que permite tratar o atributo class como uma lista. O valor da propriedade classList é um objeto iterável semelhante a Array. Embora o nome da propriedade seja classList, ela se comporta mais como um conjunto de classes e define os métodos add(), remove(), contains() e toggle():

Quando queremos que o usuário saiba que estamos ocupados, exibimos// um spinner. Para fazer isso, temos que remover a classe "hidden" e adicionar a classe // "animated" (supondo que as folhas de estilo estejam configuradas corretamente).

```
let spinner =  
document.querySelector("#spinner");  
spinner.classList.remove("oculto");  
spinner.classList.add("animado");
```

ATRIBUTO DO CONJUNTO DE DADOS Exerga anexar informações adicionais aos elementos HTML, normalmente quando o código JavaScript seleciona esses elementos e os manipula de alguma forma. Em HTML, qualquer atributo cujo nome seja minúsculo e comece com o prefixo "data-" é considerado válido e você pode usá-lo para qualquer finalidade. Esses "datasetattributes" não afetarão a apresentação dos elementos em que aparecem e definem uma maneira padrão de anexar dados adicionais sem comprometer a validade do documento.

No DOM, os objetos Element têm uma propriedade de conjunto de dados que se refere a um objeto que tem propriedades que correspondem aos atributos de dados com seu prefixo removido. Assim, dataset.x conteria o valor do atributo data-x. Os atributos hifenizados são mapeados para camelCased nomes de propriedade: o atributo data-section-number torna-se a propriedade dataset.sectionNumber.

Suponha que um documento HTML contenha este texto:

```
<h2 id="title" data-section-number="16.1">Atributos</h2>
```

Então você pode escrever JavaScript como este para acessar esse número de seção:

```
let número  
=document.querySelector("#title").dataset.sectionNumber;
```

15.3.4 Conteúdo do elemento

Olhe novamente para a árvore de documentos representada na Figura 15-1 e pergunte a si mesmo qual é o "conteúdo" do <p> elemento. Existem duas maneiras de responder a essa pergunta:

O conteúdo é a string HTML "Este é um documento <i>simples</i>". O conteúdo é a cadeia de caracteres de texto sem formatação "Este é um documento simples". Ambas são respostas válidas, e cada resposta é útil à sua maneira. As seções a seguir explicam como trabalhar com a representação HTML e a representação de texto simples do conteúdo de um elemento.

CONTEÚDO DO ELEMENTO COMO HTMLA PROPRIEDADE INNERHTML DE UM ELEMENTO RETORNA O CONTEÚDO DESSE ELEMENTO COMO UMA CADEIA DE CARACTERES DE MARCAÇÃO.
Definir essa propriedade em um elemento invoca o analisador do navegador da Web e substitui o conteúdo atual do elemento por uma representação analisada da nova cadeia de caracteres. Você pode testar isso abrindo o console do desenvolvedor e digitando:

```
document.body.innerHTML = "<h1>Opa</h1>";
```

Você verá que toda a página da web desaparece e é substituída pelo único título, "Opa". Os navegadores da Web são muito bons em analisar HTML e definir innerHTML geralmente é bastante eficiente. Observe, no entanto, que anexar texto à propriedade innerHTML com o operador += não é eficiente porque requer uma etapa de serialização para converter o conteúdo do elemento em uma cadeia de caracteres e, em seguida, uma etapa de análise para converter a nova cadeia de caracteres de volta no conteúdo do elemento.

AVISO

Ao usar essas APIs HTML, é muito importante que você nunca insira userinput no documento. Se você fizer isso, permitirá que usuários mal-intencionados injetem seus próprios scripts em seu aplicativo. Consulte "Cross-site scripting" para obter detalhes.

A propriedade outerHTML de um Element é como innerHTML, exceto que seu valor inclui o próprio elemento. Quando você queryouterHTML, o valor inclui as tags de abertura e fechamento do elemento. E quando você define outerHTML em um elemento, o newcontent substitui o próprio elemento.

Um método Element relacionado é insertAdjacentHTML(), que permite inserir uma string de marcação HTML arbitrária "adjacente" ao elemento especificado. A marcação é passada como o segundo argumento para este método, e o significado preciso de "adjacente" depende do valor do primeiro argumento. Esse primeiro argumento deve ser uma cadeia de caracteres com um dos valores "beforebegin", "afterbegin", "beforeend" ou "afterend". Esses valores correspondem aos pontos de inserção ilustrados na Figura 15-2.



Figura 15-2. Pontos de inserção para insertAdjacentHTML()

CONTEÚDO DO ELEMENTO COMO TEXTO SIMPLES Alguns vezes que você deseja consultar o conteúdo de um elemento como texto simples ou inserir texto simples em um documento (sem ter que escapar dos colchetes angulares e comerciais usados na marcação HTML). A maneira padrão de fazer isso é com a propriedade **textContent**:

```
let para = document.querySelector("p"); Primeiro <p> no
texto do documento
let = para.textContent; // Obtém o texto
do parágrafo
para.textContent = "Hello World!"; Alterar o
texto do parágrafo
```

A propriedade **textContent** é definida pela classe **Node**, portanto, funciona para nós de texto e nós de elemento. Para nós de elemento, ele localiza e retorna todo o texto em todos os descendentes do elemento.

A classe Element define uma propriedade innerText semelhante a textContent. innerText tem alguns comportamentos incomuns e complexos, como tentar preservar a formatação da tabela. No entanto, não está bem especificado nem implementado de forma compatível entre navegadores e não deve mais ser usado.

TEXTO EM <SCRIPT> ELEMENTOS

Os elementos embutidos <script> (ou seja, aqueles que não têm um atributo src) têm uma propriedade text que você pode usar para recuperar seu texto. O conteúdo de um <script> elemento nunca é exibido pelo navegador, e o analisador HTML ignora colchetes angulares e e comerciais dentro de um script. Isso torna um <script> elemento um local ideal para inserir dados textuais arbitrários para uso pelo seu aplicativo. Simplesmente defina o atributo type do elemento para algum valor (como "text/x-custom-data") que deixa claro que o script não é um código JavaScript executável. Se você fizer isso, o interpretador JavaScript ignorará o script, mas o elemento existirá na árvore do documento e sua propriedade text retornará os dados para você.

15.3.5 Criando, inserindo e excluindo nós

Vimos como consultar e alterar o conteúdo do documento usando strings de HTML e de texto simples. E também vimos que podemos percorrer um documento para examinar os nós individuais de elemento e texto dos quais ele é feito. Também é possível alterar um documento no nível de nós individuais. A classe Document define métodos para criar objetos Element, e os objetos Element e Text têm métodos para inserir, excluir e substituir nós na árvore.

Crie um novo elemento com o método createElement() da classe Document e anexe strings de texto ou outros elementos a ele com seus métodos append() e prepend():

```
let parágrafo = document.createElement("p"); Criar um  
<p> elemento vazio
```

```
deixe ênfase = document.createElement("em"); Cria <em>
umelementemphasis.append("World");// Adiciona texto ao <em>
elementparagraph.append("Hello ", accent, "!"); Adicione
texto e<em> a <p>paragraph.prepend("i");// Adicione mais
texto no início de <p>paragraph.innerHTML// =>
" ;Hello<em>World</em>! "
```

append() e prepend() recebem qualquer número de argumentos, que podem ser objetos Node ou strings. Os argumentos de cadeia de caracteres são convertidos automaticamente em nós de texto. (Você pode criar nós de texto explicitamente comdocument.createTextNode(), mas raramente há qualquer razão para isso.) append() adiciona os argumentos ao elemento no final da lista filha. prepend() adiciona os argumentos no início da lista filha.

Se você quiser inserir um nó Elemento ou Texto no meio da lista filho do elemento que contém, nem append() nem prepend() funcionarão para você. Nesse caso, você deve obter uma referência a um nó irmão e chamar before() para inserir o newcontent antes desse irmão ou after() para inseri-lo após esse irmão. Por exemplo:

Encontre o elemento de cabeçalho com class="greetings" let greetings = document.querySelector("h2.greetings");

Agora insira o novo parágrafo e uma régua horizontal após esse títulosaudações.after(parágrafo, document.createElement("hr"));

Como append() e prepend(), after() e before() pegam qualquer número de argumentos de string e elemento e os inserem todos em

o documento após a conversão de strings em nós de texto. `append()` e `prepend()` são definidos apenas em objetos `Element`, mas `after()` e `before()` funcionam nos nós `Element` e `Text`: você pode usá-los para inserir conteúdo relativo a um nó `Text`.

Observe que os elementos só podem ser inseridos em um ponto do documento. Se um elemento já estiver no documento e você inseri-lo em outro lugar, ele será movido para o novo local, não copiado:

Inserimos o parágrafo após este elemento, mas agora nós // o movemos para que apareça antes do elementogreetings.before(paragraph);

Se você quiser fazer uma cópia de um elemento, use o método `cloneNode()`, passando `true` para copiar todo o seu conteúdo:

Faça uma cópia do parágrafo e insira-o após oelemento saudaçõessaudações.depois(parágrafo.cloneNode(verdadeiro));

Você pode remover um nó `Element` ou `Text` do documento chamando seu método `remove()` ou pode substituí-lo chamando `replaceWith()`. `remove()` não recebe argumentos, and `replaceWith()` usa qualquer número de strings e elementos como `before()` e `after()` fazem:

Remova o elemento greetings do documento e substitua-o por// o elemento paragraph (movendo o parágrafo de seu localização atual// se já estiver inserido no documento).greetings.replaceWith(paragraph);

E agora remova o parágrafo.

```
parágrafo.remove();
```

A API DOM também define uma geração mais antiga de métodos para inserir e remover conteúdo. `appendChild()`, `insertBefore()`, `replaceChild()` e `removeChild()` são mais difíceis de usar do que os métodos mostrados aqui e nunca devem ser necessários.

15.3.6 Exemplo: Gerando um sumário

O Exemplo 15-1 mostra como criar dinamicamente um sumário para um documento. Ele demonstra muitas das técnicas de script de documentos descritas nas seções anteriores. O exemplo é bem comentado e você não deve ter problemas para seguir o código.

Exemplo 15-1. Gerando um sumário com a API DOM

```
/**  
 * TOC.js: cria um sumário para um documento.** Este script  
é executado quando o evento DOMContentLoaded é disparado e  
  
* Gera automaticamente um sumário para o documento.  
  
* Ele não define nenhum símbolo global, portanto, não  
deve entrar em conflito  
* com outros scripts.** Quando esse script é executado, ele  
primeiro procura um elemento de documentocom  
  
* um id de "TOC". Se não houver tal elemento, ele cria umno  
* início do documento. Em seguida, a função localiza tudo  
<h2>através de  
* <h6> , trata-as como títulos de seção e cria uma tabela  
de  
* conteúdo dentro do elemento TOC. A função adiciona números  
de seção  
* para cada título de seção e envolve os títulos em  
namedanchors para
```

* que o sumário pode ser vinculado a eles. As âncoras geradas têm nomes
* que começam com "TOC", então você deve evitar esse prefixo em seu próprio
* HTML.** As entradas no sumário gerado podem ser estilizadas com CSS. Todas as entradas * têm uma classe "TOCEntry". As entradas também têm uma classe que

* corresponde ao nível do título da seção. `<h1>` tags generate* entradas da classe "TOCLevel1", `<h2>` tags geram entradas de classe

* "TOCLevel2" e assim por diante. Os números de seção inseridos nos títulos têm
* classe "TOCSectNum".** Você pode usar este script com uma folha de estilo como esta:**#TOC { border: solid black 1px; margin: 10px; padding: 10px; }

```
 *. TOCEntry { margem: 5px 0px; }*. TOCEntry a { text-decoration: none; }*. TOCLevel1 {tamanho da fonte: 16pt; peso da fonte: negrito; }*. TOCLevel2 {tamanho da fonte: 14pt; margem esquerda: .25in; }*. TOCLevel3 {tamanho da fonte: 12pt; margem esquerda: 5 pol; }*. TOCSectNum:after { content: ":"; }** Para ocultar os números das seções, use isto:**. TOCSectNum { display: none }**/document.addEventListener("DOMContentLoaded", () => {
```

Encontre o elemento do contêiner TOC.// Se não houver um, crie um no início do documento.

```
let toc = document.querySelector("#TOC");
if (!toc) {
  toc = document.createElement("div");
  toc.id = "TOC";
  document.body.prepend(toc);}
```

Encontre todos os elementos do título da seção. Estamos assumindo aqui que o título do documento usa `<h1>` e que as seções dentro do

documento são marcado com <h2> através <h6>de cabeçalhos .let = document.querySelectorAll("h2,h3,h4,h5,h6");

Inicialize uma matriz que acompanhe os números da seção.let sectionNumbers = [0,0,0,0,0];

Agora percorra os elementos do cabeçalho da seção que encontramos.for(let cabeçalho dos títulos) {

Pule o cabeçalho se ele estiver dentro do contêiner de sumário.if (heading.parentNode === toc) { continuar;}

Descubra qual é o cabeçalho de nível.// Subtraia 1 porque <h2> é um nível 1 heading.let level = parseInt(heading.tagName.charAt(1)) - 1;

Incremente o número da seção para este nível de título// e redefina todos os números de nível de título inferior para zero.sectionNumbers[level-1]++; for(let i = level; i < sectionNumbers.length; i++) { sectionNumbers[i] = 0;}

Agora combine os números de seção para todos os níveis de título // para produzir um número de seção como 2.3.1.let sectionNumber = sectionNumbers.slice(0, level).join(".");

Adicione o número da seção ao título do cabeçalho da seção.// Colocamos o número em um para torná-lo estilizável.

```
let span = document.createElement("span");
span.className = "TOCSectNum";
span.textContent = sectionNumber;
título.prepend(span);
```

Envolva o título em uma âncora nomeada para que possamos vincular a

```
let âncora = document.createElement("a");
let fragmentName = 'TOC${sectionNumber}';
anchor.name = fragmentName;
```

```
heading.before(anchor); // Insira a âncora antes
rubrica
âncora.append(título);      e se move para dentro
âncora

Agora crie um link para esta seção. let link =
document.createElement("a"); link.href =
'#${fragmentName}'; // Destino do link

Copie o texto do título no link. Este é um cofre
uso de
innerHTML porque não estamos inserindo nenhum
Strings.
link.innerHTML = cabeçalho.innerHTML;

Coloque o link em um div que seja estilizável com base em
o nível.
let entrada = document.createElement("div");
entry.classList.add("TOCEntry",
'TOCLevel${nível}'); entrada.append(link);

E adicione o div ao TOC
container.toc.append(entry));});
```

15.4 Script CSS

Vimos que o JavaScript pode controlar a estrutura lógica e o conteúdo dos documentos HTML. Ele também pode controlar a aparência visual e o layout desses documentos por meio de scripts CSS. As subseções a seguir explicam algumas técnicas diferentes que o código JavaScript pode usar para trabalhar com CSS.

Este é um livro sobre JavaScript, não sobre CSS, e esta seção pressupõe que você já tenha um conhecimento prático de como o CSS é usado para estilizar conteúdo HTML. Mas vale a pena mencionar alguns dos

Estilos CSS que são comumente roteirizados a partir de JavaScript:

- Definir o estilo de exibição como "nenhum" oculta um elemento. Posteriormente, você pode mostrar o elemento definindo display para algum outro valor. Você pode
- posicionar elementos dinamicamente definindo o estilo de posição como "absoluto", "relativo" ou "fixo" e, em seguida, definindo os estilos superior e esquerdo para as coordenadas desejadas. Isso é importante ao usar JavaScript para exibir conteúdo dinâmico, como diálogos modais e dicas de ferramentas. Você pode deslocar, dimensionar e girar elementos com o estilo de transformação. Você pode animar alterações em outros estilos CSS com o estilo de transição.
- Essas animações são manipuladas automaticamente pelo navegador da Web e não exigem JavaScript, mas você pode usar JavaScript para iniciar as animações.

15.4.1 Classes CSS

A maneira mais simples de usar JavaScript para afetar o estilo do conteúdo do documento é adicionar e remover nomes de classe CSS do atributo de classe de tags HTML. Isso é fácil de fazer com a propriedade `classList` dos objetos Element, conforme explicado em "[O atributo class](#)".

Suponha, por exemplo, que a folha de estilo do seu documento inclua uma definição para uma classe "oculta":

```
.hidden {  
exibição:nenhu  
m;}
```

Com esse estilo definido, você pode ocultar (e depois mostrar) um elemento com código como este:

```
Suponha que este elemento "tooltip" tenha class="hidden" no arquivo HTML.// Podemos torná-lo visível assim:document.querySelector("#tooltip").classList.remove("hidden");
```

```
E podemos ocultá-lo novamente assim:document.querySelector("#tooltip").classList.add("hidden");
```

15.4.2 Estilos embutidos

Para continuar com o exemplo de dica de ferramenta anterior, suponha que o documento esteja estruturado com apenas um único elemento de dica de ferramenta e queremos posicioná-lo dinamicamente antes de exibi-lo. Em geral, não podemos criar uma classe de folha de estilo diferente para cada posição possível da dica de ferramenta, portanto, a propriedade classList não nos ajudará no posicionamento.

Nesse caso, precisamos criar um script do atributo style do tooltipElement para definir estilos embutidos específicos para esse elemento. O DOM define uma propriedade de estilo em todos os objetos Element que correspondem ao atributo style. Ao contrário da maioria dessas propriedades, no entanto, a propriedade style não é uma cadeia de caracteres. Em vez disso, é um objeto CSSStyleDeclaration: uma representação analisada dos estilos CSS que aparecem em textual form no atributo style. Para exibir e definir a posição de nossa dica de ferramenta hipotética com JavaScript, podemos usar um código como este:

```
function displayAt(dica de ferramenta, x, y) {  
    tooltip.style.display = "bloco";  
    tooltip.style.position = "absoluto";  
    tooltip.style.left = '${x}px';
```

```
tooltip.style.top = '${y}px';}
```

CONVENÇÕES DE NOMENCLATURA: PROPRIEDADES CSS EM JAVASCRIPT

Muitas propriedades de estilo CSS, como font-size, contêm hífens em seus nomes. Em JavaScript, o hífen é interpretado como um sinal de menos e não é permitido em nomes de propriedade ou outros identificadores. Portanto, os nomes das propriedades do objeto CSSStyleDeclaration são ligeiramente diferentes dos nomes das propriedades CSS reais. Se um nome de propriedade CSS contiver um ou mais hífens, o nome da propriedade CSSStyleDeclaration será formado removendo os hífens e colocando a letra em maiúscula imediatamente após cada hífen. A propriedade CSS border-left-width é acessada por meio da propriedade borderLeftWidth do JavaScript, por exemplo, e a propriedade CSS font-family é gravada como fontFamily em JavaScript.

Ao trabalhar com as propriedades de estilo do objeto CSSStyleDeclaration, lembre-se de que todos os valores devem ser especificados como cadeias de caracteres. Em uma folha de estilo ou atributo de estilo, você pode escrever:

```
exibição: bloco; família de fontes: sans-serif; cor de  
fundo:#ffffff;
```

Para fazer a mesma coisa para um elemento e com JavaScript, você deve citar todos os valores:

```
e.style.display = "bloco";  
e.style.fontFamily = "sans-serif";  
e.style.backgroundColor = "#ffffff";
```

Observe que os ponto-e-vírgula saem das strings. Estes são apenas pontos e vírgulas normais do JavaScript; os pontos e vírgulas que você usa nas folhas de estilo CSS não são necessários como parte dos valores de cadeia de caracteres definidos com JavaScript.

Além disso, lembre-se de que muitas propriedades CSS requerem unidades como "px" para pixels ou "pt" para pontos. Portanto, não é correto definir o

marginLeft como esta:

```
e.style.marginLeft = 300;      Incorreto: este é um número,
não é uma string.e.style.marginLeft = "300";//
Incorreto: as unidades estão ausentes
```

As unidades são necessárias ao definir propriedades de estilo em JavaScript, assim como ao definir propriedades de estilo em folhas de estilo. A maneira correta de definir o valor da propriedade marginLeft de um elemento e para 300 pixels é:

```
e.style.marginLeft = "300px";
```

Se você deseja definir uma propriedade CSS para um valor calculado, certifique-se de anexar as unidades no final do cálculo:

```
e.style.left = '${x0 + left_border + left_padding}px';
```

Lembre-se de que algumas propriedades CSS, como margin, são atalhos para outras propriedades, como margin-top, margin-right, margin-bottom e margin-left. O objeto CSSStyleDeclaration tem propriedades que correspondem a essas propriedades de atalho. Por exemplo, você pode definir a propriedade margin assim:

```
e.style.margin = '${top}px ${right}px ${bottom}px ${left}px';
```

Às vezes, você pode achar mais fácil definir ou consultar o estilo embutido de um elemento como um único valor de cadeia de caracteres em vez de como um objeto CSSStyleDeclaration. Para fazer isso, você pode usar os métodos Element.getAttribute() e setAttribute() ou pode usar a propriedade cssText

do objeto CSSStyleDeclaration:

```
Copie os estilos embutidos do elemento e para o elemento  
f:f.setAttribute("style", e.getAttribute("style"));
```

```
Ou faça assim: f.style.cssText =  
e.style.cssText;
```

Ao consultar a propriedade style de um elemento, lembre-se de que ela representa apenas os estilos embutidos de um elemento e que a maioria dos estilos para a maioria dos elementos são especificados em folhas de estilo em vez de embutidas. Além disso, os valores obtidos ao consultar a propriedade style usarão quaisquer unidades e qualquer formato de propriedade de atalho que seja realmente usado no atributo HTML, e seu código pode ter que fazer uma análise sofisticada para interpretá-los. Em geral, se você quiser consultar os estilos de um elemento, provavelmente desejará o estilo computado, que é discutido a seguir.

15.4.3 Estilos calculados

O estilo calculado para um elemento é o conjunto de valores de propriedade que o navegador deriva (ou calcula) do estilo embutido do elemento mais todas as regras de estilo aplicáveis em todas as folhas de estilo: é o conjunto de propriedades realmente usado para exibir o elemento. Assim como os estilos embutidos, os estilos computados são representados com um objeto CSSStyleDeclaration. Ao contrário dos estilos embutidos, no entanto, os estilos computados são somente leitura. Você não pode definir esses estilos, mas o objeto CSSStyleDeclaration calculado para um elemento permite que você determine quais valores de propriedade de estilo o navegador usou ao renderizar esse elemento.

Obtenha o estilo calculado para um elemento com o

`getComputedStyle()` do objeto Window. O primeiro argumento para esse método é o elemento cujo estilo calculado é desejado. O segundo argumento opcional é usado para especificar um CSSPseudoelement, como "`::before`" ou "`::after`":

```
let título =  
document.querySelector("#section1title"); let estilos  
= window.getComputedStyle(título); let beforeStyles =  
window.getComputedStyle(title,"::before");
```

O valor de retorno de `getComputedStyle()` é um objeto `CSSStyleDeclaration` que representa todos os estilos que se aplicam ao elemento especificado (ou pseudoelemento). Há várias diferenças importantes entre um objeto `CSSStyleDeclaration` que representa estilos embutidos e um que representa estilos calculados:

- As propriedades de estilo computadas são somente leitura. As propriedades de estilo calculadas são absolutas: unidades relativas como porcentagens e pontos são convertidas em valores absolutos. Qualquer propriedade que especifique um tamanho (como um tamanho de margem ou um tamanho de fonte) terá um valor medido em pixels. Este valor será um string com um sufixo "px", então você ainda precisará analisá-lo, mas você não terá que se preocupar em analisar ou converter outras unidades. As propriedades cujos valores são cores serão retornadas no formato "rgb()" ou "rgba()". As propriedades de atalho não são computadas, apenas as propriedades fundamentais nas quais elas se baseiam são. Não consulte a `marginproperty`, por exemplo, mas use `marginLeft`, `marginTop` e assim por diante. Da mesma forma, não consulte `border` ou `evenborderWidth`. Em vez disso, use `borderLeftWidth`, `borderTopWidth` e assim por diante.

A propriedade `cssText` do estilo calculado é indefinida. Um objeto `CSSStyleDeclaration` retornado por `getComputedStyle()` geralmente contém muito mais informações sobre um elemento do que o `CSSStyleDeclaration` obtido da propriedade de estilo embutido desse elemento. Mas os estilos calculados podem ser complicados, e consultá-los nem sempre fornece as informações que você espera. Considere o atributo `font-family`: ele aceita uma lista separada por vírgulas de famílias de fontes desejadas para portabilidade entre plataformas. Ao consultar a propriedade `fontFamily` de um estilo calculado, você está simplesmente obtendo o valor do estilo de família de fontes mais específico que se aplica ao elemento. Isso pode retornar um valor como `"arial,helvetica,sans-serif"`, que não informa qual tipo de letra está realmente em uso. Da mesma forma, se um elemento não estiver absolutamente posicionado, tentar consultar sua posição e tamanho por meio das propriedades `top` e `left` de seu estilo calculado geralmente retorna o valor `auto`. Este é um valor CSS perfeitamente legal, mas provavelmente não é o que você estava procurando.

Embora o CSS possa ser usado para especificar com precisão a posição e o tamanho dos elementos do documento, consultar o estilo calculado de um elemento não é a maneira preferida de determinar o tamanho e a posição do elemento. Consulte §15.5.2 para obter uma alternativa mais simples e portátil.

15.4.4 Folhas de estilo de script

Além de criar scripts de atributos de classe e estilos embutidos, o JavaScript também pode manipular as próprias folhas de estilo. As folhas de estilo são associadas a um documento HTML com uma `<style>` tag ou com uma tag `<link rel="stylesheet">`. Ambas são tags HTML regulares, então

você pode fornecer a ambos os atributos id e, em seguida, procurá-los com document.querySelector().

Os objetos Element para tags <style> e <link> têm uma propriedade disabled que você pode usar para desabilitar toda a folha de estilo.

Você pode usá-lo com um código como este:

Esta função alterna entre os temas "claros" e "escuros" função toggleTheme() {let lightTheme = document.querySelector("#light-theme");

```
let darkTheme = document.querySelector("#dark-theme"); if  
(darkTheme.disabled) { // Atualmente claro, mude para  
escuro  
    lightTheme.disabled = true;  
    darkTheme.disabled = false;}  
else {  
    Atualmente escuro,  
Mudar para luz  
    lightTheme.disabled = false;  
    darkTheme.disabled = true;}}
```

Outra maneira simples de criar scripts de folhas de estilo é inserir novas no documento usando técnicas de manipulação de DOM que já vimos. Por exemplo:

```
function setTheme(nome) {  
    Crie um novo elemento <link rel="stylesheet"> para  
carregar a folha de estilo nomeada  
    let link =  
        document.createElement("link"); link.id =  
        "tema"; link.rel = "folha de estilo";  
        link.href = 'temas/${nome}.css';
```

Procure um link existente com id "tema"

```
let currentTheme = document.querySelector("#theme");
if (currentTheme) {
    Se houver um tema existente, substitua-o pelo novo.
    currentTheme.replaceWith(link);}
else {
    Caso contrário, basta inserir o link para o tema folha de estilo.
document.head.append(link);}}
```

Menos sutilmente, você também pode simplesmente inserir uma string de HTML contendo uma<style> tag em seu documento. Este é um truque divertido, por exemplo:

```
document.head.insertAdjacentHTML(
"beforeend", "<style>body{transform:rotate(180deg)}
</style>");
```

Os navegadores definem uma API que permite que o JavaScript olhe dentro stylesheets para consultar, modificar, inserir e excluir regras de estilo nesse stylesheet. Essa API é tão especializada que não está documentada aqui. Você pode ler sobre isso no MDN pesquisando por "CSSStyleSheet" e "CSS Object Model".

15.4.5 Animações e eventos CSS

Suponha que você tenha as duas classes CSS a seguir definidas em uma folha de estilo:

```
.transparent { opacity: 0; }. fadeable {
transição: opacidade .5s facilidade }
```

Se você aplicar o primeiro estilo a um elemento, ele será totalmente transparente e

portanto, invisível. Mas se você aplicar o segundo estilo que informa ao navegador que, quando a opacidade do elemento muda, essa mudança deve ser animada por um período de 0,5 segundos, "ease-in" especifica que a animação de mudança de opacidade deve começar lenta e depois acelerar.

Agora suponha que seu documento HTML contenha um elemento com a classe "fadeable":

```
<div id="subscribe" class="fadeable notification">...</div>
```

Em JavaScript, você pode adicionar a classe "transparent":

```
document.querySelector("#subscribe").classList.add("transparent");
```

Esse elemento é configurado para animar alterações de opacidade. Adicionar a classe "transparente" altera a opacidade e aciona uma animação: o navegador "desaparece" o elemento para que ele se torne totalmente transparente durante o período de meio segundo.

Isso também funciona ao contrário: se você remover a classe "transparente" de um elemento "fadeable", isso também é uma mudança de opacidade, e o elemento desaparece e se torna visível novamente.

O JavaScript não precisa fazer nenhum trabalho para fazer essas animações acontecerem: elas são um efeito CSS puro. Mas o JavaScript pode ser usado para acioná-los.

O JavaScript também pode ser usado para monitorar o progresso de uma transição CSS PORQUE o navegador da Web dispara eventos no início e no final de um

transição. O evento "transitionrun" é despachado quando a transição é acionada pela primeira vez. Isso pode acontecer antes do início de qualquer alteração visual, quando o estilo de atraso de transição tiver sido especificado. Uma vez que as mudanças visuais começam, um evento "transitionstart" é despachado e, quando a animação é concluída, um evento "transitionend" é despachado. O alvo de todos esses eventos é o elemento que está sendo animado, é claro. O objeto de evento passado para manipuladores para esses eventos é um objeto TransitionEvent. Ele tem uma propriedade propertyName que especifica a propriedade CSS que está sendo animada e uma propriedade elapsedTime que para eventos "transitionend" especifica quantos segundos se passaram desde o evento "transitionstart".

Além das transições, o CSS também suporta uma forma mais complexa de animação conhecida simplesmente como "Animações CSS". Eles usam propriedades CSS, como animation-name e animation-duratione uma regra de @keyframes especial para definir detalhes da animação. Detalhes de como as animações CSS funcionam estão além do escopo deste livro, mas, mais uma vez, se você definir todas as propriedades de animação em uma classe CSS, poderá usar JavaScript para acionar a animação simplesmente adicionando a classe ao elemento que deve ser animado.

E, como as transições CSS, as animações CSS também acionam eventos que seu código JavaScript pode ouvir. "animationstart" é despachado quando a animação é iniciada e "animationend" é despachado quando é concluída. Se a animação se repetir mais de uma vez, um evento de "animaçãoiteração" será despachado após cada repetição, exceto a última. O destino do evento é o elemento animado e o objeto de evento passado para funções de manipulador é um objeto AnimationEvent. Esses eventos

inclua uma propriedade `animationName` que especifica a propriedade `animation-name` que define a animação e a propriedade `anelapsedTime` que especifica quantos segundos se passaram desde o início da animação.

15.5 Geometria e rolagem do documento

Neste capítulo até agora, pensamos em documentos como árvores abstratas de elementos e nós de texto. Mas quando um navegador renderiza um documento dentro de uma janela, ele cria uma representação visual do documento em que cada elemento tem uma posição e um tamanho. Muitas vezes, os aplicativos da Web podem tratar documentos como árvores de elementos e nunca precisam pensar em como esses elementos são renderizados na tela. Às vezes, no entanto, é necessário determinar a geometria precisa de um elemento. Se, por exemplo, você quiser usar CSS para posicionar dinamicamente um elemento (como uma dica de ferramenta) ao lado de algum elemento comum posicionado no navegador, você precisa ser capaz de determinar a localização desse elemento.

As subseções a seguir explicam como você pode ir e voltar entre o modelo abstrato baseado em árvore de um documento e a exibição geométrica baseada em coordenadas do documento conforme ele é disposto em uma janela do navegador.

15.5.1 Coordenadas do documento e ViewportCoordinates

A posição de um elemento de documento é medida em pixels CSS, com a coordenada x aumentando para a direita e a coordenada y aumentando

à medida que descemos. No entanto, existem dois pontos diferentes que podemos usar como origem do sistema de coordenadas: as coordenadas x e y de um elemento podem ser relativas ao canto superior esquerdo do documento ou relativas ao canto superior esquerdo da janela de exibição na qual o documento é exibido. Em janelas e guias de nível superior, a "janela de visualização" é a parte do navegador que realmente exibe o conteúdo do documento: exclui o "chrome" do navegador, como menus, barras de ferramentas e guias. Para documentos exibidos em <iframe> tags, é o elemento iframe no DOM que define a janela de visualização do documento aninhado. Em ambos os casos, quando falamos sobre a posição de um elemento, devemos deixar claro se estamos usando coordenadas do documento ou coordenadas da janela de visualização. (Observe que as coordenadas da janela de visualização às vezes são chamadas de "coordenadas de janela").

Se o documento for menor que a janela de visualização ou se não tiver sido rolado, o canto superior esquerdo do documento estará no canto superior esquerdo da janela de exibição e os sistemas de coordenadas do documento e da janela de visualização serão os mesmos. Em geral, no entanto, para converter entre os dois sistemas de coordenadas, devemos adicionar ou subtrair os deslocamentos de rolagem. Se um elemento tiver uma coordenada y de 200 pixels nas coordenadas do documento, por exemplo, e se o usuário tiver rolado para baixo em 75 pixels, esse elemento terá uma coordenada y de 125 pixels nas coordenadas da janela de visualização. Da mesma forma, se um elemento tiver uma coordenada x de 400 nas coordenadas da janela de visualização depois que o usuário rolar a janela de visualização 200 pixels horizontalmente, a coordenada x do elemento nas coordenadas do documento será 600.

Se usarmos o modelo mental de documentos impressos em papel, é lógico supor que cada elemento em um documento deve ter uma posição única nas coordenadas do documento, independentemente de quanto o usuário tenha rolado

o documento. Essa é uma propriedade atraente de documentos em papel e se aplica a documentos simples da Web, mas, em geral, as coordenadas de documentos realmente não funcionam na Web. O problema é que a propriedade CSSOverflow permite que os elementos dentro de um documento contenham mais conteúdo do que ele pode exibir. Os elementos podem ter suas próprias barras de rolagem e servir como janelas de visualização para o conteúdo que contêm. O fato de a web permitir elementos de rolagem dentro de um documento de rolagem significa que simplesmente não é possível descrever a posição de um elemento dentro do documento usando um único ponto (x,y).

Como as coordenadas do documento realmente não funcionam, o JavaScript do lado do cliente tende a usar coordenadas de janela de visualização. Os métodos getBoundingClientRect() e elementFromPoint() descritos a seguir usam coordenadas de viewport, por exemplo, e as propriedades clientX e clientY de objetos de evento de mouse e ponteiro também usam esse sistema de coordenadas.

Quando você posiciona explicitamente um elemento usando CSSposition:fixed, as propriedades top e left são interpretadas nas coordenadas da janela de visualização. Se você usar position:relative, o elemento será posicionado em relação ao local em que estaria se não tivesse a propriedade position definida. Se você usar position:absolute, then top e left são relativos ao documento ou ao elemento contendingpositioned mais próximo. Isso significa, por exemplo, que um elemento posicionado de forma absoluta dentro de um elemento relativamente posicionado é posicionado em relação ao elemento de contêiner, não em relação ao documento geral. Às vezes, é muito útil criar um contêiner relativamente posicionado com top e left definidos como 0 (para que o contêiner seja disposto normalmente) em

para estabelecer uma nova origem do sistema de coordenadas para os elementos posicionados de forma absoluta que ele contém. Podemos nos referir a esse novo sistema de coordenadas como "coordenadas de contêiner" para distingui-lo das coordenadas do documento e das coordenadas da janela de visualização.

CSS PIXELS

Se, como eu, você tem idade suficiente para se lembrar de monitores de computador com resoluções de 1024×768 e telefones com tela sensível ao toque com resoluções de 320×480 , então você ainda pode pensar que a palavra "pixel" se refere a um único "elemento de imagem" no hardware. Os monitores 4K e telas "retina" de hoje têm uma resolução tão alta que os pixels do software foram desacoplados dos pixels do hardware. Um pixel CSS é, portanto, um pixel JavaScript do lado do cliente, pode consistir em vários pixels de dispositivo. A propriedade `devicePixelRatio` do objeto `Window` especifica quantos pixels de dispositivo são usados para cada pixel de software. Um "dpr" de 2, por exemplo, significa que cada pixel de software é na verdade uma grade de 2×2 pixels de hardware. O valor `devicePixelRatio` depende da resolução física do seu hardware, das configurações do sistema operacional e do nível de zoom do navegador.

`devicePixelRatio` não precisa ser um inteiro. Se você estiver usando um tamanho de fonte CSS de "12px" e a proporção de pixels do dispositivo for 2,5, o tamanho real da fonte, em pixels do dispositivo, será 30. Como os valores de pixel que usamos em CSS não correspondem mais diretamente a pixels individuais na tela, as coordenadas de pixel não precisam mais ser inteiras. Se o `devicePixelRatio` for 3, uma coordenada de 3,33 fará todo o sentido. E se a razão for realmente 2, então uma coordenada de 3,33 será arredondada para 3,5.

15.5.2 Consultando a geometria de um elemento

Você pode determinar o tamanho (incluindo a borda e o preenchimento CSS, mas não a margem) e a posição (nas coordenadas da janela de visualização) de um elemento chamando seu método `getBoundingClientRect()`. Ele recebe no argumento e retorna um objeto com as propriedades `left`, `right`, `top`, `bottom`, `width` e `height`. As propriedades `left` e `top` fornecem as coordenadas x e y do canto superior esquerdo do elemento, e as propriedades `right` e `bottom` fornecem as coordenadas do canto inferior direito. As diferenças entre esses valores são as propriedades `width` e `height`.

Elementos de bloco, como imagens, parágrafos e elementos, <div> são sempre retangulares quando dispostos pelo navegador. Elementos embutidos, como , <code>, e elementos, no entanto, podem abranger várias linhas e, portanto, podem consistir em vários retângulos. Imagine, por exemplo, algum texto dentro e tags que por acaso são exibidos de modo que se encaixam em duas linhas. Seus retângulos consistem no final da primeira linha e no início da segunda linha. Se você chamar getBoundingClientRect() nesse elemento, o boundingrectangle incluirá toda a largura de ambas as linhas. Se você quiser consultar os retângulos individuais de elementos embutidos, chame o método getClientRects() para obter um objeto somente leitura, semelhante a um arraycujos elementos são objetos retangulares como os retornados porgetBoundingClientRect().

15.5.3 Determinando o elemento em um ponto

O método getBoundingClientRect() nos permite determinar a posição atual de um elemento em uma janela de visualização. Às vezes, queremos ir na outra direção e determinar qual elemento está em um determinado local na janela de visualização. Você pode determinar isso com o método elementFromPoint() do objeto Document. Chame thismethod com as coordenadas x e y de um ponto (usando coordenadas viewport, não coordenadas de documento: as coordenadas clientX e clientY de um evento de mouse funcionam, por exemplo).elementFromPoint() retorna um objeto Element que está na posição especificada. O algoritmo de detecção de ocorrências para selecionar o elementonão é especificado com precisão, mas a intenção desse método é que ele retorne o mais interno (mais profundamente aninhado) e o mais alto (CSS z- mais alto

index atributo) nesse ponto.

15.5.4 Rolagem

O método scrollTo() do objeto Window pega as coordenadas x e y de um ponto (nas coordenadas do documento) e as define como os deslocamentos da barra de rolagem. Ou seja, ele rola a janela para que o ponto especificado esteja no canto superior esquerdo da janela de exibição. Se você especificar um ponto que esteja muito próximo da parte inferior ou muito perto da borda direita do documento, o navegador o moverá o mais próximo possível do canto superior esquerdo, mas não conseguirá levá-lo até lá. O código a seguir rola o navegador para que a página inferior do documento fique visível:

```
Obter as alturas do documento e viewport.let
documentHeight = document.documentElement.offsetHeight;
let viewportHeight = window.innerHeight;// E role para que
a última "página" apareça no viewportwindow.scrollTo(0,
documentHeight - viewportHeight);
```

O método scrollBy() da janela é semelhante a scrollTo(), mas seus argumentos são relativos e são adicionados à posição de rolagem atual:

```
Role 50 pixels para baixo a cada 500 ms. Observe que não
há como desativar isso!setInterval(() => {
scrollBy(0,50)}, 500);
```

Se você quiser rolar suavemente com scrollTo() ou scrollBy(), passe um único argumento de objeto em vez de dois números, assim:

```
window.scrollTo({
    esquerda: 0,
```

```
top: documentHeight -  
viewportHeight, behavior: "smooth"});
```

Muitas vezes, em vez de rolar para um local numérico em um documento, nós apenas queremos rolar para que um determinado elemento do documento fique visível. Você pode fazer isso com o método scrollIntoView() no elemento HTML desejado. Esse método garante que o elemento no qual ele é invocado esteja visível na janela de exibição. Por padrão, ele tenta colocar a borda superior do elemento na parte superior ou próxima à parte superior da janela de exibição. Se false for passado como o único argumento, ele tentará colocar a borda inferior do elemento na parte inferior da janela de visualização. O navegador também rolará a janela de visualização horizontalmente conforme necessário para tornar o elemento visível.

Você também pode passar um objeto para scrollIntoView(), definindo a propriedade behavior:"smooth" para rolagem suave. Você pode definir a propriedade block para especificar onde o elemento deve ser posicionado verticalmente e a propriedade inline para especificar como ele deve ser posicionado horizontalmente se a rolagem horizontal for necessária. Os valores legais para ambas as propriedades são início, fim, mais próximo e central.

15.5.5 Tamanho da janela de visualização, tamanho do conteúdo e ScrollPosition

Como discutimos, as janelas do navegador e outros elementos HTML podem exibir conteúdo de rolagem. Quando esse é o caso, às vezes precisamos saber o tamanho da janela de visualização, o tamanho do conteúdo e os deslocamentos de rolagem do conteúdo dentro da janela de visualização. Esta seção cobre esses detalhes.

Para janelas do navegador, o tamanho da janela de visualização é fornecido pelas propriedades `window.innerWidth` e `window.innerHeight`. (As páginas da Web otimizadas para dispositivos móveis geralmente usam uma tag `<meta name="viewport">` para `<head>` definir a largura desejada da janela de visualização da página.) O tamanho total do documento é o mesmo que o tamanho do `<html>` elemento, `document.documentElement`. Você pode chamar `getBoundingClientRect()` em `document.documentElement` para obter a largura e a altura do documento, ou você pode usar as propriedades `offsetWidth` e `offsetHeight` de `document.documentElement`. Os deslocamentos de rolagem do documento em sua janela de visualização estão disponíveis como `window.scrollX` e `window.scrollY`. Essas são propriedades somente leitura, portanto, você não pode configurá-las para rolar o documento: use `window.scrollTo()` em vez disso.

As coisas são um pouco mais complicadas para os elementos. Cada `Element` define os três grupos de propriedades a seguir:

<code>offsetWidth</code>	<code>clientWidth</code>	<code>scrollWidth</code>
<code>offsetHeight</code>	<code>clientHeight</code>	<code>scrollHeight</code>
<code>offsetLeft</code>	<code>clientLeft</code>	<code>rolagemEsquerda</code>
<code>offsetTop</code>	<code>clientTop</code>	<code>scrollTop</code>
<code>offsetParent</code>		

As propriedades `offsetWidth` e `offsetHeight` de um elemento retornam seu tamanho na tela em pixels CSS. Os tamanhos retornados incluem o elemento, borda e preenchimento, mas não margens. As propriedades `offsetLeft` e `offsetTop` retornam as coordenadas x e y do elemento. Para muitos elementos, esses valores são coordenadas de documento. Mas para descendentes de elementos posicionados e para alguns outros elementos, tais como

Como células de tabela, essas propriedades retornam coordenadas relativas ao elemento Ancestor em vez do próprio documento. A propriedade offsetParent especifica a qual elemento as propriedades são relativas. Essas propriedades de deslocamento são todas somente leitura.

clientWidth e clientHeight são como offsetWidth e offsetHeight, exceto que não incluem o tamanho da borda — apenas a área de conteúdo e seu preenchimento. As propriedades clientLeft e clientTop não são muito úteis: elas retornam a distância horizontal e vertical entre a parte externa do preenchimento de um elemento e a parte externa de sua borda. Normalmente, esses valores são apenas a largura das bordas esquerda e superior. Essas propriedades do cliente são todas somente leitura. Para elementos embutidos como `<i>`, `<code>`, e ``, todos eles retornam 0.

scrollWidth e scrollHeight retornam o tamanho da área de conteúdo de um elemento, além de seu preenchimento e qualquer conteúdo excedente. Quando o conteúdo se encaixa na área de conteúdo sem estouro, essas propriedades são iguais a clientWidth e clientHeight. Mas quando há estouro, eles incluem o conteúdo excedente e retornam valores maiores que clientWidth e clientHeight. scrollLeft e scrollTop fornecem o deslocamento de rolagem do conteúdo do elemento dentro da janela de visualização do elemento. Ao contrário de todas as outras propriedades descritas aqui, scrollLeft e scrollTop são propriedades graváveis, e você pode configurá-las para rolar o conteúdo dentro de um elemento. (Na maioria dos navegadores, os objetos Element também têm os métodos scrollTo() e scrollBy() como o objeto Window, mas eles ainda não são universalmente suportados.)

15.6 Componentes Web

HTML é uma linguagem para marcação de documentos e define um rico conjunto de tags para essa finalidade. Nas últimas três décadas, tornou-se uma linguagem usada para descrever as interfaces de usuário de aplicativos da web, mas tags HTML básicas, como `<input>` e `<button>` são inadequadas para designs modernos de interface do usuário. Os desenvolvedores da Web são capazes de fazê-lo funcionar, mas apenas usando CSS e JavaScript para aumentar a aparência e o comportamento das tags HTML básicas. Considere um componente típico da interface do usuário, como a caixa de pesquisa mostrada na Figura 15-3.

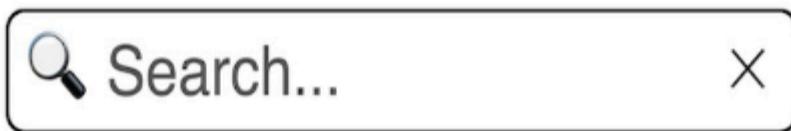


Figura 15-3. Um componente de interface do usuário da caixa de pesquisa

O `<input>` elemento HTML pode ser usado para aceitar uma única linha de entrada do usuário, mas não tem como exibir ícones como a lupa à esquerda e o X de cancelamento à direita. Para implementar um elemento de interface de usuário moderno como este para a web, precisamos usar pelo menos quatro elementos HTML: um `<input>` elemento para aceitar e exibir a entrada do usuário, dois `` elementos (ou, neste caso, dois `` elementos exibindo glifos Unicode) e um elemento de contêiner `<div>` para armazenar esses três filhos. Além disso, temos que usar CSS para ocultar a borda padrão do `<input>` elemento e definir uma borda para o contêiner. E precisamos usar JavaScript para fazer com que todos os elementos HTML funcionem juntos. Quando o usuário clica no ícone X, precisamos de um manipulador de eventos para limpar a entrada

do <input> elemento, por exemplo.

Isso é muito trabalho a ser feito toda vez que você deseja exibir uma caixa de pesquisa em um aplicativo da web, e a maioria dos aplicativos da web hoje não são escritos usando HTML "bruto". Em vez disso, muitos desenvolvedores da Web usam estruturas como React e Angular que suportam a criação de componentes de interface de usuário reutilizáveis, como a caixa de pesquisa mostrada aqui. Componentes da Web é uma alternativa nativa do navegador para essas estruturas baseadas em três adições relativamente recentes aos padrões da Web que permitem que o JavaScript estenda o HTML com novas tags que funcionam como componentes de interface do usuário independentes e reutilizáveis.

As subseções a seguir explicam como usar componentes da Web definidos por outros desenvolvedores em suas próprias páginas da Web, depois explicam cada uma das três tecnologias nas quais os componentes da Web se baseiam e, finalmente, unem as três em um exemplo que implementa o elemento da caixa de pesquisa ilustrado na Figura 15-3.

15.6.1 Usando componentes da Web

Os componentes da Web são definidos em JavaScript, portanto, para usar um componente da Web em seu arquivo HTML, você precisa incluir o arquivo JavaScript que define o componente. Como os componentes da Web são uma tecnologia relativamente nova, eles geralmente são escritos como módulos JavaScript, portanto, você pode incluir um em seu HTML assim:

```
<tipo de script="módulo" src="componentes/search-box.js">
```

Os componentes da Web definem seus próprios nomes de tags HTML, com a restrição importante de que esses nomes de tags devem incluir um hífen. (Este

significa que versões futuras do HTML podem introduzir novas tags sem hífens, e não há chance de que as tags entrem em conflito com o componente da web de alguém.) Para usar um componente da web, basta usar sua tag em seu arquivo HTML:

```
<espaço reservado da caixa de pesquisa="Pesquisar..."></search-box>
```

Os componentes da Web podem ter atributos da mesma forma que as tags HTML comuns; A documentação do componente que você está usando deve informar quais atributos são suportados. Os componentes da Web não podem ser definidos com tags de fechamento automático. Você não pode escrever <search-box/>, por exemplo. Seu arquivo HTML deve incluir a tag de abertura e a tag de fechamento.

Como elementos HTML regulares, alguns componentes da web são escritos para esperar filhos e outros são escritos de forma que não esperem (e não exibam) filhos. Alguns componentes da web são escritos para que possam aceitar opcionalmente crianças especialmente rotuladas que aparecerão em "slots" nomeados. O <search-box> componente ilustrado na Figura 15-3 e implementado no Exemplo 15-3 usa "slots" para os dois ícones que ele exibe. Se você quiser usar um <search-box> com ícones diferentes, você pode usar HTML assim:

```
<search-box>
</search-box>
```

O atributo slot é uma extensão do HTML que é usada para especificar quais crianças devem ir para onde. Os nomes dos slots — "left" e "right" neste exemplo — são definidos pelo componente Web. Se o componente

você está usando slots de suporte, esse fato deve ser incluído em sua documentação.

Observei anteriormente que os componentes da web são frequentemente implementados com módulos JavaScript e podem ser carregados em arquivos HTML com uma tag <script type="module">. Você deve se lembrar desde o início deste capítulo que os módulos são carregados depois que o conteúdo do documento é analisado, como se tivessem uma tag adiada. Portanto, isso significa que um navegador da web normalmente analisa e renderiza tags como <search-box> antes de executar o código que lhe dirá o que <search-box> é a. Isso é normal ao usar componentes da Web. Os analisadores HTML em navegadores da web são flexíveis e muito tolerantes com a entrada que eles não entendem. Quando eles encontram uma tag de componente da Web antes que esse componente tenha sido definido, eles adicionam um HTMLElement genérico à árvore DOM, mesmo que não saibam o que fazer com ele. Posteriormente, quando o elemento personalizado é definido, o elemento genérico é "atualizado" para que tenha a aparência e o comportamento desejados.

Se um componente da Web tiver filhos, esses filhos provavelmente serão exibidos incorretamente antes que o componente seja definido. Você pode usar este CSS para manter os componentes da web ocultos até que sejam definidos:

```
/*
 * Torne o <search-box> componente invisível antes de ser
 * definido.
 * E tente duplicar seu eventual layout e tamanho para
 * quenas proximidades
 * o conteúdo não se move quando é
 * definido.*/
<search-box:not(:defined) {
    opacidade: 0; exibição:
    bloco embutido;
```

```
largura: 300px;  
altura: 50px;}
```

Como os elementos HTML regulares, os componentes da web podem ser usados em JavaScript. Se você incluir uma <search-box> tag em sua página da web, poderá obter uma referência a ela com `querySelector()` e um seletor CSS apropriado, assim como faria com qualquer outra tag HTML. Geralmente, só faz sentido fazer isso depois que o módulo que define o componente for executado, portanto, tenha cuidado ao consultar componentes da web que você não faça isso muito cedo. Implementações de componentes da Web normalmente (mas isso não é um requisito) definem uma propriedade JavaScript para cada atributo HTML que eles suportam. E, como os elementos HTML, eles também podem definir métodos úteis. Mais uma vez, a documentação do componente da Web que você está usando deve especificar quais propriedades e métodos estão disponíveis para o código JavaScript.

Agora que você sabe como usar componentes da web, as próximas três seções cobrem os três recursos do navegador da web que nos permitem implementá-los.

NÓS DE DOCUMENTFRAGMENT

Antes de podermos abordar as APIs de componentes da Web, precisamos retornar brevemente à API do DOM para explicar o que é a `DocumentFragment`. A API DOM organiza um documento em uma árvore de objetos `Node`, onde `aNode` pode ser um Documento, um Elemento, um nó de Texto ou até mesmo um Comentário. Nenhum desses tipos de nó permite que você represente um fragmento de um documento que consiste em um conjunto de nós irmãos sem seu pai. É aqui que entra o `DocumentFragment`: é outro tipo de nó que serve como pai temporário quando você deseja manipular um grupo de nós irmãos como uma única unidade. Você pode criar um nó `DocumentFragment` com `document.createDocumentFragment()`. Depois de ter um `DocumentFragment`, você pode usá-lo como um Element e anexar () conteúdo a ele. O `DocumentFragment` é diferente de um Element porque não tem um pai. Mas, o mais importante, quando você insere um nó `DocumentFragment` no documento, o próprio `DocumentFragment` não é inserido. Em vez disso, todos os seus filhos são inseridos.

15.6.2 Modelos HTML

A <template> tag HTML está apenas vagamente relacionada a componentes da web, mas permite uma otimização útil para componentes que aparecem com frequência em páginas da web. <template> tags e seus filhos nunca são renderizados por um navegador da web e são úteis apenas em páginas da web que usam JavaScript. A ideia por trás dessa tag é que, quando uma página da web contém várias repetições da mesma estrutura HTML básica (como linhas em uma tabela ou a implementação interna de um componente da web), podemos usar a <template> para definir essa estrutura de elemento uma vez e, em seguida, usar JavaScript para duplicar a estrutura quantas vezes forem necessárias.

Em JavaScript, uma <template> marca é representada por um objeto `HTMLTemplateElement`. Esse objeto define uma única `contentproperty`, e o valor dessa propriedade é um `DocumentFragment` de todos os nós filho do <template>. Você pode clonar este `DocumentFragment` e inserir a cópia clonada em seu documento conforme necessário. O fragmento em si não será inserido, mas seus filhos serão. Suponha que você esteja trabalhando com um documento que inclui uma <table> tag e <template id="row"> e que o modelo define a estrutura das linhas dessa tabela. Você pode usar o modelo assim:

```
let tableBody = document.querySelector("tbody"); let
template = document.querySelector("#row"); let clone =
template.content.cloneNode(true); // clone profundo // ... Use
o DOM para inserir conteúdo nos <td> elementos do clone... //
Agora adicione a linha clonada e inicializada na
tabletbody.append(clone);
```

Os elementos do modelo não precisam aparecer literalmente em um documento HTML para serem úteis. Você pode criar um modelo em seu Código JavaScript, criar seus filhos com innerHTML e, em seguida, fazer quantos clones forem necessários sem a sobrecarga de análise de innerHTML. É assim que os modelos HTML são normalmente usados em webcomponents, e o Exemplo 15-3 demonstra essa técnica.

15.6.3 Elementos personalizados

O segundo recurso do navegador da Web que habilita os componentes da Web são os "elementos personalizados": a capacidade de associar uma classe JavaScript a um nome de tag HTML para que essas tags no documento sejam automaticamente transformadas em instâncias da classe na árvore DOM. O método customElements.define() usa um nome de tag de componente da Web como seu primeiro argumento (lembre-se de que o nome da tag deve incluir ahyphen) e uma subclasse de HTMLElement como seu segundo argumento. Quaisquer elementos existentes no documento com esse nome de tag são "atualizados" para instâncias recém-criadas da classe. E se o navegador analisar qualquerHTML no futuro, ele criará automaticamente uma instância da classe para cada uma das tags que encontrar.

A classe passada para customElements.define() deve extendHTMLElement e não um tipo mais específico como HTMLButtonElement. Lembre-se do Capítulo 9 que quando uma classe JavaScript estende outra classe, a função construtora deve chamar super() antes de usar a palavra-chave this, portanto, se a classe de elemento personalizada tiver um construtor, ela deve chamar super() (sem argumentos) antes de fazer qualquer outra coisa.⁴

O navegador invocará automaticamente certos "métodos de ciclo de vida" de uma classe de elemento personalizado. O método `connectedCallback()` é invocado quando uma instância do elemento personalizado é inserida no documento, e muitos elementos usam esse método para executar inicialização. Há também um método `disconnectedCallback()` invocado quando (e se) o elemento é removido do documento, embora isso seja usado com menos frequência.

Se uma classe de elemento personalizado definir uma propriedade `observedAttributes` estática cujo valor é uma matriz de nomes de atributos, e se qualquer um dos atributos nomeados for definido (ou alterado) em uma instância do elemento `custom`, o navegador invocará o método `attributeChangedCallback()`, passando o nome do atributo, seu valor antigo e seu novo valor. Esse retorno de chamada pode executar as etapas necessárias para atualizar o componente com base em seus valores de atributo.

As classes de elementos personalizados também podem definir quaisquer outras propriedades e métodos que desejarem. Normalmente, eles definirão getter e setter métodos que disponibilizam os atributos do elemento como JavaScript properties.

Como exemplo de um elemento personalizado, suponha que queremos ser capazes de exibir círculos dentro de parágrafos de texto normal. Gostaríamos de ser capazes de escrever HTML como este para renderizar problemas matemáticos de história como o mostrado na Figura 15-4:

```
<p>
  O documento tem uma bolinha de gude: <inline-circle>
</inline-circle>
```

```
O analisador HTML instancia mais duas bolinhas de gude:  
<inline-circle diameter="1.2em" color="blue"></inline-  
circle>  
<diâmetro do círculo embutido = ".6em" cor = "ouro" >  
</inline-circle>. Quantas bolinhas de gude o documento contém  
agora?</p>
```

The document has one marble: . The HTML parser instantiates two more marbles:  . How many marbles does the document contain now?

Figura 15-4. Um elemento personalizado de círculo embutido

Podemos implementar esse `<inline-circle>` elemento personalizado com o código mostrado no Exemplo 15-2:

Exemplo 15-2. O `<inline-circle>` elemento personalizado

```
customElements.define("inline-circle", class  
  InlineCircle extends HTMLElement {  
    O navegador chama esse método quando um <inline-  
circle> elemento  
    é inserido no documento. Há também  
    a disconnectedCallback()  
    que não precisamos neste  
    exemplo.connectedCallback() {  
      Defina os estilos necessários para criar  
      circle  
      this.style.display = "inline-  
block"; this.style.borderRadius = "50%";  
      this.style.border = "preto sólido 1px";  
      this.style.transform = "translateY(10%)";
```

```
Se ainda não houver um tamanho definido, defina um
tamanho padrão
que se baseia no tamanho da fonte
atual.if (!this.style.width) {
    this.style.width = "0.8em";
    this.style.height = "0.8em";}}
```

A propriedade `observedAttributes` est谩tica especifica
whichAttributes
Queremos ser notificados sobre as altera珲es no. (Usamos
a getter aqui desde
s o podemos usar "est tico" com m etodos.)`static get
observedAttributes() { return ["diâmetro", "cor"]; }`

Esse retorno de chamada  e invocado quando um dos
atributos listados acima
altera珲es, quando o elemento personalizado  e analisado pela
primeira vez ou posteriormente.

```
attributeChangedCallback(nome, oldValue, newValue) {
    switch (nome)
        {case "diâmetro":
            Se o atributo de diâmetro for alterado, atualize o
            Estilos de tamanho
            this.style.width = novoValor;
            this.style.height = novoValor;
            quebrar; mai sculas e min sculas
            "cor":
            Se o atributo de cor for alterado, atualize a cor
            Estilos
            this.style.backgroundColor = newValue;
            quebra;}}
```

Defina as propriedades JavaScript que correspondem ao
elemento
Atributos. Esses getters e setters apenas obtêm e definem o
subjacente
Atributos. Se uma propriedade JavaScript for definida, isso
define o atributo
que aciona uma chamada para `attributeChangedCallback()`

```
quais atualizações
o elemento styles.get diameter() { return
this.getAttribute("diameter"); }set diâmetro(diâmetro) {
this.setAttribute("diâmetro",diâmetro); }

get color() { return this.getAttribute("color"); }set
color(color) { this.setAttribute("color", color); }});
```

15.6.4 Shadow DOM

O elemento personalizado demonstrado no Exemplo 15-2 não está bem encapsulado. Quando você define seus atributos de diâmetro ou cor, ele responde alterando seu próprio atributo de estilo, o que não é um comportamento que jamais esperaríamos de um elemento HTML real. Para transformar um `customelement` em um verdadeiro componente da web, ele deve usar o poderoso mecanismo de encapsulamento conhecido como shadow DOM.

O Shadow DOM permite que uma "raiz de sombra" seja anexada a um elemento `custom`(e também a um `<div>`elemento , , `<body>`, `<article>`, `<main><nav>`, , , `<header><footer>``<section><p>` `<blockquote><aside>`ou `<h1>` through `<h6>`) conhecido como "host de sombra". Os elementos do host de sombra, como todos os elementos HTML, já são a raiz de uma árvore DOM normal de elementos descendentes e nós de texto. Uma raiz de sombra é a raiz de outra árvore de elementos descendentes, mais privada, que brota do hospedeiro de sombra e pode ser pensada como um minidocumento distinto.

A palavra "sombra" em "shadow DOM" refere-se ao fato de que os elementos que descendem de uma raiz de sombra estão "escondidos nas sombras": eles não fazem parte da árvore normal do DOM, não aparecem nos filhos

array de seu elemento host e não são visitados por métodos DOMtraversal normais, como querySelector(). Por outro lado, os filhos normais e regulares do DOM de um hospedeiro sombra às vezes são chamados de "DOM leve".

Para entender o propósito do shadow DOM, imagine o HTML<audio> e os <video> elementos: eles exibem uma interface de usuário não trivial para controlar a reprodução de mídia, mas os botões reproduzir e pausar e outros elementos da interface do usuário não fazem parte da árvore do DOM e não podem ser manipulados pelo JavaScript. Dado que os navegadores da web são projetados para exibir HTML, é natural que os fornecedores de navegadores queiram exibir interfaces de usuário internas como essas usando HTML. Na verdade, a maioria dos navegadores faz algo assim há muito tempo, e o shadowDOM o torna uma parte padrão da plataforma web.

SHADOW DOM CAPSULATIONA principal característica do shadow DOM é o encapsulamento que ele fornece. Os descendentes de uma raiz de sombra estão ocultos e independentes da árvore DOM regular, quase como se estivessem em um documento independente. Existem três tipos muito importantes de encapsulamento fornecidos pelo shadow DOM:

- Como já mencionado, os elementos no shadow DOM são ocultos dos métodos DOM regulares como querySelectorAll(). Quando uma raiz de sombra é criada e anexada ao seu hospedeiro de sombra, ela pode ser criada no modo "aberto" ou "fechado". Uma raiz de sombra fechada é completamente selada e inacessível. Mais comumente, porém, as raízes de sombra são criadas no modo "aberto", o que significa que o host de sombra tem uma propriedade shadowRoot que o JavaScript pode usar para obter

- acesso aos elementos da raiz da sombra, se tiver algum motivo para fazê-lo. Os estilos definidos abaixo de uma raiz de sombra são privados para essa árvore e nunca afetarão os elementos DOM claros do lado de fora. (Uma raiz de sombra pode definir estilos padrão para seu elemento host, mas eles serão substituídos por estilos DOM claros.) Da mesma forma, os estilos DOM claros que se aplicam ao elemento hospedeiro de sombra não têm efeito sobre os descendentes da raiz de sombra. Os elementos no shadow DOM herdarão coisas como tamanho da fonte e cor de fundo do DOM claro, e os estilos no shadow DOM podem optar por usar variáveis CSS definidas no light DOM. Na maioria das vezes, no entanto, os estilos do lightDOM e os estilos do shadow DOM são completamente independentes: o autor de um componente da web e o usuário de um componente da web não precisam se preocupar com colisões ou conflitos entre suas folhas de estilo. Ser capaz de "escopo" CSS dessa maneira é talvez o recurso mais importante do shadow DOM. Alguns eventos (como "load") que ocorrem dentro do DOM de sombra estão confinados ao DOM de sombra. Outros, incluindo eventos de foco, mouse e teclado, surgem e desaparecem. Quando um evento que se origina no shadow DOM cruza o limite e começa a se propagar no DOM light, sua targetproperty é alterada para o elemento shadow host, então parece ter se originado diretamente nesse elemento.

SLOTS SHADOW DOM E LIGHT DOM CHILDREN Um elemento HTML que é um host de sombra tem duas árvores de descendentes. Um é o array `children[]` - os descendentes DOM leves regulares do elemento host - e o outro é a raiz da sombra e todos os seus descendentes, e você pode estar se perguntando como duas árvores de conteúdo distintas podem ser exibidas no mesmo elemento host. Veja como funciona:

- Os descendentes da raiz da sombra são sempre exibidos dentro do host da sombra. Se esses descendentes incluírem um <slot> elemento, os filhos DOM leves regulares do elemento host serão exibidos como se fossem filhos desse <slot>, substituindo qualquer conteúdo DOM de sombra no slot. Se o shadow DOM não incluir um <slot>, então qualquer conteúdo DOM claro do host nunca será exibido. Se o shadow DOM tiver um <slot>, mas o shadow host não tiver filhos DOM claros, o conteúdo shadowDOM do slot será exibido como padrão. Quando o conteúdo DOM claro é exibido em um slot DOM de sombra, dizemos que esses elementos foram "distribuídos", mas é importante entender que os elementos não se tornam realmente parte do DOM de sombra. Eles ainda podem ser consultados com querySelector() e ainda aparecem no lightDOM como filhos ou descendentes do elemento host. Se o shadow DOM definir mais de um <slot> e nomear esses slots com um atributo name, os filhos do shadow host poderão especificar em qual slot eles gostariam de aparecer especificando um atributo slot="slotname". Vimos um exemplo desse uso no §15.6.1 quando demonstramos como personalizar os ícones exibidos pelo <search-box> componente.

API SHADOW DOM Para todo o seu poder, o Shadow DOM não tem muito de uma API JavaScript. Para transformar um elemento DOM light em um host de sombra, basta chamar seu método attachShadow(), passando {mode:"open"} como o único argumento. Esse método retorna um objeto raiz de sombra e também define esse objeto como o valor da propriedade shadowRoot do host. O

shadow root é um DocumentFragment, e você pode usar DOMmethods para adicionar conteúdo a ele ou apenas definir sua propriedade innerHTML como uma string de HTML.

Se o seu componente da Web precisar saber quando o conteúdo do DOM claro de um shadow DOM <slot> foi alterado, ele poderá registrar um ouvinte para eventos "slotchanged" diretamente no <slot> elemento.

15.6.5 Exemplo: um <search-box> componente Web

A Figura 15-3 ilustrou um <search-box> componente da web. O exemplo 15-3 demonstra as três tecnologias habilitadoras que definem os componentes da web: ele implementa o <search-box> componente como um elemento personalizado que usa uma <template> tag para eficiência e shadow root para encapsulamento.

Este exemplo mostra como usar as APIs do componente Web de baixo nível diretamente. Na prática, muitos componentes da web desenvolvidos hoje os criam usando bibliotecas de nível superior, como "lit-element". Um dos razões para usar uma biblioteca é que criar componentes reutilizáveis e personalizáveis é realmente muito difícil de fazer bem, e há muitos detalhes para acertar. O exemplo 15-3 demonstra componentes da Web e faz algum tratamento básico de foco do teclado, mas ignora a acessibilidade e não faz nenhuma tentativa de usar atributos ARIA adequados para fazer o componente funcionar com leitores de tela e outras tecnologias assistivas.

Exemplo 15-3. Implementando um componente Web

```
/**
```

```
* Esta classe define um elemento HTML personalizado <search-box> que
```

exibe um

- * <input> Campo de entrada de texto mais dois ícones ou emoji. Por padrão, ele exibe um*
- * Emoji de lupa (indicando pesquisa) à esquerda do campo de texto*
- * e um emoji X (indicando cancelar) à direita do campo de texto. Ela*
- * oculta a borda no campo de entrada e exibe uma borda em torno de si mesma,*
- * criando a aparência de que os dois emojis estão dentro da entrada.*
- *campo. Da mesma forma, quando o campo de entrada interno está focado, o anel de foco*
- * é exibido ao redor do <search-box>.** Você pode substituir os ícones padrão incluindo ou filhos*

** de <search-box> com slot="left" e slot="right" atributos.*

*** <search-box> suporta os atributos HTML normais desabilitados e ocultos e*

** também atributos de tamanho e espaço reservado, que têm o mesmo significado para isso*

** como fazem para o <input> elemento.** Eventos de entrada do <input> elemento interno borbulham para cima e aparecem com*

** seu campo de destino definido como o <search-box> elemento.** O elemento dispara um evento de "pesquisa" com a propriedade detail definida como o elemento*

** string de entrada atual quando o usuário clica no emoji esquerdo (a ampliação*

** vídro). O evento "search" também é despachado quando o campo de texto interno*

** gera um evento "change" (quando o texto foi alterado e o usuário digita*

** Return ou Tab).** O elemento dispara um evento "clear" quando o usuário clica no emoji direito* (o X). Se nenhum manipulador chamar preventDefault() no evento, o elemento*

** limpa a entrada do usuário assim que o despacho do evento é concluído.** Observe que não há propriedades ou atributos onsearch e onclear:*

```
* Os manipuladores para os eventos "search" e "clear" só  
podem ser registrados com  
* addEventListener().*/class  
SearchBox extends HTMLElement {  
  
    construtor() {  
        super(); Invocar o construtor de superclasse; deve ser  
primeiro.  
  
        Crie uma árvore DOM sombra e anexe-a a ela  
elemento, configuração  
o valor de  
this.shadowRoot.this.attachShadow({mode: "open"});  
  
        Clone o modelo que define os descendentes e  
folha de estilo para  
esse componente personalizado e anexar esse conteúdo a  
a raiz da sombra.  
  
        this.shadowRoot.append(SearchBox.template.content.cloneNode(true));  
  
        Obtenha referências aos elementos importantes no  
sombra DOM  
        this.input = this.shadowRoot.querySelector("#input");  
        let leftSlot =  
this.shadowRoot.querySelector('slot[name="left"]');  
        let rightSlot =  
this.shadowRoot.querySelector('slot[name="right"]');  
  
        Quando o campo de entrada interno recebe ou perde o foco,  
Definir ou remover  
o atributo "focused" que fará com que nosso  
Folha de estilo interna  
para exibir ou ocultar um anel de foco falso em todo o  
componente. Nota  
que os eventos "blur" e "focus" borbulham e apareçam  
originar  
do <search-  
box>.this.input.onfocus = () => {  
this.setAttribute("focado", ""); };  
        this.input.onblur = () => {  
this.removeAttribute("focado");};  
  
        Se o usuário clicar na lupa, acione
```

```
uma "pesquisa"
    acontecime Também o acione se o campo de entrada disparar um
"mudanç nto.
a"      acontecimento. (O evento "mudança" não sai de
o Shadow DOM.)
    leftSlot.onclick = this.input.onchange = (evento) => {
        evento.stopPropagation();   Impedir eventos de clique
de borbulhar
        if (this.disabled)           Não faça nada quando
desactivado retornar;
        this.dispatchEvent(new CustomEvent("pesquisar", {
            detalhe: this.input.value}));};
```

Se o usuário clicar no X, acione um "limpar" acontecimento.

Se preventDefault() não for chamado no evento, Limpe a entrada.

```
rightSlot.onclick = (evento) => {
    evento.stopPropagation();   Não deixe o clique
bolha para cima
    if (this.disabled)           Não faça nada se
desactivado retornar;
    let e = new CustomEvent("clear", { cancelable: true
});;
    this.dispatchEvent(e); if
    (!e.defaultPrevented) {      Se o evento não foi
"cancelado"
        this.input.value = "";   em seguida, limpe a entrada
campo
    }};}
```

*Quando alguns de nossos atributos são definidos ou alterados, precisamos definir o valor correspondente no elemento interno <input> . Este ciclo de vida , juntamente com a propriedade estática observedAttributes abaixo, cuida disso.attributeChangedCallback(nome,
oldValue, newValue) {*

```

        if (nome === "desabilitado") {
            this.input.disabled = newValue !== null;}
        else if (nome === "espaço reservado") {
            this.input.placeholder = novoValor;}
        else if (nome === "tamanho") {
            this.input.size = novoValor;}
        else if (nome === "valor") {
            this.input.value = newValue;}}
    
```

Por fim, definimos getters e setters de propriedade para propriedades que correspondem aos atributos HTML que suportamos. Os getters simplesmente retornam o valor (ou a presença) do atributo. E esses jogadores acabaram de definir

o valor (ou a presença) do atributo. Quando o método asetter altera um atributo, o navegador invocará automaticamente o

attributeChangedCallback acima.

```

get placeholder() {
    return this.getAttribute("espaço reservado"); }

get size() { return this.getAttribute("size"); }get
value() { return this.getAttribute("value"); }get
disabled() { return this.hasAttribute("disabled"); }get
hidden() { return this.hasAttribute("hidden"); }

set placeholder(valor) { this.setAttribute("espaço
reservado", valor); }

set size(value) { this.setAttribute("size", value);
}set value(text) { this.setAttribute("value", text);
}set disabled(valor) {
    if (valor) this.setAttribute("desabilitado",
    ""); else
    this.removeAttribute("desabilitado");}
set
oculto(valor) {
    if (valor) this.setAttribute("oculto",
    ""); else this.removeAttribute("oculto");}
    
```

```
}}
```

Este campo estático é necessário para o método attributeChangedCallback.// Somente os atributos nomeados nesta matriz acionarão chamadas para esse método.
`SearchBox.observedAttributes = ["desativado", "espaço reservado","tamanho", "valor"];`

Crie um <template> elemento para conter a folha de estilo e a árvore de// elementos que usaremos para cada instância do SearchBoxelement.
`SearchBox.template = document.createElement("template");`

Inicializamos o modelo analisando essa string de HTML. Observe, no entanto,// que quando instanciamos um SearchBox, podemos apenas clonar os nós// no modelo e ter que analisar o HTML novamente.
`SearchBox.template.innerHTML = '<style>/*`

```
* O seletor :host refere-se ao <search-box> elemento na luz
* DOM. Esses estilos são padrões e podem ser substituídos
pelo usuário do
* <search-box> com estilos no DOM
leve.*:host {

    exibição: bloco embutido; /* O padrão é exibição embutida */
    borda: preto sólido 1px; /* Uma borda arredondada ao
redor do<input> e <slots> */
    raio da borda: 5px;
    preenchimento: 4px 6px; /* E algum espaço dentro da borda
*/}:host([oculto
]) {
                                /* Observe os parênteses: quando o host
escondeu... */
    exibição:nenhuma;        /* ... conjunto de atributos não o exibe
*/}:host([desativa
do]) {
                                /* Quando o host tem o arquivo desativado
atributo...*/
    opacidade: 0,5;           /* ... acinzentado */
```

```

}:host([focado])
{
    /* Quando o host tem o elemento
atributo...*/
    sombra da caixa: 0 0 2px 2px #6AE;    /* exibir este foco falso
anel.
*}

/* O restante da folha de estilo se aplica apenas aos
elementos no Shadow DOM. */input {

    largura da borda: 0;        /* Ocultar a borda do interno
campo de entrada. */
    contorno: nenhum;          /* Esconda o anel de foco também. */
    fonte: herdar;              /* <input> elementos não herdam
fonte por padrão */
    fundo: herdar;              /* O mesmo para a cor de fundo. */
}slot
{
    cursor: padrão;            /* Um cursor de seta sobre o
Botões*/
    user-select: nenhum;        /* Não deixe o usuário selecionar o
texto emoji
*}
</style>
<div>

    <nome do slot = "esquerda">\u{1f50d}</slot><!-- U+1F50D é um
Lupa -->
    <tipo de entrada="texto" id="entrada" />    <!-- A entrada real
elemento -->
    <nome do slot = "right">\u{2573}</slot><!-- U+2573 é um X -->
</div
>';

```

Por fim, chamamos `customElement.define()` para registrar `oSearchBox` element// como a implementação da `<search-box>` tag. Customelements são necessários// para ter um nome de tag que contenha um hyphen.`customElements.define("search-box", SearchBox);`

15.7 SVG: Gráficos vetoriais escaláveis

SVG (gráficos vetoriais escaláveis) é um formato de imagem. A palavra "vetor" em seu nome indica que é fundamentalmente diferente de raster

formatos de imagem, como GIF, JPEG e PNG, que especificam uma matriz de valores de pixel. Em vez disso, uma "imagem" SVG é uma descrição precisa, independente da resolução (portanto, "escalável") das etapas necessárias para desenhar o gráfico desejado. As imagens SVG são descritas por arquivos de texto usando a linguagem de marcação XML, que é bastante semelhante ao HTML.

Existem três maneiras de usar o SVG em navegadores da web:

1. Você pode usar arquivos de imagem .svg com tags HTML regulares , assim como usaria uma imagem .png ou .jpeg.
2. Como o formato SVG baseado em XML é tão semelhante ao HTML, você pode incorporar tags SVG diretamente em seus documentos HTML. Se você fizer isso, o analisador HTML do navegador permitirá você omitir namespaces XML e tratar tags SVG como se fossem tags HTML.
3. Você pode usar a API DOM para criar dinamicamente elementos SVG para gerar imagens sob demanda. As subseções a seguir demonstram o segundo e o terceiro usos do SVG. Observe, no entanto, que o SVG tem uma gramática grande e moderadamente complexa. Além de primitivos simples de desenho de forma, ele inclui suporte para curvas, texto e animação arbitrários. Os gráficos SVG podem até incorporar scripts JavaScript e folhas de estilo CSS para adicionar informações de comportamento e apresentação. Uma descrição completa do SVG está muito além do escopo deste livro. O objetivo desta seção é apenas mostrar como você pode usar SVG em seus documentos HTML e scriptá-lo com JavaScript.

15.7.1 SVG em HTML

As imagens SVG podem, é claro, ser exibidas usando tags HTML. Mas você também pode incorporar SVG diretamente em HTML. E se você fizer isso, poderá até usar folhas de estilo CSS para especificar coisas como fontes, cores e larguras de linha. Aqui, por exemplo, está um arquivo HTML que usa SVG para exibir um mostrador de relógio analógico:

```
<html><head><title>Relógio analógico</title><style>/*
Todos
esses estilos CSS se aplicam aos elementos SVG definidos
abaixo */#clock {/* Estilos para tudono relógio:*/
    traço: preto; /* linhas pretas */
    stroke-linecap: redondo; /* com extremidades arredondadas */
    preenchimento: #ffe; /* em um esbranquiçado
fundo */#clock .face { largura
do traço: 3; }

#clock .ticks {largura do traço: 2; } /* Contorno do mostrador do relógio */
hora */#clock .hands { largura do
traço: 3; } /* Linhas que marcam cada
mãos */#clock
.numbers { /* Como desenhar o relógio
Números*/
família de fontes: sans-serif; tamanho da fonte: 10;
peso da fonte: negrito;
âncora de texto: meio; acidente vascular cerebral:
nenhum; preenchimento: preto;}</style></head>
<body>

<svg id="clock" viewBox="0 0 100 100"
width="250"height="250">
<!-- Os atributos de largura e altura são o tamanho da tela
do gráfico -->
<!-- O atributo viewBox fornece o sistema de coordenadas
interno -->
    <circle class="face" cx="50" cy="50" r="45"/> <!-- o
mostrador do relógio -->
```

```

<g class="ticks">    <!-- marcas de verificação para cada um dos 12
horas -->
<linha x1='50' y1='5.000' x2='50.00' y2='10.00' /><linha
x1='72.50' y1='11.03' x2='70.00' y2='15.36' /><linha
x1='88.97' y1='27.50' x2='84.64' y2='30.00' /><linha
x1='95.00' y1='50.00' x2='90.00.00' y2='50,00' /><linha
x1='88,97' y1='72,50' x2='84,64' y2='70,00' /><linha
x1='72,50' y1='88,97' x2='70,00' y2='84,64' /><linha
x1='50,00' y1='95,00' x2='50,00' y2='90,00' /><linha
x1='27,50' y1='88,97' x2='30,00' y2='84,64' /><linha
x1='11,03' y1='72,50' x2='15,36' y2='70,00' /><linha
x1='5.000' y1='50.00' x2='10.00' y2='50.00' /><linha
x1='11.03' y1='27.50' x2='15.36' y2='30.00' /><linha
x1='27.50' y1='11.03' x2='30.00' y2='15.36' /></g><g
class="numbers"> <!-- Numere as direções cardinais -->

<texto x="50" y="18">12</text><texto
x="85"y="53">3</text>
<texto x = "50" y = "88" >6</text><texto x =
"15" y = "53">9</text>
</g><g
class="hands">      <!-- Desenhe as mãos apontando para cima.
--><line class="ponteiro das horas" x1="50" y1="50"
x2="50"y2="25" /><line class="ponteiro dos minutos"
x1="50" y1="50" x2="50"y2="20" /></g></svg><script
src="clock.js"></script></body></html>

```

Você notará que os descendentes da `<svg>` tag não são tags HTML normais. `<circle>`, `<line>`, e `<text>` tags têm propósitos óbvios, no entanto, e deve ficar claro como esse gráfico SVG funciona. Existem muitas outras tags SVG, no entanto, e você precisará consultar uma referência SVG para saber mais. Você também pode notar que a folha de estilo é estranha. Estilos como `fill`, `stroke-width` e `text-anchor` não são propriedades normais de estilo CSS. Nesse caso, CSS é

essencialmente sendo usado para definir atributos de tags SVG que aparecem no documento. Observe também que a propriedade abreviada CSS font não funciona para tags SVG e você deve definir explicitamente font-family, font-size e font-weight como propriedades de estilo separadas.

15.7.2 Script SVG

Um motivo para incorporar SVG diretamente em seus arquivos HTML (em vez de apenas usar `` tags estáticas) é que, se você fizer isso, poderá usar a API DOM para manipular a imagem SVG. Suponha que você use SVG para exibir ícones em seu aplicativo da web. Você pode incorporar SVG em uma `<template>` tag (§15.6.2) e, em seguida, clonar o conteúdo do modelo sempre que precisar inserir uma cópia desse ícone em sua interface do usuário. E se você quiser que o ícone responda à atividade do usuário - mudando de cor quando o usuário passa o ponteiro sobre ele, por exemplo - muitas vezes você pode conseguir isso com CSS.

Também é possível manipular dinamicamente gráficos SVG que são diretamente incorporados em HTML. O exemplo do mostrador do relógio na seção anterior exibe um relógio estático com os ponteiros das horas e dos minutos voltados para cima, exibindo a hora do meio-dia ou da meia-noite. Mas você deve ter notado que o arquivo HTML inclui uma `<script>` tag. Esse script executa uma função periodicamente para verificar a hora e transformar os ponteiros das horas e dos minutos, girando-os o número apropriado de graus para que o relógio realmente exiba a hora atual, conforme mostrado na Figura 15-

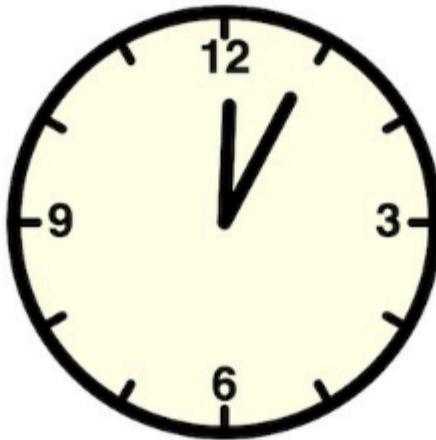


Figura 15-5. Um relógio analógico SVG com script

O código para manipular o relógio é direto. Ele determina o ângulo adequado dos ponteiros das horas e dos minutos com base na hora atual, em seguida, usa querySelector() para procurar os elementos SVG que exibem esses ponteiros e, em seguida, define um atributo de transformação neles para girá-los em torno do centro do mostrador do relógio. A função usessetTimeout() para garantir que ela seja executada uma vez por minuto:

```
(function updateClock() { // Atualiza o gráfico do relógio
  SVG para mostrar a hora atual
  let now = new Data();                               Atual
  Hora
  let seg = now.getSeconds();                         Segundos
  let min = now.getMinutes() + sec/60;               Fracionário
  ata
  let hora = (now.getHours() % 12) + min/60;         Fracionário
  Horas
  minangle fácil = min * 6;                          6 graus
  por minuto
  deixe hourangle = hora * 30;                      30 graus
  por hora}
```

```
Obter elementos SVG para os ponteiros do clocklet
minhand =
document.querySelector("#clock.minutehand");
let hourhand =
document.querySelector("#clock.hourhand");

Defina um atributo SVG neles para movê-los ao redor do
mostrador do relógio
minhand.setAttribute("transformar",'
rotate(${minangle},50,50)');
hourhand.setAttribute("transform", 'ro
tate(${hourangle},50,50)');

Execute esta função novamente em 10
segundossetTimeout(updateClock, 10000);}());
Observe a
invocação imediata da função aqui.
```

15.7.3 Criando imagens SVG com JavaScript

Além de simplesmente criar scripts de imagens SVG incorporadas em seus documentos HTML, você também pode criar imagens SVG do zero, o que pode ser útil para criar visualizações de dados carregados dinamicamente, por exemplo. O exemplo 15-4 demonstra como você pode usar JavaScript para criar gráficos SVGpie, como o mostrado na Figura 15-6.

Embora as tags SVG possam ser incluídas em documentos HTML, elas são tecnicamente tags XML, não tags HTML, e se você quiser criar elementos SVG com a API JavaScript DOM, não poderá usar a função `normalcreateElement()` que foi introduzida em §15.3.5. Em vez disso, você deve usar `createElementNS()`, que usa uma string `XMLnamespace` como seu primeiro argumento. Para SVG, esse namespace é a cadeia de caracteres literal "`http://www.w3.org/2000/svg`".

Programming languages by percentage of professional developers who report their use

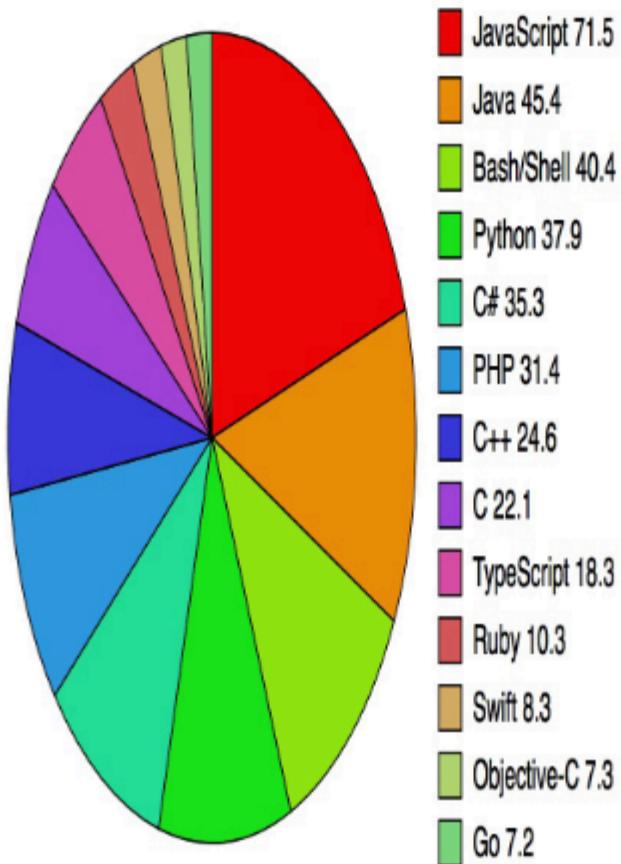


Figura 15-6. Um gráfico de pizza SVG construído com JavaScript (dados do Stack Overflow de 2018 Pesquisa de desenvolvedores das tecnologias mais populares)

Além do uso de `createElementNS()`, o código de desenho do gráfico de pizza no Exemplo 15-4 é relativamente simples. Há um pouco de matemática para converter os dados que estão sendo mapeados em ângulos de fatia de pizza. A maior parte

do exemplo, no entanto, é o código DOM que cria elementos SVG e define atributos nesses elementos.

A parte mais opaca deste exemplo é o código que desenha as fatias de torta real. O elemento usado para exibir cada fatia é <path>. Este elemento SVG descreve formas arbitrárias compostas por linhas e curvas. A descrição da forma é especificada pelo atributo d do <path>elemento. O valor desse atributo usa uma gramática compacta de códigos de letras e números que especificam coordenadas, ângulos e outros valores. A letra M, por exemplo, significa "mover para" e é seguida por x e ycoordinates. A letra L significa "linha para" e desenha uma linha do ponto atual para as coordenadas que o seguem. Este exemplo também usa a letra A para desenhar um arco. Esta letra é seguida por sete números descrevendo o arco, e você pode procurar a sintaxe online se quiser saber mais.

Exemplo 15-4. Desenhando um gráfico de pizza com JavaScript e SVG

```
/**
```

```
* Crie um <svg> elemento e desenhe um gráfico de pizza  
nele.** Esta função espera um argumento de objeto com as  
seguintes propriedades:
```

```
**width, height: o tamanho do gráfico SVG, em pixels*cx, cy,  
r: o centro e o raio da pizza*lx, ly: o canto superior  
esquerdo da legenda do gráfico*data: um objeto cujos nomes  
de propriedade são rótulos de dados e cujo
```

```
*           são os valores associados a cada  
etiqueta
```

```
** A função retorna um <svg> elemento. O chamador  
deve inseri-lo em
```

```
* o documento para torná-lo  
visível.*/function pieChart(options) {
```

```
let {largura, altura, cx, cy, r, lx, ly, data} = opções;
```

Este é o namespace XML para elementos svglet
svg = "http://www.w3.org/2000/svg";

Crie o <svg> elemento e especifique o tamanho do pixel e as coordenadas do usuário

```
let gráfico = document.createElementNS(svg, "svg");  
chart.setAttribute("largura", largura);  
chart.setAttribute("altura", altura);  
chart.setAttribute("viewBox", '0 0 ${width} ${height}');
```

Defina os estilos de texto que usaremos para o gráfico. Se deixarmos esses

valores não definidos aqui, eles podem ser definidos com CSS instead.
chart.setAttribute("font-family", "sans-serif"); chart.setAttribute("tamanho da fonte", "18");

Obtenha rótulos e valores como matrizes e some os valores para que saibamos como

```
grande a torta é.  
let labels =  
Object.keys(dados); let valores =  
Object.values(dados); let total =  
valores.reduce((x,y) => x+y);
```

Descubra os ângulos de todas as fatias. Slice i startsat
ângulos[i]
e termina em ângulos[i+1]. Os ângulos são medidos em
radianos.

```
let ângulos = [0]; valores.paraCada((x, i) =>  
ângulos.empurrar(ângulos[i] + x/total *2 * Matemática.PI));
```

Agora percorra as fatias do
pievalues.forEach((value, i) => {
Calcule os dois pontos onde nossa fatia se cruza
o círculo

Essas fórmulas são escolhidas de forma que um ângulo de 0 seja às 12 horas
e os ângulos positivos aumentam no sentido
horário.
let xl = cx + r * Math.sin(ângulos[i]);
seja yl = cy - r * Math.cos(ângulos[i]); seja
x2 = cx + r * Math.sin(ângulos[i+1]);

```
seja y2 = cy - r * Math.cos(ângulos[i+1]);
```

Esta é uma bandeira para ângulos maiores que um semicírculo// É exigida pelo componentlet de desenho de arco SVG big = (angles[i+1] - angles[i] > Math.PI) ? 1 : 0;

Esta string descreve como desenhar uma fatia da torta gráfico:

```
let caminho = 'M${cx},${cy}' + Mover para círculo centro.  
'L${x1},${y1}' + Desenhe uma linha para (x1, y1).  
'A${r},${r} 0 ${big} 1' + Desenhe um arco de raio r...  
'${x2},${y2}' + // ... terminando em para (x2, y2).  
"Z"; Fechar caminho de volta para (cx, cy).
```

Calcule a cor CSS para esta fatia. Esta fórmula funciona apenas para cerca de 15 cores. Portanto, não inclua mais de 15 fatias em um gráfico.

```
let cor = 'hsl(${(i*40)%360}, ${90-3*i}%, ${50+2*i}%)';
```

Descrevemos uma fatia com um <path> elemento. Nota createElementNS().

```
let slice = document.createElementNS(svg, "caminho");
```

Agora defina os atributos no <path> elementslice.setAttribute("d", path); // Defina o caminho para esta fatia
slice.setAttribute("preenchimento", Cor cor); Definir fatia
slice.setAttribute("traço", "preto"); Contorno
fatia em preto
slice.setAttribute("largura do traço", "1"); 1 pixel CSS grosso
chart.append(fatia); Adicionar fatia para o gráfico

Agora desenhe um pequeno quadrado correspondente para o ícone do keylet = document.createElementNS(svg, "rect"); icon.setAttribute("x", lx); // Posição

A praça

```

        icon.setAttribute("y", ly + 30*i);
        icon.setAttribute("largura", 20);           Dimensione o
quadrado
        icon.setAttribute("altura", 20);
        icon.setAttribute("preenchimento", cor); Mesmo preenchimento
cor como a fatia
        icon.setAttribute("traço", "preto");      Mesmo
esboço também.
        icon.setAttribute("largura do traço",
        "1"); chart.append(ícone);               Adicione ao
gráfico

E adicione um rótulo à direita do retângulo rótulo
= document.createElementNS(svg, "text");
label.setAttribute("x", lx + 30); // Posição
o texto
        label.setAttribute("y", ly + 30*i + 16);
        label.append('${rótulos[i]} ${valor}'); Adicionar texto a
etiqueta
Adicionachártuappend(xádfiúdo);

});

gráfico de
retorno;
}

```

O gráfico de pizza na Figura 15-6 foi criado usando a função pieChart() do Exemplo 15-4, assim:

```

document.querySelector("#chart").append(pieChart({
    largura: 640, altura: 400, Tamanho total do gráfico
    CX: 200, CY: 200, R: 180, Centro e raio do
    pé
    lx: 400, ly: 10,           Posição da lenda
    dados: {                  Os dados a serem mapeados
        "JavaScript": 71.5,
        "Java": 45.4, "Bash
        / Shell": 40.4,
        "Python": 37.9, "C
        #": 35.3,
    }
})

```

```
"PHP": 31.4, "C++":  
24.6, "C":  
22.1, "TypeScript":  
18.3, "Ruby": 10.3, "Swift":  
8.3, "Objective-C":  
7.3, "Go": 7.2, {}));
```

15.8 Gráficos em um <canvas>

O <canvas> elemento não tem aparência própria, mas cria uma superfície de desenho dentro do documento e expõe uma poderosa drawingAPI ao JavaScript do lado do cliente. A principal diferença entre a <canvas> API e o SVG é que com a tela você cria desenhos chamando métodos, e com o SVG você cria desenhos construindo uma árvore de elementos XML. Essas duas abordagens são igualmente poderosas: uma pode ser simulada com a outra. Na superfície, eles são bem diferentes, no entanto, e cada um tem seus pontos fortes e fracos. Um desenho SVG, por exemplo, é facilmente editado removendo elementos de sua descrição. Para remover um elemento do mesmo gráfico em um <canvas>, muitas vezes é necessário apagar o desenho e redesenharlo do zero. Como a API de desenho do Canvas é baseada em JavaScript e relativamente compacta (ao contrário da gramática SVG), ela está documentada com mais detalhes neste livro.

GRÁFICOS 3D EM UMA TELA

Você também pode chamar getContext() com a string "webgl" para obter um objeto de contexto que permite desenhar gráficos 3D usando a API WebGL. WebGL é uma API grande, complicada e de baixo nível que permite que os programadores JavaScript acessem a GPU, escrevam shaders personalizados e executem outros muito poderosos

operações gráficas. O WebGL não está documentado neste livro, no entanto: os desenvolvedores da Web são mais propensos a usar bibliotecas de utilitários construídas sobre o WebGL do que usar a API WebGL diretamente.

A maior parte da API de desenho do Canvas é definida não no <canvas>elemento em si, mas em um objeto de "contexto de desenho" obtido com o método getContext() da tela. Chame getContext() com o argumento "2d" para obter um objeto CanvasRenderingContext2D que você pode usar para desenhar gráficos bidimensionais na tela.

Como um exemplo simples da API do Canvas, o seguinte documento HTML usa <canvas> elementos e um pouco de JavaScript para exibir duas formas simples:

```
<p>Este é um quadrado vermelho: <canvas id="square" width=10height=10></canvas>. <p>Este é um círculo azul: <canvas id="circle" width=10height=10></canvas>. <script>let canvas = document.querySelector("#square");// Obtém firstcanvas elementlet context = canvas.getContext("2d");// Obtém 2Ddrawing contextcontext.fillStyle = "#f00";// Define fillcolor como redcontext.fillRect(0,0,10,10);// Preenche um quadrado
```

```
canvas = document.querySelector("#circle");           Segundo  
canvas elementcontext =                         Obtenha seu  
canvas.getContext("2d");  
context.beginPath();// Começa de novo  
"path"context.arc(5, 5, 5, 0, 2*Math.PI, true);// Adiciona  
um círculo ao pathcontext.fillStyle = "#00f";// Define a  
cor do bluefill
```

```
context.fill();
caminho<
/script>
```

Preencha o

Vimos que o SVG descreve formas complexas como um "caminho" de linhas e curvas que podem ser desenhadas ou preenchidas. A API do Canvas também usa a noção de um caminho. Em vez de descrever um caminho como uma sequência de letras e números, um caminho é definido por uma série de chamadas de método, como as invocações beginPath() e arc() no código anterior. Depois que a path é definido, outros métodos, como fill(), operam nesse caminho. Várias propriedades do objeto de contexto, como fillStyle, especificam como essas operações são executadas.

As subseções a seguir demonstram os métodos e propriedades da API do 2D Canvas. Grande parte do código de exemplo que se segue opera em uma variável c. Essa variável contém o objeto CanvasRenderingContext2D da tela, mas o código para inicializar essa variável às vezes não é mostrado. Para executar esses exemplos, você precisaria adicionar marcação HTML para definir uma tela com atributos apropriados de largura e altura e, em seguida, adicionar um código como este para inicializar a variável c:

```
let canvas = document.querySelector("#my_canvas_id");
let c = canvas.getContext('2d');
```

15.8.1 Caminhos e polígonos

Para desenhar linhas em uma tela e preencher as áreas delimitadas por essas linhas, você começa definindo um caminho. Um caminho é uma sequência de um ou mais subcaminhos. Um subcaminho é uma sequência de dois ou mais pontos conectados por segmentos de linha (ou, como veremos mais adiante, por segmentos de curva). Comece um novo

path com o método beginPath(). Comece um novo subcaminho com o método moveTo(). Depois de estabelecer o ponto inicial de um subcaminho com moveTo(), você pode conectar esse ponto a um novo ponto com uma linha reta chamando lineTo(). O código a seguir define um caminho que inclui dois segmentos de linha:

```
c.beginPath() caminhoc.moveTo(100, 100); // Inicie  
um subcaminho em (100,100)c.lineTo(200, 200); //  
Adicione uma linha de (100,100)  
a(200,200)c.lineTo(100, 200); // Adicione uma linha  
de (200,200) a(100,200)
```

Esse código simplesmente define um caminho; não desenha nada na tela. Para desenhar (ou "traçar") os dois segmentos de linha no caminho, chame o método stroke() e para preencher a área definida por esses segmentos de linha, chame fill():

c.fill(); c.stroke();	<i>Preencher uma área triangular Traçar dois lados do triângulo</i>
--------------------------	---

Esse código (junto com algum código adicional para definir larguras de linha e cores de preenchimento) produziu o desenho mostrado na Figura 15-7.



Figura 15-7. Um caminho simples, preenchido e acariciado

Observe que o subcaminho definido na Figura 15-7 é "aberto". Consiste em apenas dois segmentos de linha e o ponto final não está conectado de volta ao ponto inicial. Isso significa que ele não inclui uma região. O método `fill()` preenche subcaminhos abertos agindo como se uma linha reta conectasse o último ponto do subcaminho ao primeiro ponto do subcaminho. É por isso que esse código preenche um triângulo, mas traça apenas dois lados do triângulo.

Se você quisesse traçar todos os três lados do triângulo que acabamos de mostrar, chamaria o método `closePath()` para conectar o ponto final do subcaminho ao ponto inicial. (Você também pode chamar `lineTo(100,100)`, mas acaba com três segmentos de linha que compartilham um início e um ponto final, mas não são realmente fechados. Ao desenhar com linhas largas, os resultados visuais são melhores se você usar `closePath()`.)

Há dois outros pontos importantes a serem observados sobre `stroke()` e `fill()`. Primeiro, ambos os métodos operam em todos os subcaminhos no caminho atual. Suponha que tivéssemos adicionado outro subcaminho no código anterior:

```
c.moveTo(300.100);    Comece um novo subcaminho em (300.100);
c.lineTo(300.200);    Desenhe uma linha vertical até
(300,200);
```

Se chamássemos `stroke()`, desenhariímos duas arestas conectadas de um triângulo e uma linha vertical desconectada.

O segundo ponto a ser observado sobre `stroke()` e `fill()` é que nenhum deles altera o caminho atual: você pode chamar `fill()` e o caminho ainda

Esteja lá quando você chamar stroke(). Quando você terminar um caminho e quiser começar outro, lembre-se de chamar beginPath(). Se você não fizer isso, acabará adicionando novos subcaminhos ao caminho existente e poderá acabar desenhando esses subcaminhos antigos repetidamente.

O exemplo 15-5 define uma função para desenhar polígonos regulares e demonstra o uso de moveTo(), lineTo() e closePath() para definir subcaminhos e de fill() e stroke() para desenhar esses caminhos. Ele produz o desenho mostrado na Figura 15-8.



Figura 15-8. Polígonos regulares

Exemplo 15-5. Polígonos regulares com moveTo(), lineTo() e closePath()

Defina um polígono regular com n lados, centralizado em (x,y) com raio r.// Os vértices são igualmente espaçados ao longo da circunferência de um círculo.// Coloque o primeiro vértice para cima ou no ângulo especificado.// Gire no sentido horário, a menos que o último argumento seja verdadeiro. função polígono(c, n, x, y, r, ângulo=0,

```

sentido anti-horário = falso) {
    c.moveTo(x + r*Math.sin(ângulo), Comece um novo subcaminho em
0 primeiro vértice
        y - r*Math.cos(ângulo)); Use trigonometria para
Posição de computação
    seja delta = 2*Math.PI/n;           Distância angular
entre vértices
Parafor da etapa de desenho é só subtraír o ângulo += sentido anti-
horário?-delta:delta; Ajuste o ângulo

    c.lineTo(x + r*Math.sin(ângulo),           Adicionar linha a
próximo vértice
        y - r*Math.cos(ângulo));
    }c.closePath(                                Conecte o último vértice
);
de volta ao
primeiro}

```

Suponha que haja apenas uma tela e obtenha seu objeto de contexto para desenhar com.let c =
document.querySelector("canvas").getContext("2d");

Inicie um novo caminho e adicione o
polígono subpathsc.beginPath(); polígono
(c, 3, 50, 70, 50); Triângulo
polígono (c, 4, 150, 60, 50, Math.PI / 4); Quadrado
polígono (c, 5, 255, 55, 50); Pentágono
polígono (c, 6, 365, 53, 50, Math.PI / 6); Hexágono
polígono(c, 4, 365, 53, 20, Math.PI/4, verdadeiro);
Quadrado pequeno dentro do hexágono

Defina algumas propriedades que controlam a aparência dos
gráficosc.fillStyle = "#ccc";// Interior cinza
clarosc.strokeStyle = "#008";// contornado com linhas azuis
escurasc.lineWidth = 5;// cinco pixels de largura.

Agora desenhe todos os polígonos (cada um em seu próprio
subcaminho) com these callsc.fill();// Preencha o
shapesc.stroke();// E trace seus contornos

Observe que este exemplo desenha um hexágono com um quadrado dentro dele. O

quadrado e o hexágono são subcaminhos separados, mas se sobrepõem. Quando isso acontece (ou quando um único subcaminho se cruza), a tela precisa ser capaz de determinar quais regiões estão dentro do caminho e quais estão fora. A tela usa um teste conhecido como "regra de enrolamento diferente de zero" para conseguir isso. Neste caso, o interior do quadrado não é preenchido porque o quadrado e o hexágono foram desenhados em direções opostas: os vértices do hexágono foram conectados com segmentos de linha movendo-se no sentido horário ao redor do círculo. Os vértices do quadrado foram conectados no sentido anti-horário. Se o quadrado também tivesse sido desenhado no sentido horário, a chamada para `fill()` também teria preenchido o interior do quadrado.

15.8.2 Dimensões e coordenadas da tela

Os atributos de largura e altura do `<canvas>` elemento e as propriedades de largura e altura correspondentes do objeto `Canvas` especificam as dimensões da tela. O sistema de coordenadas de tela padrão coloca a origem `(0,0)` no canto superior esquerdo da tela. As coordenadas `x` aumentam para a direita e as coordenadas `y` aumentam à medida que você desce na tela. Os pontos na tela podem ser especificados usando valores de ponto flutuante.

As dimensões de uma tela não podem ser alteradas sem redefinir completamente a tela. Definir as propriedades de largura ou altura de uma tela (mesmo definindo-as com seu valor atual) limpa a tela, apaga o caminho atual e redefine todos os atributos gráficos (incluindo a transformação atual e a região de recorte) para seu estado original.

Os atributos de largura e altura de uma tela especificam o valor real

número de pixels que a tela pode desenhar. Quatro bytes de memória são alocados para cada pixel, portanto, se width e height forem definidos como 100, a tela alocará 40.000 bytes para representar 10.000 pixels.

Os atributos width e height também especificam o tamanho padrão (pixels inCSS) no qual a tela será exibida na tela.

If window.devicePixelRatio é 2, então 100×100 pixels CSS são na verdade 40.000 pixels de hardware. Quando o conteúdo da tela é desenhado na tela, os 10.000 pixels na memória precisarão ser ampliados para cobrir 40.000 pixels físicos na tela, e isso significa que seus gráficos não serão tão nítidos quanto poderiam ser.

Para obter a melhor qualidade de imagem, você não deve usar os atributos width e height para definir o tamanho da tela na tela. Em vez disso, defina o tamanho desejado na tela Tamanho do pixel CSS da tela com atributos de estilo CSS width e height. Em seguida, antes de começar a desenharem seu código JavaScript, defina as propriedades de largura e altura do objeto canvas para o número de pixels CSS times window.devicePixelRatio. Continuando com o exemplo anterior, essa técnica resultaria na exibição da tela em 100×100 pixels CSS, mas alocando memória para 200×200 pixels. (Mesmo com essa técnica, o usuário pode ampliar a tela e ver gráficos difusos ou pixelados se o fizerem. Isso contrasta com os SVGgraphics, que permanecem nítidos, independentemente do tamanho da tela ou do nível de zoom.)

15.8.3 Atributos gráficos

O exemplo 15-5 define as propriedades fillStyle, strokeStyle e

`lineWidth` no objeto de contexto da tela. Essas propriedades são atributos gráficos que especificam a cor a ser usada por `fill()` e `stroke()` e a largura das linhas a serem desenhadas por `stroke()`. Observe que esses parâmetros não são passados para os métodos `fill()` e `stroke()`, mas fazem parte do estado gráfico geral da tela. Se você definir um método que desenha uma forma e não definir essas propriedades por conta própria, o chamador do método poderá definir a cor da forma definindo as propriedades `strokeStyle` e `fillStyle` antes de chamar o método. Essa separação do estado gráfico dos comandos de desenho é fundamental para a API do Canvas e é semelhante à separação da apresentação do conteúdo obtida pela aplicação de folhas de estilo CSS a documentos HTML.

Há várias propriedades (e também alguns métodos) no objeto de contexto que afetam o estado gráfico da tela. Eles são detalhados abaixo.

ESTILOS DE LINHAA propriedade `lineWidth` especifica a largura (em pixels CSS) das linhas desenhadas por `stroke()`. O valor padrão é 1. É importante entender que a largura da linha é determinada pela propriedade `lineWidth` no momento em que `stroke()` é chamado, não no momento em que `lineTo()` e outros métodos de construção de caminho são chamados. Para entender completamente a propriedade `lineWidth`, é importante visualizar caminhos como linhas unidimensionais infinitamente finas. As linhas e curvas desenhadas pelo método `stroke()` são centralizadas sobre o caminho, com metade da larguraLinha em cada lado. Se você está acariciando um caminho fechado e apenas

deseja que a linha apareça fora do caminho, trace o caminho primeiro e, em seguida, preencha com uma cor opaca para ocultar a parte do traçado que aparece dentro do caminho. Ou se você quiser que a linha apareça apenas dentro de um closedpath, chame os métodos save() e clip() primeiro, depois chame () e restore(). (Os métodos save(), restore() e clip() são descritos posteriormente.)

Ao desenhar linhas com mais de dois pixels de largura, as propriedades lineCap e lineJoin podem ter um impacto significativo na aparência visual das extremidades de um caminho e nos vértices nos quais dois segmentos de caminho se encontram. A Figura 15-9 ilustra os valores e a aparência gráfica resultante de lineCap e lineJoin.

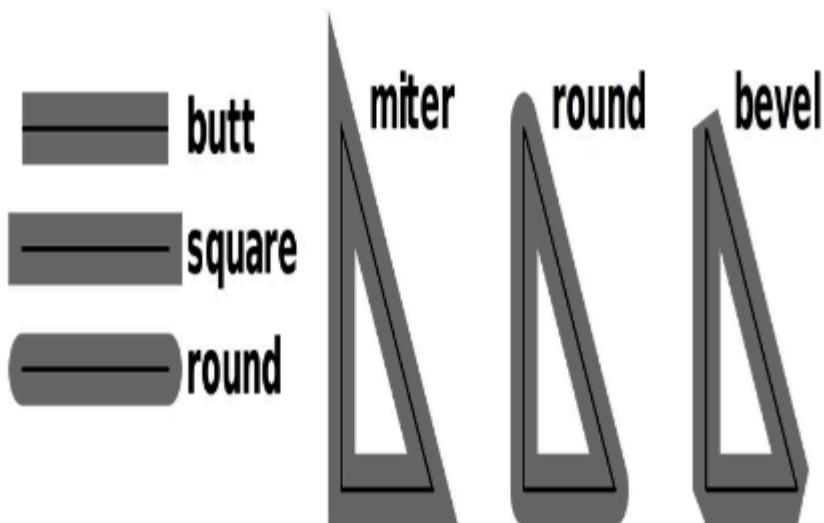


Figura 15-9. Os atributos lineCap e lineJoin

O valor padrão para lineCap é "butt". O valor padrão para

`lineJoin` é "mitra". Observe, no entanto, que se duas linhas se encontrarem em um ângulo muito estreito, a esquadria resultante pode se tornar bastante longa e visualmente perturbadora. Se a mitra em um determinado vértice for maior que a metade da largura da linha vezes a propriedade `mitreLimit`, esse vértice será desenhado com uma junção chanfrada em vez de uma junção mitrada. O valor padrão para `miterLimit` é 10.

O método `stroke()` pode desenhar linhas tracejadas e pontilhadas, bem como linhas sólidas, e o estado gráfico de uma tela inclui uma matriz de números que serve como um "padrão de traço", especificando quantos pixels desenhar e, em seguida, quantos omitir. Ao contrário de outras propriedades de desenho de linha, o padrão de traços é definido e consultado com os métodos `setLineDash()` e `getLineDash()` em vez de com uma propriedade. Para especificar um padrão de traço pontilhado, você pode usar `setLineDash()` assim:

```
C.setLineDash ([18, 3, 3, 3]); Painel de 18PX, espaço de  
3PX, em 3PXD, espaço de 3PX
```

Por fim, a propriedade `lineDashOffset` especifica até que ponto o desenho do padrão de traço deve começar. O padrão é 0. Os caminhos traçados com o padrão de traço mostrado aqui começam com um traço de 18 pixels, mas `lineDashOffset` é definido como 21, então esse mesmo caminho começaria com um ponto seguido por um espaço e um traço.

CORES, PADRÕES E GRADIENTES As propriedades `fillStyle` e `strokeStyle` especificam como os caminhos são preenchidos e traçados. A palavra "estilo" geralmente significa cor, mas essas propriedades também podem ser usadas para especificar um gradiente de cor ou uma imagem a ser usada para preencher e acariciar. (Observe que desenhar uma linha é basicamente o

o mesmo que preencher uma região estreita em ambos os lados da linha, e encher e acariciar são fundamentalmente a mesma operação.)

Se você deseja preencher ou traçar com uma cor sólida (ou uma cor translúcida), basta definir essas propriedades como uma string de cor CSS válida. Nada mais é necessário.

Para preencher (ou traçar) com um gradiente de cor, defina `fillStyle` (`strokeStyle`) como um objeto `CanvasGradient` retornado pelos métodos `createLinearGradient()` ou `createRadialGradient()` do contexto. Os argumentos para `criarGradiente Linear()` são as coordenadas de dois pontos que definem uma linha (não precisa ser horizontal ou vertical) ao longo da qual as cores irão variar. Os argumentos para `createRadialGradient()` especificam os centros e raios de dois círculos. (Eles não precisam ser concêntricos, mas o primeiro círculo normalmente fica inteiramente dentro do segundo.) As áreas dentro do círculo menor ou fora do maior serão preenchidas com cores sólidas; As áreas entre os dois serão preenchidas com um gradiente de cores.

Depois de criar o objeto `CanvasGradient` que define as regiões da tela que serão preenchidas, você deve definir as cores do gradiente chamando o método `addColorStop()` do `CanvasGradient`. O primeiro argumento para este método é um número entre 0,0 e 1,0. O segundo argumento é uma especificação de cores CSS. Você deve chamar esse método pelo menos duas vezes para definir um gradiente de cor simples, mas você pode chamá-lo de mais do que isso. A cor em 0,0 aparecerá no início do gradiente e a cor em 1,0 aparecerá no final. Se você especificar cores adicionais, elas aparecerão na posição fracionária especificada dentro do gradiente.

Entre os pontos que você especificar, as cores serão interpoladas suavemente. Aqui estão alguns exemplos:

```
Um gradiente linear, diagonalmente ao longo da tela  
(supondo que não haja transformações)let bgfade  
=c.createLinearGradient(0,0,canvas.width,canvas.height);  
bgfade.addColorStop(0.0, "#88f");// Comece com azul claro no  
canto superior esquerdo bgfade.addColorStop(1.0, "#fff");//  
Desvanecer-se para branco no canto inferior direito
```

```
Um gradiente entre dois círculos concêntricos. Transparente  
no meio// desbotando para cinza translúcido e depois de  
volta para transparente.let donut =  
c.createRadialGradient(300,300,100, 300,300,300);  
donut.addColorStop(0.0,  
"transparent");//Transparente donut.addColorStop(0.7,  
"rgba(100,100,100,.9)");//Cinza  
translúcido donut.addColorStop(1.0,  
"rgba(0,0,0,0)");//Transparente novamente
```

Um ponto importante a entender sobre gradientes é que eles não são independentes de posição. Ao criar um gradiente, você especifica bounds para o gradiente. Se você tentar preencher uma área fora desses limites, obterá a cor sólida definida em uma extremidade ou na outra do gradiente.

Além de cores e gradientes de cores, você também pode preencher e traçar usando imagens. Para fazer isso, defina fillStyle ou strokeStyle como umCanvasPattern retornado pelo método createPattern() do objeto context. O primeiro argumento para esse método deve ser um <canvas> elemento ou que contém a imagem com a qual você deseja preencher orstroke. (Observe que a imagem ou tela de origem não precisa ser

inserido no documento para ser usado dessa maneira.) O segundo argumento para createPattern() é a string "repeat", "repeat-x", "repeat-y" ou "no-repeat", que especifica se (e em quais dimensões) as imagens de fundo se repetem.

ESTILOS DE TEXTO A propriedade font especifica a fonte a ser usada pelos métodos text-drawing: fillText() e strokeText() (consulte "Texto"). O valor da propriedade font deve ser uma string na mesma sintaxe que o atributo CSSfont.

A propriedade textAlign especifica como o texto deve ser alinhado horizontalmente em relação à coordenada X passada tofillText() ou strokeText(). Os valores legais são "início", "esquerda", "centro", "direita" e "fim". O padrão é "start", que, para texto da esquerda para a direita, tem o mesmo significado que "left".

A propriedade textBaseline especifica como o texto deve ser alinhado verticalmente em relação à coordenada y. O valor padrão é "alfabético" e é apropriado para scripts latinos e semelhantes. O valor "ideográfico" destina-se ao uso com scripts como chinês e japonês. O valor "pendurado" destina-se ao uso com Devanagari e scripts semelhantes (que são usados para muitos dos idiomas da Índia). As linhas de base "superior", "intermediária" e "inferior" são linhas de base puramente geométricas, com base no "em" quadrado da fonte.

SHADOWS Para propriedades do objeto de contexto controlam o desenho de soltar

Sombras. Se você definir essas propriedades adequadamente, qualquer linha, área, texto ou imagem que você desenhar receberá uma sombra, o que fará com que pareça estar flutuando acima da superfície da tela.

A propriedade shadowColor especifica a cor da sombra. O padrão é preto totalmente transparente e as sombras nunca aparecerão, a menos que você defina essa propriedade como uma cor translúcida ou opaca. Esta propriedade só pode ser definida como uma string de cores: padrões e gradientes não são permitidos para sombras. O uso de uma cor de sombra translúcida produz os efeitos de sombra mais realistas, pois permite que o plano de fundo apareça.

As propriedades shadowOffsetX e shadowOffsetY especificam os deslocamentos X e Y da sombra. O padrão para ambas as propriedades é 0, que coloca a sombra diretamente abaixo do desenho, onde ela não é visível. Se você definir ambas as propriedades com um valor positivo, as sombras aparecerão abaixo e à direita do que você desenha, como se houvesse uma fonte de luz acima e à esquerda, brilhando na tela de fora da tela do computador. Deslocamentos maiores produzem sombras maiores e objetos desenhados aparecem como se estivessem flutuando "mais alto" acima da tela. Esses valores não são afetados por transformações de coordenadas (§15.8.5): a direção da sombra e a "altura" permanecem consistentes mesmo quando as formas são giradas e dimensionadas.

A propriedade shadowBlur especifica o quanto borradassão as bordas da sombra. O valor padrão é 0, o que produz sombras nítidas e não desfocadas. Valores maiores produzem mais desfoque, até um limite superior definido pela implementação.

TRANSLUCIDEZ E COMPOSIÇÃO

Se você deseja traçar ou preencher um caminho usando uma cor translúcida, você pode setstrokeStyle ou fillStyle usando uma sintaxe de cor CSS como "rgba(...)" que suporta transparência alfa. O "a" em "RGBA" significa "alfa" e é um valor entre 0 (totalmente transparente) e 1 (totalmente opaco). Mas a API do Canvas fornece outra maneira de trabalhar com cores translúcidas.

Se você não quiser especificar explicitamente um canal alfa para cada cor ou se quiser adicionar translucidez a imagens opacas ou padrões, poderá definir a propriedade globalAlpha. Cada pixel que você desenhar terá seu valor alfa multiplicado por globalAlpha. O padrão é 1, o que não adiciona transparência. Se você definir globalAlpha como 0, tudo o que você desenhar será totalmente transparente e nada aparecerá na tela. Mas se você definir essa propriedade como

0,5, então os pixels que seriam opacos serão 50% opacos e os pixels que seriam 50% opacos serão 25% opacos.

Ao traçar linhas, preencher regiões, desenhar texto ou copiar imagens, você geralmente espera que os novos pixels sejam desenhados sobre os pixels que já estão na tela. Se você estiver desenhando pixels opacos, eles simplesmente substituem os pixels que já estão lá. Se você estiver desenhando compixels translúcidos, o novo pixel ("origem") é combinado com o pixel antigo ("destino") para que o pixel antigo apareça através do novo pixel com base em quanto transparente esse pixel é.

Esse processo de combinação de novos pixels de origem (possivelmente translúcidos) com pixels de destino existentes (possivelmente translúcidos) é chamado de composição, e o processo de composição descrito anteriormente é a maneira padrão pela qual a API do Canvas combina pixels. Mas você pode definir o

`globalCompositeOperation` para especificar outras maneiras de combinar pixels. O valor padrão é "source-over", o que significa que os pixels de origem são desenhados "sobre" os pixels de destino e são combinados com eles se a origem for translúcida. Mas se você definir `globalCompositeOperation` como "destination-over", a tela combinará pixels como se os novos pixels de origem fossem desenhados abaixo dos pixels de destino existentes. Se o destino for translúcido ou transparente, parte ou toda a cor do pixel de origem ficará visível na cor resultante. Como outro exemplo, o modo de composição "source-atop" combina os pixels de origem com a transparência dos pixels de destino para que nada seja desenhado em partes da tela que já são totalmente transparentes. Existem vários valores legais para `globalCompositeOperation`, mas a maioria tem apenas usos especializados e não são abordados aqui.

SALVANDO E RESTAURANDO ESTADOS GRÁFICOS Como a API do Canvas define atributos gráficos no objeto de contexto, você pode ficar tentado a chamar `getContext()` várias vezes para obter vários objetos de contexto. Se você pudesse fazer isso, poderia definir diferentes atributos em cada contexto: cada contexto seria como um pincel diferente e pintaria com uma cor diferente ou desenharia linhas de larguras diferentes. Infelizmente, você não pode usar a tela dessa maneira. Cada `<canvas>` elemento tem apenas um único objeto de contexto, e `everycall` para `getContext()` retorna o mesmo `CanvasRenderingContext2D` object.

Embora a API do Canvas permita que você defina apenas um único conjunto de atributos gráficos por vez, ela permite que você salve o

graphics para que você possa alterá-lo e restaurá-lo facilmente mais tarde. O método Thesave() envia o estado gráfico atual para uma pilha de estados salvos. O método restore() abre a pilha e restaura o estado salvo mais recentemente. Todas as propriedades descritas nesta seção fazem parte do estado salvo, assim como a transformação atual e a região de recorte (ambas explicadas posteriormente). É importante ressaltar que o caminho definido atualmente e o ponto atual não fazem parte do estado gráfico e não podem ser salvos e restaurados.

15.8.4 Operações de desenho de tela

Já vimos alguns métodos básicos de tela — beginPath(), moveTo(), lineTo(), closePath(), fill() e stroke() — para definir, preencher e desenhar linhas e polígonos. Mas o CanvasAPI também inclui outros métodos de desenho.

RECTANGLES CanvasRenderingContext2D define quatro métodos para desenhar retângulos. Todos esses quatro métodos de retângulo esperam dois argumentos que especificam um canto do retângulo seguido pela largura e altura do retângulo. Normalmente, você especifica o canto superior esquerdo e, em seguida, passa uma largura positiva e uma altura positiva, mas também pode especificar outros cantos e passar dimensões negativas.

fillRect() preenche o retângulo especificado com o currentfillStyle. strokeRect() traça o contorno do retângulo especificado usando o strokeStyle atual e outros atributos de linha. clearRect() é como fillRect(), mas ignora o preenchimento atual

e preenche o retângulo com pixels pretos transparentes (a cor padrão de todas as telas em branco). O importante sobre esses três métodos é que eles não afetam o caminho atual ou o ponto atual dentro desse caminho.

O método do retângulo final é chamado `rect()` e afeta o caminho atual: ele adiciona o retângulo especificado, em um subcaminho próprio, ao caminho. Como outros métodos de definição de caminho, ele não preenche ou acaricia nada em si.

CURVAS Um caminho é uma sequência de subcaminhos, e um subcaminho é uma sequência de pontos conectados. Nos caminhos que definimos no §15.8.1, esses pontos estavam conectados com segmentos de linha reta, mas nem sempre é esse o caso. O objeto `CanvasRenderingContext2D` define vários métodos que adicionam um novo ponto ao subcaminho e conectam o ponto atual a esse novo ponto com uma curva:

arco()

Esse método adiciona um círculo, ou uma parte de um círculo (um arco), ao caminho. O arco a ser desenhado é especificado com seis parâmetros: as coordenadas x e y do centro de um círculo, o raio do círculo, os ângulos inicial e final do arco e a direção (no sentido horário ou anti-horário) do arco entre esses dois ângulos. Se houver um ponto atual no caminho, então este método conecta o ponto atual ao início do arco com uma linha reta (o que é útil ao desenhar fatias ou fatias de pizza) e, em seguida, conecta o início do arco ao final do arco com uma parte de um círculo, deixando o final do arco como o novo ponto atual. Se não houver nenhum ponto atual quando esse método for chamado, ele apenas adicionará o arco circular ao

caminho.

ellipse()

Este método é muito parecido com arc(), exceto que adiciona uma elipse ou uma parte de uma elipse ao caminho. Em vez de um raio, ele tem dois: um raio do eixo x e um raio do eixo y. Além disso, como as elipses não são radialmente simétricas, esse método usa outro argumento que especifica o número de radianos pelos quais a elipse é girada no sentido horário em torno de seu centro.

arcTo()

Este método desenha uma linha reta e um arco circular assim como o método arc() faz, mas especifica o arco a ser desenhado usando parâmetros diferentes. Os argumentos para arcTo() especificam os pontos P1 e P2 e um raio. O arco adicionado ao caminho tem o raio especificado. Começa no ponto tangente com a reta (imaginária) do ponto atual até P1 e termina no ponto tangente com a reta (imaginária) entre P1 e P2. Este método aparentemente incomum de especificar arcos é realmente bastante útil para desenhar formas com cantos arredondados. Se você especificar um raio de 0, thismethod apenas desenha uma linha reta do ponto atual para P1. Com um raio diferente de zero, no entanto, ele desenha uma linha reta a partir do ponto atual na direção de P1, então curva essa linha em um círculo até que esteja indo na direção de P2.

bezierCurveTo()

Este método adiciona um novo ponto P ao subcaminho e o conecta ao ponto atual com uma curva de Bézier cúbica. A forma da curva é especificada por dois "pontos de controle", C1 e C2. No início da curva (no ponto atual), a curva segue na direção de C1. At final da curva (no ponto P), a curva chega da direção de C2. Entre esses pontos, a direção da curva varia suavemente. O ponto P torna-se o novo ponto atual para o

subcaminho.

quadraticCurveTo()

Esse método é como `bezierCurveTo()`, mas usa uma curva de Bézier quadrática em vez de uma curva de Bézier cúbica e tem apenas um único ponto de controle.

Você pode usar esses métodos para desenhar caminhos como os da Figura 15-10.



Figura 15-10. Caminhos curvos em uma tela

O Exemplo 15-6 mostra o código usado para criar a Figura 15-10. Os métodos demonstrados neste código são alguns dos mais complicados na API do Canvas; Consulte uma referência online para obter detalhes completos sobre os métodos e seus argumentos.

Exemplo 15-6. Adicionando curvas a um caminho

Uma função utilitária para converter ângulos de graus para radians

```
function rads(x) { return Math.PI*x/180; }
```

Obtém o objeto de contexto do elemento let canvas do documento

```
c = document.querySelector("canvas").getContext("2d");
```

Defina alguns atributos gráficos e desenhe o

```
curvesc.fillStyle = "#aaa";// Cinza fillsc.lineWidth = 2;//  
Linhas pretas de 2 pixels (por padrão)
```

Desenhe um círculo.// Não há ponto atual, então desenhe apenas o círculo com nostraight// linha do ponto atual até o início do circle.c.beginPath(); c.arc(75,100,50,// Centro em (75,100), raio 50

0, rads(360), falso); Vá no sentido horário de 0 a 360 graus.c.fill();// Preencha o círculo.c.stroke();// Trace seu contorno.

Agora desenhe uma elipse da mesma maneira.c.beginPath();// Inicie um novo caminho não conectado ao círculo.c.ellipse(200, 100, 50, 35, rads(15),// Centro, raios e rotação

0, rads(360), falso); Ângulo inicial, final de ângulo, de direção

Desenhe uma cunha. Os ângulos são medidos no sentido horário a partir do eixo x positivo.// Observe que arc() adiciona uma linha do ponto atual ao arco start.c.moveTo(325, 100);// Comece no centro do círculo.c.arc(325, 100, 50,// Centro e raio do círculo

*rads(-60), rads(0), // Comece no ângulo -60 e vá para o ângulo 0
verdadeiro); No sentido anti-horário
c.closePath(); Adicione o raio de volta ao centro de 0 círculo*

Cunha semelhante, deslocada um pouco e na direção oposta.c.moveTo(340, 92); c.arc(340, 92, 42, rads(-60), rads(0), falso); c.closePath();

Use arcTo() para cantos arredondados. Aqui desenhamos um quadrado com// canto superior esquerdo em (400,50) e cantos de raios variados.c.moveTo(450, 50);// Comece no meio da borda superior.c.arcTo(500,50,500,150,30);// Adicione parte da borda superior e canto superior direito.c.arcTo(500,150,400,150,20); Adicione a borda direita e o canto inferior direito.

```
c.arcTo(400,150,400,50,10); Adicione a borda inferior e o canto inferior  
canto.c.arcTo(400,50,500  
,50,0); Adicione a borda esquerda e o canto superior  
corner.c.clos  
ePath(); Caminho fechado para adicionar o restante de  
a borda superior.
```

```
Curva de Bézier quadrática: um ponto de  
controlec.moveTo(525, 125);// Comece  
aqui.c.quadraticCurveTo(550, 75, 625, 125);// Desenhe uma  
curva para(625, 125)c.fillRect(550-3, 75-3, 6, 6);// Marque  
o ponto de controle (550,75)
```

```
Curva de Bézier  
cúbicac.moveTo(625, 100); Comece em (625, 100)  
c.bezierCurveTo(645,70,705,130,725,100); Curva para (725,  
100)c.fillRect(645-3, 70-3, 6, 6);// Marcar pontos de  
controlec.fillRect(705-3, 130-3, 6, 6);
```

```
Por fim, preencha as curvas e trace seus  
outlines.c.fill(); c.stroke();
```

TEXTOPara desenhar texto em uma tela, você normalmente usa o método `fillText()`, que desenha o texto usando a cor (ou gradiente ou padrão) especificada pela propriedade `fillStyle`. Para efeitos especiais em tamanhos de texto grandes, você pode usar `strokeText()` para desenhar o contorno dos fontglyphs individuais. Ambos os métodos usam o texto a ser desenhado como primeiro argumento e usam as coordenadas x e y do texto como segundo e terceiro argumentos. Nenhum dos métodos afeta o caminho atual ou o ponto atual.

`fillText()` e `strokeText()` recebem um quarto argumento opcional. Se fornecido, esse argumento especifica a largura máxima do texto a ser exibido. Se o texto for mais largo do que o valor especificado

Quando desenhado usando a propriedade font, a tela fará com que ela se ajuste dimensionando-a ou usando uma fonte mais estreita ou menor.

Se você precisar medir o texto antes de desenhá-lo, passe-o para o método measureText(). Este método retorna um objeto TextMetrics que especifica as medidas do texto quando desenhado com a fonte atual. No momento da redação deste artigo, a única "métrica" contida no objeto TextMetrics é a largura. Consulte a largura na tela de uma cadeia de caracteres como esta:

```
let largura = c.measureText(texto).largura;
```

Isso é útil se você quiser centralizar uma sequência de texto em uma tela, por exemplo.

IMAGENS Em adição a gráficos vetoriais (caminhos, linhas, etc.), a API do Canvas também oferece suporte a imagens bitmap. O método drawImage() copia os pixels de uma imagem de origem (ou de um retângulo dentro da imagem de origem) para a tela, dimensionando e girando os pixels da imagem conforme necessário.

drawImage() pode ser invocado com três, cinco ou nove argumentos. Em todos os casos, o primeiro argumento é a imagem de origem da qual os pixels devem ser copiados. Esse argumento de imagem geralmente é um elemento, mas também pode ser outro <canvas> elemento ou até mesmo um <video> elemento (do qual um único quadro será copiado). Se você especificar um <video> elemento or que ainda está carregando seus dados, a chamada drawImage() não fará nada.

Na versão de três argumentos de `drawImage()`, o segundo e o terceiro argumentos especificam as coordenadas x e y nas quais o canto superior esquerdo da imagem deve ser desenhado. Nesta versão do método, toda a imagem de origem é copiada para a tela. As coordenadas x e y são interpretadas no sistema de coordenadas atual, e a imagem é dimensionada e girada, se necessário, dependendo da transformação da tela atualmente no efeito.

A versão de cinco argumentos de `drawImage()` adiciona argumentos `width` e `height` aos argumentos x e y descritos anteriormente.

Esse quatro argumentos definem um retângulo de destino dentro da tela. O canto superior esquerdo da imagem de origem fica em (x,y) e o canto inferior direito fica em (x+largura, y+altura). Novamente, a imagem entire resource é copiada. Com esta versão do método, a imagem de origem será dimensionada para caber no retângulo de destino.

A versão de nove argumentos de `drawImage()` especifica um retângulo de origem e um retângulo de destino e copia apenas os pixels dentro do retângulo de origem. Os argumentos de dois a cinco especificam o `sourceRectangle`. Eles são medidos em pixels CSS. Se a imagem de origem for outra tela, o retângulo de origem usará o sistema de coordenadas padrão para essa tela e ignorará todas as transformações que foram especificadas. Os argumentos de seis a nove especificam o retângulo de destino no qual a imagem é desenhada e estão no sistema de coordenadas atual da tela, não no sistema de coordenadas padrão.

Além de desenhar imagens em uma tela, também podemos extrair o conteúdo de uma tela como uma imagem usando o método `toDataURL()`.

Ao contrário de todos os outros métodos descritos aqui, `toDataURL()` é um método do próprio elemento Canvas, não do objeto context.

Normalmente, você invoca `toDataURL()` sem argumentos e retorna o conteúdo da tela como uma imagem PNG, codificada como uma string usando adata: URL. A URL retornada é adequada para uso com um ``elemento, e você pode fazer um instantâneo estático de uma tela com um código como este:

```
let img = document.createElement("img"); Crie um <img>
elemento
img.src = canvas.toDataURL(); Defina seu src
attributedocument.body.appendChild(img); Acrescente-o ao
documento
```

15.8.5 Transformações do sistema de coordenadas

Como observamos, o sistema de coordenadas padrão de uma tela coloca a origem no canto superior esquerdo, tem coordenadas x aumentando para a direita e tem coordenadas y aumentando para baixo. Neste sistema padrão, as coordenadas de um ponto são mapeadas diretamente para um pixel CSS (que então mapeia diretamente para um ou mais pixels do dispositivo).

Determinadas operações e atributos da tela (como extrair valores brutos de pixel e definir sombreamentos) sempre usam esse sistema de coordenadas padrão. Além do sistema de coordenadas padrão, no entanto, cada tela tem uma "matriz de transformação atual" como parte de seu estado gráfico. Essa matriz define o sistema de coordenadas atual da tela. Na maioria das operações de tela, quando você especifica as coordenadas de um ponto, ele é considerado um ponto no sistema de coordenadas atual, não no sistema de coordenadas padrão. A matriz de transformação atual é usada para converter as coordenadas especificadas nas coordenadas equivalentes no sistema de coordenadas padrão.

O método `setTransform()` permite que você defina a matriz de transformação de uma tela diretamente, mas as transformações do sistema de coordenadas geralmente são mais fáceis de especificar como uma sequência de translações, rotações e operações de dimensionamento. A Figura 15-11 ilustra essas operações e seus efeitos no sistema de coordenadas da tela. O programa que produziu a figura desenhou o mesmo conjunto de machados sete vezes seguidas. A única coisa que mudou a cada vez foi a transformação atual. Observe que as transformações afetam o texto, bem como as linhas desenhadas.

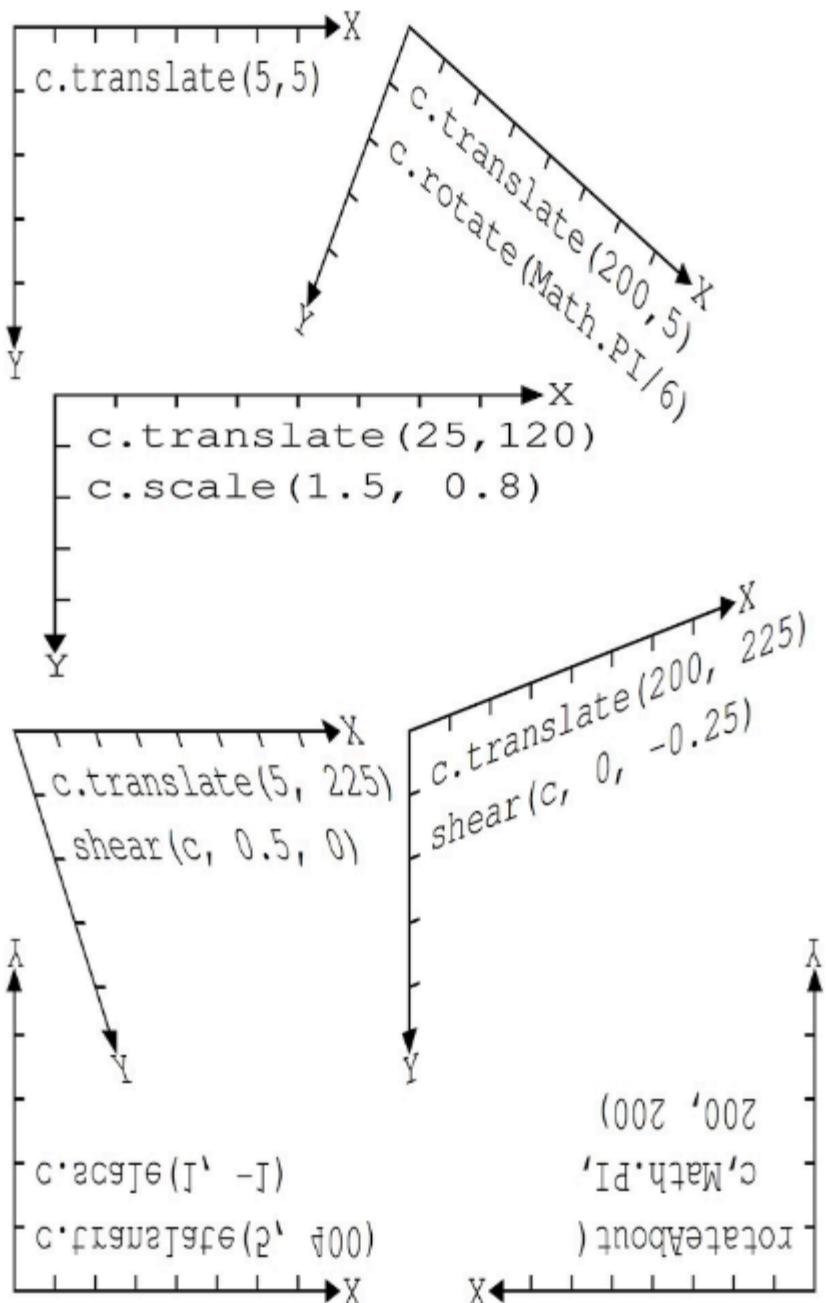


Figura 15-11. Transformações do sistema de coordenadas

O método `translate()` simplesmente move a origem do sistema de coordenadas para a esquerda, direita, para cima ou para baixo. O método `rotate()` gira os eixos no sentido horário pelo ângulo especificado. (A API do Canvas sempre especifica ângulos em radianos. Para converter graus em radianos, divida por 180 e multiplique por `Math.PI`.) O método `scale()` estica ou contrai distâncias ao longo dos eixos x ou y.

Passar um fator de escala negativo para o método `scale()` inverte esse eixo na origem, como se estivesse refletido em um espelho. Isto é o que foi feito no canto inferior esquerdo da Figura 15-11: `translate()` foi usado para mover a origem para o canto inferior esquerdo da tela, então `scale()` foi usado para inverter o eixo y para que as coordenadas y aumentem à medida que subimos na página. Um sistema de coordenadas invertidas como este é familiar da classe de álgebra e pode ser útil para plotar pontos de dados em gráficos. Observe, no entanto, que isso dificulta a leitura do texto!

ENTENDENDO AS TRANSFORMAÇÕES MATEMATICAMENTE Acho mais fácil entender as transformações geometricamente, pensando em `translate()`, `rotate()` e `scale()` como transformando os eixos do sistema de coordenadas, conforme ilustrado na Figura 15-11. Também é possível entender as transformadas algebricamente como equações que mapeiam as coordenadas de um ponto (x,y) no sistema de coordenadas transformado de volta às coordenadas (x',y') do mesmo ponto no sistema de coordenadas anterior.

A chamada de método `c.translate(dx,dy)` pode ser descrita com

Essas equações:

```
x' = x + dx; Uma coordenada X de 0 no novo sistema é dx  
no antigo'  
= y + dy;
```

As operações de dimensionamento têm equações igualmente simples. Um `callc.scale(sx,sy)` pode ser descrito assim:

```
x' = sx * x;y'  
= sy * y;
```

As rotações são mais complicadas. A chamada `c.rotate(a)` é descrita por estas equações trigonométricas:

```
x' = x * cos(a) - y * sin(a);  
y' = y * cos(a) + x * sin(a);
```

Observe que a ordem das transformações é importante. Suponha que começemos com o sistema de coordenadas padrão de uma tela, depois o traduzamos e, em seguida, o dimensionamos. Para mapear o ponto (x,y) no sistema de coordenadas atual de volta ao ponto (x'',y'') no sistema de coordenadas padrão, devemos primeiro aplicar as equações de escala para mapear o ponto para um ponto intermediário (x',y') no sistema de coordenadas traduzido, mas não dimensionado, e, em seguida, usar as equações de translação para mapear desse ponto intermediário para (x'',y'') . O resultado é este:

```
x'' = sx*x + dx;y''  
= sy*y + dy;
```

Se, por outro lado, tivéssemos chamado `scale()` antes de chamar `translate()`, as equações resultantes seriam diferentes:

```
x'' = sx*(x + dx);  
y'' = sy*(y + dy);
```

A principal coisa a lembrar ao pensar algebricamente sobre sequências de transformações é que você deve trabalhar de trás para frente da última transformação (mais recente) para a primeira. Ao pensar geometricamente sobre eixos transformados, no entanto, você avança da primeira para a última transformação.

As transformações suportadas pela tela são conhecidas como `affinetransforms`. As transformações afins podem modificar as distâncias entre os pontos e os ângulos entre as linhas, mas as linhas paralelas sempre permanecem paralelas após uma transformação afim — não é possível, por exemplo, especificar uma distorção de lente olho de peixe com uma transformação afim. Uma transformação afim arbitrária pode ser descrita pelos seis parâmetros `a`, `b`, `c`, `d`, `e` e `f` nestas equações:

```
x' = ax + cy +  
ey' = bx + dy + f
```

Você pode aplicar uma transformação arbitrária ao sistema de coordenadas atual passando esses seis parâmetros para o método `transform()`. A Figura 15-11 ilustra dois tipos de transformações — cisalhamentos e rotações em torno de um ponto especificado — que você pode implementar com o método `transform()` da seguinte forma:

```
Cisalhamento transform://x' = x + kx*y; //y' = ky*x + y;  
função cisalhamento(c, kx, ky) { c.transform(1, ky, kx, 1,  
0, 0);}
```

Gire os radianos no sentido anti-horário em torno do ponto

```
(x,y)// Isso também pode ser feito com uma função de
sequência translate, rotate,translate rotateAbout(c,
theta, x, y) {

let ct = Math.cos(theta); seja st = Math.sin();
c.transform(ct, -st, st, ct, -x*ct-y*st+x, x*st-y*ct+y);}
```

O método setTransform() usa os mesmos argumentos que transform(), mas em vez de transformar o sistema de coordenadas atual, ele ignora o sistema atual, transforma o sistema de coordenadas padrão e torna o resultado o novo sistema de coordenadas atual. setTransform() é útil para redefinir temporariamente a tela para seu sistema de coordenadas padrão:

```
c.save();                                Salvar coordenada atual
systemc.setTransform(1,0,0,
1,0,0);                                Reverter para o padrão
sistema de coordenadas// Execute operações usando o
pixel CSS padrão coordinates.c.restore();// Restaure o
sistema de coordenadas salvo
```

EXEMPLO DE TRANSFORMAÇÃO Example 15-7 demonstra o poder das transformações do sistema de coordenadas usando os métodos translate(), rotate() e scale() recursivamente para desenhar um fractal de floco de neve de Koch. A saída deste exemplo aparece na Figura 15-12, que mostra os flocos de neve de Koch com 0, 1, 2, 3 e 4 níveis de recursão.

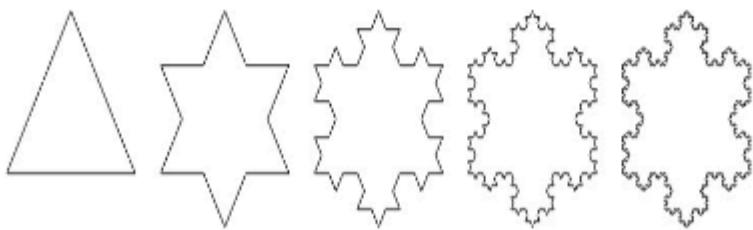


Figura 15-12. Flocos de neve Koch

O código que produz essas figuras é elegante, mas seu uso de transformações recursivas do sistema de coordenadas o torna um pouco difícil de entender. Mesmo que você não siga todas as nuances, observe que o código inclui apenas uma única invocação do método `lineTo()`. Cada segmento de linha na Figura 15-12 é desenhado assim:

```
c.lineTo(len, 0);
```

O valor da variável `len` não muda durante a execução do programa, portanto, a posição, a orientação e o comprimento de cada um dos segmentos de linha são determinados por translações, rotações e operações de escala.

Exemplo 15-7. Um flocos de neve Koch com transformações

seja deg = Math.PI/180; Para converter graus em radianos

Desenhe um fractal de flocos de neve de Koch de nível n no contexto da telac,// com o canto inferior esquerdo em (x,y) e comprimento lateral len.function snowflake(c, n, x, y, len) {

```
c.save();           Salve a transformação atual
c.translate(x,y); Traduzir origem para ponto de partida
c.moveTo(0,0);    Inicie um novo subcaminho no novo
origem
perna(n);         Desenhe a primeira perna do flocos de neve
c.rotate(-120*deg); Agora gire 120 graus
```

No sentido anti-horário

```
perna(n);           Empate a segunda mão
c.rotate(-120*deg); Gire againleg(n);// Desenhe o
legc.closePath();// Feche o subpathc.restore();// E
restaure a transformação original
```

*Desenhe uma única perna de um floco de neve Koch de nível n.//
Esta função deixa o ponto atual no final da perna que possui*

*desenhado e translade o sistema de coordenadas para que
o ponto atual seja (0,0).*

*Isso significa que você pode facilmente chamar rotate()
depois de desenhar aleg.*

```
function perna(n) {
    c.save();           Salve o atual
transformaçāoif (n
== 0) {           Caso não recursivo:
    c.lineTo(len, 0); // Basta desenhar uma horizontal
linha
}
—
else {           Caso recursivo: desenhe 4 sub-
pernas   \c.escala(1/3,1
como:     /3);
esta perna
    perna (n-1);      Recursa para o primeiro sub-
perna
    c.girar (60 * graus); Gire 60 graus no sentido horário
    perna (n-1);      Segunda subetapa
    c.rotate(-120*deg); Gire 120 graus para trás (n-
1); // Terceira sub-legc.rotate(60*deg); // Gire de
    volta para o nosso original
rubrica
    perna (n-1);      Sub-perna final
    }c.restore(
    );
    c.translate(len, 0); Restaurar a transformação
perna (0,0)        Mas traduzir para fazer o fim de
}}                
```

```
let c = document.querySelector("canvas").getContext("2d");
floco de neve(c, 0, 25, 125, 125); // Um floco de neve de nível
0 é um triângulo! floco de neve(c, 1, 175, 125, 125); Um floco de
neve de nível 1 é um floco de neve estrelado de 6 lados (c, 2,
325, 125, 125); etc. floco de neve (c, 3, 475, 125, 125);
floco de neve (c, 4, 625, 125, 125); Um floco de neve de nível
4 parece um floco de neve! c.stroke(); // Acaricie este caminho
muito complicado
```

15.8.6 Recorte

Depois de definir um caminho, você geralmente chama `stroke()` ou `fill()` (ou ambos). Você também pode chamar o método `clip()` para definir uma região de recorte. Depois que uma região de recorte for definida, nada será desenhado fora dela. A Figura 15-13 mostra um desenho complexo produzido usando regiões de recorte. A faixa vertical que desce pelo meio e o texto ao longo da parte inferior da figura foram traçados sem nenhuma região de recorte e, em seguida, preenchidos após a definição da região de recorte triangular.

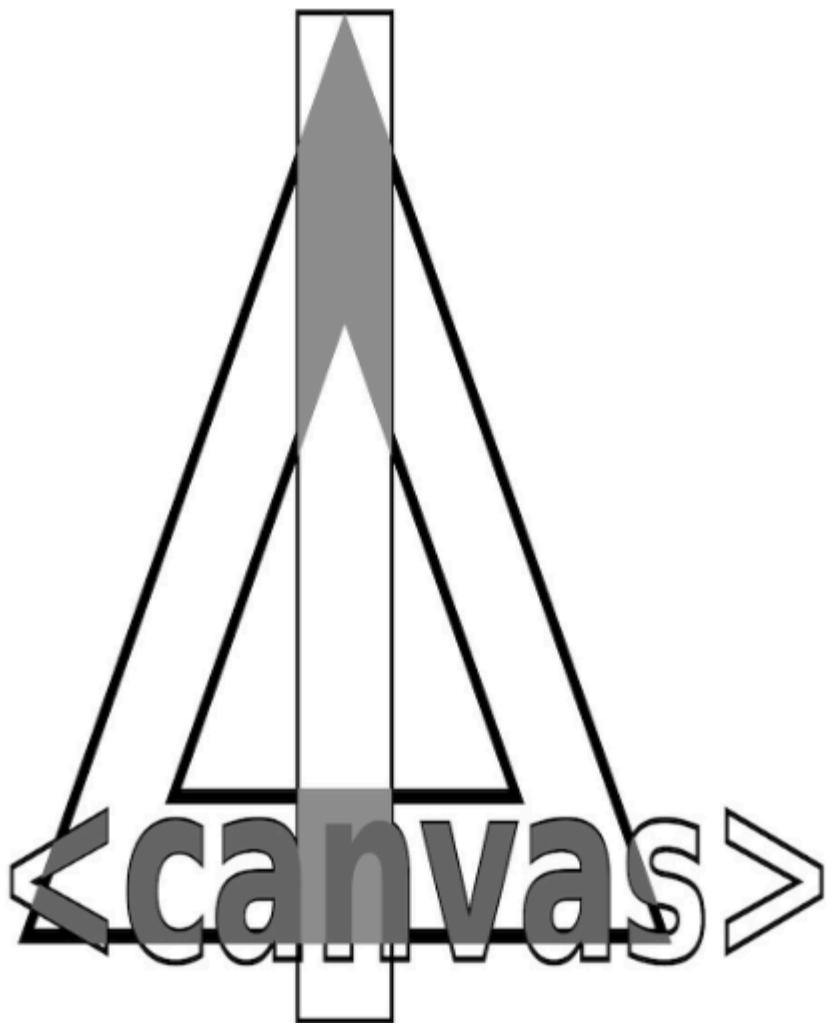


Figura 15-13. Traçados não recortados e preenchimentos recortados

A Figura 15-13 foi gerada usando o método `polygon()` do Exemplo 15-5 e o seguinte código:

```
Defina alguns atributos de  
desenhos.c.font = "bold 60pt sans-    Fonte grande  
serif";
```

```
c.lineWidth = 2;           Linhas estreitas  
c.strokeStyle = "#000";    Linhas pretas  
  
Contorne um retângulo e algum textoc.strokeRect(175, 25, 50,  
325); // Uma faixa vertical no meio c.strokeText("<canvas>",  
15, 330); // Nota strokeText() em vez de fillText()
```

*Defina um caminho complexo com um interior que esteja
fora.polygon(c, 3, 200, 225, 200); // Triângulo
grandepolygon(c, 3, 200, 225, 100, 0, true); // Triângulo
reverso menor dentro*

*Torne esse caminho o recorte
region.c.clip();*

*Trace o caminho com uma linha de 5 pixels, inteiramente
dentro da região de recorte.c.lineWidth = 10; // Metade desta
linha de 10 pixels será cortada c.stroke();*

*Preencha as partes do retângulo e do texto que estão dentro
da região de recorte c.fillStyle = "#aaa"; // Cinza
claroc.fillRect(175, 25, 50, 325); // Preencha a faixa
verticalec.fillStyle = "#888"; // Cinza mais
escuroc.fillText("<canvas>", 15, 330); // Preencha o texto*

É importante notar que quando você chama `clip()`, o caminho atual é recortado para a região de recorte atual, então esse caminho recortado se torna a nova região de recorte. Isso significa que o método `clip()` pode reduzir a região de recorte, mas nunca pode aumentá-la. Não há método para redefinir a região de recorte, portanto, antes de chamar `clip()`, você normalmente deve chamar `save()` para que possa restaurar () posteriormente a região não recortada.

15.8.7 Manipulação de pixels

O método `getImageData()` retorna um objeto `ImageData` que representa os pixels brutos (como componentes R, G, B e A) de uma região retangular da tela. Você pode criar objetos `ImageData` vazios com `createImageData()`. Os pixels em um objeto `ImageData` são graváveis, então você pode defini-los da maneira que quiser e, em seguida, copiá-los de volta para a tela com `putImageData()`.

Esses métodos de manipulação de pixels fornecem acesso de nível muito baixo à tela. O retângulo que você passa para `getImageData()` está no sistema de coordenadas padrão: suas dimensões são medidas em pixels CSS e não é afetado pela transformação atual. Quando você chama `ImageData()`, a posição especificada também é medida no sistema de coordenadas padrão. Além disso, `putImageData()` ignora todos os atributos gráficos. Ele não executa nenhuma composição, não multiplica pixels por `globalAlpha` e não desenha sombras.

Os métodos de manipulação de pixels são úteis para implementar o processamento de imagens. O Exemplo 15-8 mostra como criar um desfoque de movimento simples ou efeito de "mancha" como o mostrado na Figura 15-14.



Figura 15-14. Um efeito de desfoco de movimento criado pelo processamento de imagem

O código a seguir demonstra getImageData() e putImageData() e mostra como iterar e modificar os valores de pixel em um objeto ImageData.

Exemplo 15-8. Desfoco de movimento com ImageData

Espalhe os pixels do retângulo para a direita, produzindo uma espécie de desfoco de movimento como se os objetos estivessem se movendo da direita para a esquerda.// n deve ser 2 ou maior. Valores maiores produzem manchas maiores.// O retângulo é especificado no sistema de coordenadas padrão. função mancha(c, n, x, y, w, h) {

Obtenha o objeto ImageData que representa o retângulo de pixels a serem manchados

```
let pixels = c.getImageData(x, y, w, h);
```

Essa mancha é feita in-loco e requer apenas o sourceImageData.

Alguns algoritmos de processamento de imagem exigem um additionalImageData para

*Armazene valores de pixel transformados. Se precisássemos de um buffer de saída, poderíamos
crie um novo ImageData com as mesmas dimensões assim:*

```
// seja output_pixels = c.createImageData(pixels);
```

Obtenha as dimensões da grade de pixels no objeto ImageData

```
let largura = pixels.largura, altura = pixels.altura;
```

Esta é a matriz de bytes que contém os dados brutos de pixels, da esquerda para a direita e de cima para baixo. Cada pixel ocupa 4 bytes consecutivos na ordem R, G, B, A.

```
let dados = pixels.data;
```

Cada pixel após o primeiro em cada linha é manchado substituindo-o por 1/n-ésimo de seu próprio valor mais m/enésimos do valor do pixel anterior

```
seja m = n-1;
```

```
        for(let linha = 0; linha < altura; linha++) {      Para cada linha
            let i = linha*largura*4 + 4; O deslocamento do segundo
pixel da linha
                for(let col = 1; col < width; col++, i += 4) { // Para
cada coluna
                    dados[i] = (dados[i] + dados[i-4]*m)/n;    Vermelho
componente de pixel
                    Dados[I+1] = (Dados[I+1] + Dados[I-3]*m)/N;    Verde
                    dados[i+2] = (dados[i+2] + dados[i-2]*m)/n;    Azul
                    dados[i+3] = (dados[i+3] + dados[i-
componente 1]*m)/n;
                }
            }
```

*Agora copie os dados da imagem manchada de volta para
a mesma posição na tela*

```
c.putImageData(pixels, x, y);}
```

15.9 APIs de áudio

O HTML <audio> e <video> as tags permitem que você inclua facilmente sons e vídeos em suas páginas da web. Esses são elementos complexos com APIs significativas e interfaces de usuário não triviais. Você pode controlar a reprodução de mídia com os métodos play() e pause(). Você pode definir opropriedades de volume e reproduçãoTaxa para controlar o volume de áudioe a velocidade de reprodução. E você pode pular para um horário específico dentro domídia definindo a propriedade currentTime.

No entanto, não abordaremos <audio> e <video> marcaremos mais detalhes aqui. As subseções a seguir demonstram duas maneiras de adicionar efeitos sonoros com script às suas páginas da Web.

15.9.1 O construtor Audio()

Você não precisa incluir uma <audio> tag em seu documento HTML para incluir efeitos sonoros em suas páginas da web. Você pode criar <audio> elementos dinamicamente com o método DOMdocument.createElement() normal ou, como atalho, pode simplesmente usar o construtor Audio(). Você não precisa adicionar o elemento criado ao seu documento para reproduzi-lo. Você pode simplesmente chamar seu método play():

Carregue o efeito sonoro com antecedência para que esteja pronto para usolet soundeffect = new Audio("soundeffect.mp3");

Reproduza o efeito sonoro sempre que o usuário clicar no mousebuttonondocument.addEventListener("click", () => {

soundeffect.cloneNode().play(); Carregue e reproduza o som});

Observe o uso de cloneNode() aqui. Se o usuário clicar com o mouse rapidamente, queremos ter várias cópias sobrepostas do efeito sonoro sendo reproduzidas ao mesmo tempo. Para fazer isso, precisamos de vários elementos de áudio. Como os elementos de áudio não são adicionados ao documento, eles serão coletados como lixo quando terminarem de ser reproduzidos.

15.9.2 A API WebAudio

Além da reprodução de sons gravados com elementos de áudio, os navegadores da web também permitem a geração e reprodução de sons sintetizados com a API WebAudio. Usar a API WebAudio é como conectar um sintetizador eletrônico antigo com patch cords. Com o WebAudio, você cria um conjunto de objetos AudioNode, que representam origens, transformações ou destinos de formas de onda, e depois os conecta

nós juntos em uma rede para produzir sons. A API não é particularmente complexa, mas uma explicação completa requer uma compreensão dos conceitos de música eletrônica e processamento de sinais que estão além do escopo deste livro.

O código a seguir usa a API WebAudio para sintetizar um acorde curto que desaparece em cerca de um segundo. Este exemplo demonstra as noções básicas da API WebAudio. Se isso for interessante para você, você pode encontrar muito mais sobre esta API online:

```
Comece criando um objeto audioContext. O Safari ainda  
exige // que usemos webkitAudioContext em vez de  
AudioContext.let audioContext = new(this.  
AudioContext||this.webkitAudioContext());
```

```
Defina o som de base como uma combinação de três notas  
senoidais puras = [ 293,7, 370,0, 440,0 ]; Acorde de ré  
maior: D, F# e A
```

```
Crie nós osciladores para cada uma das notas que  
queremosplaylet osciladores = notes.map(note => {  
  
let o = audioContext.createOscillator();  
o.frequency.value = nota; retornar o;});
```

```
Modele o som controlando seu volume ao longo do tempo.//  
Começando no tempo 0, aumente rapidamente para o volume  
máximo.// Em seguida, a partir do tempo 0.1, diminua lentamente  
para 0.let volumeControl = audioContext.createGain();  
volumeControl.gain.setTargetAtTime(1, 0.0, 0.02);  
volumeControl.gain.setTargetAtTime(0, 0.1, 0.2);
```

```
Vamos enviar o som para o padrão destination:// alto-  
falantes do usuáriolet speakers = audioContext.destination;
```

Conekte cada uma das notas de origem ao controle de volumeosciladores.forEach(o => o.connect(volumeControl));

E conepte a saída do controle de volume ao alto-falantes.controle de volume.conectar(alto-falantes);

Agora comece a reproduzir os sons e deixe-os rodar por 1,25 segundos.let startTime = audioContext.currentTime; let stopTime = startTime + 1.25; oscillators.forEach(o => {

*o.start(startTime);
o.stop(stopTime);});*

Se quisermos criar uma sequênciade sons, podemos usar eventhandlersoscillators[0].addEventListener("ended", () => {

Esse manipulador de eventos é invocado quando a nota para de tocar});

15.10 Localização, navegação e histórico

A propriedade location dos objetos Window e Document refere-se ao objeto Location, que representa a URL atual do documento exibido na janela e que também fornece uma API para carregar novos documentos na janela.

O objeto Location é muito parecido com um objeto de URL (§11.9) e você pode usar propriedades como protocolo, nome do host, porta e caminho para acessar as várias partes da URL do documento atual. A propriedade href retorna a URL inteira como uma string, assim como o método `toString()`.

As propriedades de hash e pesquisa do objeto Location são interessantes. A propriedade hash retorna a parte "identificador de fragmento" da URL, se houver: uma marca de hash (#) seguida por um ID de elemento. A propriedade de pesquisa é semelhante. Ele retorna a parte da URL que começa com um ponto de interrogação: geralmente algum tipo de querystring. Em geral, essa parte de uma URL é usada para parametrizar a URL e fornece uma maneira de inserir argumentos nela. Embora esses argumentos sejam geralmente destinados a scripts executados em um servidor, não há razão para que eles também não possam ser usados em páginas habilitadas para JavaScript.

Os objetos de URL têm uma propriedade searchParams que é uma representação analisada da propriedade de pesquisa. O objeto Location não tem uma propriedade searchParams, mas se você quiser analisar window.location.search, basta criar um objeto URL a partir do objeto Location e usar os searchParams da URL:

```
let url = new URL(window.location); let consulta =
urlSearchParams.get("q"); let numResults =
parseInt(urlSearchParams.get("n") || "10");
```

Além do objeto Location que você pode consultar com window.location ou document.location, e o construtor URL() que usamos anteriormente, os navegadores também definem adocument.URL. Surpreendentemente, o valor dessa propriedade não é um objeto de URL, mas apenas uma cadeia de caracteres. A string contém a URL do documento atual.

15.10.1 Carregando novos documentos

Se você atribuir uma string a window.location ou todocument.location, essa string será interpretada como uma URL e o navegador a carregará, substituindo o documento atual por um novo:

```
window.location = "http://www.oreilly.com"; Vá comprar  
alguns livros!
```

Você também pode atribuir URLs relativos ao local. Eles são resolvidosem relação à URL atual:

```
document.location = "page2.html";  
Carregue o próximo  
página
```

Um identificador de fragmento simples é um tipo especial de URL relativo que não faz com que o navegador carregue um novo documento, mas simplesmente role para que o elemento do documento com id ou nome que corresponda ao fragmento fique visível na parte superior da janela do navegador. Como um caso especial, o identificador de fragmento #top faz com que o navegador pule para o início do documento (supondo que nenhum elemento tenha um atributo id="top"):

```
localização = "#top";  
Pule para o  
Parte superior do documento
```

As propriedades individuais do objeto Location são graváveis econfigurá-las altera a URL do local e também faz com que o navegador carregue um novo documento (ou, no caso da propriedade hash, navegue no documento atual):

```
document.location.path = "pages/3.html"; Carregar uma nova  
páginadocument.location.hash = "TOC";// Rolar até a tabela  
de conteúdolocalização.search = "?page=" + (page+1);//  
Recarregar com newquery string
```

Você também pode carregar uma nova página passando uma nova string para o método assign() do objeto Location. No entanto, isso é o mesmo que atribuir a cadeia de caracteres à propriedade location, portanto, não é particularmente interessante.

O método replace() do objeto Location, por outro lado, é bastante útil. Quando você passa uma string para replace(), ela é interpretada como uma URL e faz com que o navegador carregue uma nova página, assim como assign() faz. A diferença é que replace() substitui o documento atual no histórico do navegador. Se um script no documento A define a propriedade location ou chama assign() para carregar o documento Band e o usuário clica no botão Voltar, o navegador volta ao documento A. Se você usar replace() em vez disso, o documento A será apagado do histórico do navegador e, quando o usuário clicar no botão Voltar, o navegador retornará ao documento exibido antes do documento A.

Quando um script carrega incondicionalmente um novo documento, o método replace() é uma escolha melhor do que assign(). Caso contrário, o botão Voltar levaria o navegador de volta ao documento original e o mesmo script carregaria novamente o novo documento. Suponha que você tenha uma versão aprimorada por JavaScript de sua página e uma versão estática que não usa JavaScript. Se você determinar que o navegador do usuário não oferece suporte às APIs da plataforma da Web que deseja usar, use location.replace() para carregar a versão estática:

Se o navegador não suportar as APIs JavaScript de que precisamos, // redirecione para uma página estática que não use JavaScript.if (!isBrowserSupported())

```
localização.replace("staticpage.html");
```

Observe que a URL passada para replace() é relativa. Os URLs relativos são interpretados em relação à página em que aparecem, assim como seriam se fossem usados em um hiperlink.

Além dos métodos assign() e replace(), o objeto Location também define reload(), que simplesmente faz com que o navegador recarregue o documento.

15.10.2 Histórico de navegação

A propriedade history do objeto Window refere-se ao objeto History da janela. O objeto History modela o histórico de navegação de uma janela como uma lista de documentos e estados de documentos. A propriedade length do objeto History especifica o número de elementos na lista de histórico de navegação, mas, por motivos de segurança, os scripts não têm permissão para acessar as URLs armazenadas. (Se pudesse, qualquer script poderia bisbilhotar seu histórico de navegação.)

O objeto History tem métodos back() e forward() que se comportam como os botões Back e Forward do navegador: eles fazem o navegador retroceder ou avançar um passo em seu histórico de navegação. Um terceiro método, go(), recebe um argumento inteiro e pode pular qualquer número de páginas para frente (para argumentos positivos) ou para trás (para argumentos negativos) na lista de histórico:

```
history.go(-2); Volte 2, como clicar no botão Voltar  
duas  
vezeshistória.go(Outra maneira de recarregar a página atual  
0);
```

Se uma janela contiver janelas <iframe> filhas (como elementos), os históricos de navegação das janelas filhas serão intercalados cronologicamente com o histórico da janela principal. Isso significa que chamar `history.back()` (por exemplo) na janela principal pode fazer com que uma das janelas filhas navegue de volta para um documento exibido anteriormente, mas deixa a janela principal em seu estado atual.

O objeto History descrito aqui remonta aos primórdios da web, quando os documentos eram passivos e toda a computação era executada no servidor. Hoje, os aplicativos da Web geralmente geram ou carregam conteúdo dinamicamente e exibem novos estados do aplicativo sem realmente carregar novos documentos. Aplicativos como esses devem realizar seu próprio gerenciamento de histórico se quiserem que o usuário possa usar os botões Voltar e Avançar (ou os gestos equivalentes) para navegar de um estado do aplicativo para outro de forma intuitiva. Existem duas maneiras de fazer isso, descritas nas próximas duas seções.

15.10.3 Gerenciamento de histórico com hashchangeEvents

Uma técnica de gerenciamento de histórico envolve `location.hash` e o evento "hashchange". Aqui estão os principais fatos que você precisa saber para entender essa técnica:

- A propriedade `location.hash` define o identificador de fragmento da URL e é tradicionalmente usada para especificar a ID de uma seção de documento para a qual rolar. Mas `location.hash` não precisa ser um ID de elemento: você pode defini-lo como qualquer string. Enquanto nenhum elemento tiver essa string como seu ID, o navegador não rolará quando você definir a propriedade `hash` como

este. Definir a propriedade `location.hash` atualiza o URL exibido na barra de localização e, muito importante, adiciona uma entrada ao histórico do navegador. Sempre que o identificador de fragmento do documento é alterado, o navegador dispara um evento de "hashchange" no objeto `Window`. Se você definir `location.hash` explicitamente, um evento "hashchange" será disparado. E, como mencionamos, essa alteração no objeto `Location` cria uma nova entrada no histórico de navegação do navegador. Portanto, se o usuário clicar no botão Voltar, o navegador retornará ao URL anterior antes de você definir `location.hash`. Mas isso significa que o identificador do fragmento foi alterado novamente, portanto, outro evento "hashchange" é acionado neste caso. Isso significa que, desde que você possa criar um identificador de fragmento exclusivo para cada estado possível de seu aplicativo, os eventos de "hashchange" irão notificá-lo se o usuário retroceder e avançarem seu histórico de navegação. Para usar esse mecanismo de gerenciamento de histórico, você precisará ser capaz de codificar as informações de estado necessárias para renderizar uma "página" do seu aplicativo em uma sequência de texto relativamente curta que seja adequada para uso como um identificador de fragmento. E você precisará escrever uma função para converter o estado da página em uma string e outra função para analisar a string e recriar o estado da página que ela representa.

Depois de escrever essas funções, o resto é fácil. Defina uma função `window.onhashchange` (ou registre um ouvinte "hashchange" com `addEventListener()`) que lê `location.hash`, converte essa string em uma representação do estado do aplicativo e, em seguida, executa as ações necessárias para exibir esse novo estado do aplicativo.

Quando o usuário interage com seu aplicativo (por exemplo, clicando em um link) de uma forma que faria com que o aplicativo entrasse em um novo estado, não renderize o novo estado diretamente. Em vez disso, codifique o newstate desejado como uma string e defina location.hash como essa string. Isso acionará um evento de "hashchange" e seu manipulador para esse evento exibirá o novo estado. O uso dessa técnica de rotatória garante que o novo estado seja inserido no histórico de navegação para que os botões Voltar e Avançar continuem funcionando.

15.10.4 Gerenciamento de histórico com pushState()

A segunda técnica para gerenciar o histórico é um pouco mais complexa, mas é menos hackeada do que o evento "hashchange". Essa técnica de gerenciamento de histórico mais robusta é baseada no método history.pushState() e no evento "popstate". Quando um aplicativo da web entra em um novo estado, ele chama history.pushState() para adicionar um objeto que representa o estado ao histórico do navegador. Se o usuário clicar no botão Voltar, o navegador disparará um evento "popstate" com uma cópia desse objeto de estado salvo e o aplicativo usará esse objeto para recriar seu estado anterior. Além do objeto de estado salvo, os aplicativos também podem salvar uma URL com cada estado, o que é importante se você deseja que os usuários possam marcar e compartilhar links para os estados internos do aplicativo.

O primeiro argumento para pushState() é um objeto que contém todas as informações de estado necessárias para restaurar o estado atual do documento. Este objeto é salvo usando o algoritmo de clone estruturado do HTML, que é mais versátil que JSON.stringify() e pode suportar objetos Map, Set e Date, bem como matrizes digitadas e ArrayBuffer.

O segundo argumento pretendia ser uma string de título para o estado, mas a maioria dos navegadores não o suporta e você deve apenas passar uma string vazia. O terceiro argumento é uma URL opcional que será exibida na barra de localização imediatamente e também se o usuário retornar a esse estado por meio dos botões Voltar e Avançar. Os URLs relativos são resolvidos em relação ao local atual do documento. Associar uma URL a cada estado permite que o usuário marque os estados internos do seu aplicativo. Lembre-se, porém, de que se o usuário salvar um favorito e visitá-lo um dia depois, você não receberá um evento "popstate" sobre essa visita: você terá que restaurar o estado do aplicativo analisando a URL.

O ALGORITMO DE CLONE ESTRUTURADO

O método `history.pushState()` não usa `JSON.stringify()` ([§11.6](#)) para serializar dados de estado. Em vez disso, ele (e outras APIs de navegador sobre as quais aprenderemos mais tarde) usa uma técnica de serialização mais robusta conhecida como algoritmo de clone estruturado, definido pelo padrão HTML.

O algoritmo de clone estruturado pode serializar qualquer coisa que `JSON.stringify()` pode, mas, além disso, permite a serialização da maioria dos outros tipos de JavaScript, incluindo Map, Set, Date, RegExp e typedarrays, e pode lidar com estruturas de dados que incluem referências循环. No entanto, o algoritmo de clone estruturado não pode serializar funções ou classes. Ao clonar objetos, ele não copia o objeto `prototype`, getters e setters ou propriedades não enumeráveis. Embora o algoritmo de clone estruturado possa clonar a maioria dos tipos JavaScript integrados, ele não pode copiar tipos definidos pelo hostenvironment, como objetos `Element` de documento.

Isso significa que o objeto de estado que você passa para `history.pushState()` não precisa ser limitado aos objetos, matrizes e valores primitivos que `JSON.stringify()` suporta. Observe, no entanto, que se você passar uma instância de uma classe que você definiu, essa instância será serializada como um objeto JavaScript comum e perderá seu protótipo.

Além do método `pushState()`, o objeto `History` também define `replaceState()`, que usa os mesmos argumentos, mas substitui o estado do histórico atual em vez de adicionar um novo estado ao histórico de navegação. Quando um aplicativo que usa `pushState()` é carregado pela primeira vez, geralmente é uma boa ideia chamar `replaceState()` para

Defina um objeto de estado para esse estado inicial do aplicativo.

Quando o usuário navega para estados de histórico salvos usando os botões Voltar ou Avançar, o navegador dispara um evento "popstate" no objeto Window. O objeto de evento associado ao evento tem uma propriedade named state, que contém uma cópia (outro clone estruturado) do objeto state que você passou para pushState().

O exemplo 15-9 é um aplicativo Web simples — o jogo de adivinhação de números ilustrado na Figura 15-15 — que usa pushState() para salvar seu histórico, permitindo que o usuário "volte" para revisar ou refazer suas suposições.

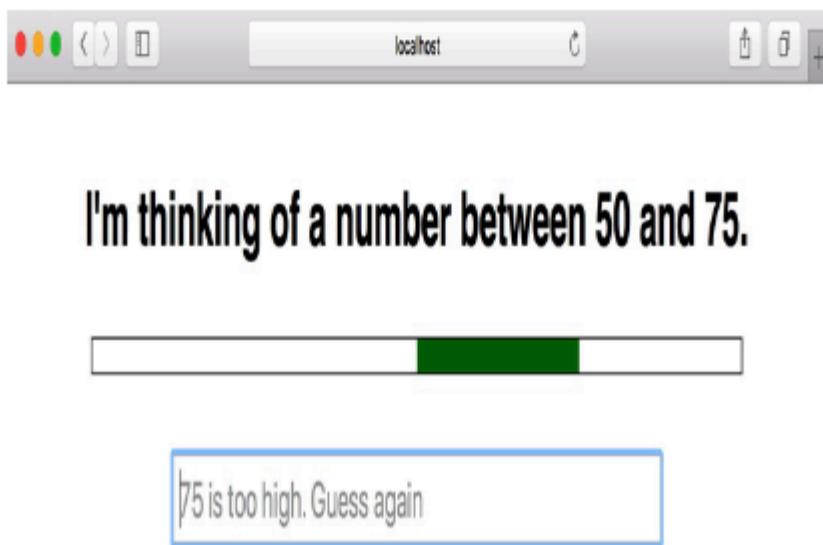


Figura 15-15. Um jogo de adivinhação de números

Exemplo 15-9. Gerenciamento de histórico com pushState()

```
<html><head><title>Estou pensando em um número ...</title><style>
```

```
corpo {altura: 250px; exibição: flex; direção flexível: coluna;
       alinhar itens: centro; justify-content: espaço uniforme; }
#heading { fonte: negrito 36px sans-serif; margem: 0;
}#container { borda: preto sólido 1px; altura: 1em; largura: 80%; }#range { background-color: verde; margem-esquerda: 0%;
altura: 1em; largura: 100%; }#input { display: block; font-size: 24px; width: 60%; padding: 5px; }#playagain {tamanho da
fonte: 24px; preenchimento: 10px; raio da borda: 5px; }
</style></head><body><h1 id="heading">Estou pensando em um
número...</h1><!-- Uma representação visual dos números que
não foram descartados --><div id="container"><div id="range">
</div></div><!-- Onde o usuário insere seu palpite --><input
id="input" type="text"><!-- Um botão que é recarregado sem
string de pesquisa. Oculto até o fim do jogo. --><button
id="playagain" hidden onclick="location.search='';">Jogar
novamente</button><script>/**
```

```
* Uma instância desta classe GameState representa o  
internalstate de  
* Nosso jogo de adivinhação de números. A classe define  
métodos de fábrica estáticos para  
* inicializando o estado do jogo a partir de diferentes  
fontes, um método para  
* atualizando o estado com base em uma nova suposição e um  
método para modificar o  
* documento baseado no estado  
atual.*/class GameState {
```

```

        s.high = 100;           Os palpites devem ser
menos do que isso
        s.numGuesses = 0;       Quantos palpites
foram feitas
        s.guess = nulo;       Qual é o último palpite
Foi
        retornar s;}

```

Quando salvamos o estado do jogo com `history.pushState()`, é apenas um objeto JavaScript simples que é salvo, não uma instância de `GameState`.

Portanto, essa função de fábrica recria um objeto `GameState` com base no

```

        objeto simples que obtemos de um evento
        popstate.static fromStateObject(stateObject) {
            let s = new GameState(); for(let chave de
            Object.keys(stateObject)) {
                s[chave] =
                stateObject[chave];}retornar s;}

```

Para habilitar a marcação, precisamos ser capazes de codificar o estado de qualquer jogo como um URL. Isso é fácil de fazer com `URLSearchParams`.

```

        toURL() {
            let url = new URL(window.location);
            url.searchParams.set("l", this.low);
            url.searchParams.set("h", this.high);
            url.searchParams.set("n", this.numGuesses);
            url.searchParams.set("g", this.guess); // Observe que
            não podemos codificar o número secreto no
url ou TI
            vai revelar o segredo. Se o usuário marcar o
página com
            esses parâmetros e, em seguida, retorna a ele,
basta escolher um
            novo número aleatório entre baixo e
alto.return url.href;}
```

Esta é uma função de fábrica que cria um novo objeto GameState e inicializa-o a partir da URL especificada. Se o URL não contiver os parâmetros esperados ou se eles estiverem malformados, ele apenas retorna nulo.

```
estático fromURL(url) {  
    let s = new GameState(); let params =  
    new URL(url).searchParams; s.low =  
    parseInt(params.get("l")); s.high =  
    parseInt(params.get("h")); s.numGuesses  
    = parseInt(params.get("n")); s.palpite =  
    parseInt(params.get("g"));
```

Se o URL estiver faltando algum dos parâmetros de que precisamos ou se os parâmetros não forem analisados como inteiros e, em seguida, retornarem nulos; if (isNaN(s.low) || isNaN(s.high) || isNaN(s.numGuesses) || isNaN(s.guess)) {return null;}

Escolha um novo número secreto no intervalo certo a cada tempo nós

```
restaurar um jogo de um URL.s.secret =  
s.randomUUID(s.low, s.high); retornar s;}
```

*Retorna um inteiro n, min < n < maxrandomInt(min, max) {
 return min + Math.ceil(Math.random() * (max - min -
1));
}*

Modifique o documento para exibir o estado atual do jogo.

```
render() {  
    let cabeçalho = document.querySelector("#heading"); //  
0 <h1> no topo  
    let intervalo = document.querySelector("#range"); //  
Exibir intervalo de palpites
```

```
let input = document.querySelector("#input");      //  
Campo de entrada de adivinhação  
let playagain = document.querySelector("#playagain");  
  
Atualize o título do documento e  
titleheading.textContent = document.title =  
    'Estou pensando em um número entre ${this.low} e  
${this.high}.';  
  
Atualize o intervalo visual de  
numbersrange.style.marginLeft = '${this.low}%';  
range.style.width = '${(this.high-this.low)}%';  
  
Certifique-se de que o campo de entrada esteja  
vazio e focado.input.value = ""; input.focus();  
  
Exiba comentários com base no último palpite do usuário. O  
entrada  
espaço reservado será exibido porque fizemos a entrada  
vazio.  
if (this.guess === null) {  
    input.placeholder = "Digite seu palpite e acerte  
Entre";  
} else if (this.guess < this.secret) {  
    input.placeholder = '${this.guess} é muito baixo.  
Adivinhe de novo';  
} else if (this.guess > this.secret) {  
    input.placeholder = '${this.guess} é muito alto.  
Adivinhe de novo';  
} else {  
    input.placeholder = document.title = '${this.guess}  
está correto!';  
    heading.textContent = 'Você ganha em  
${this.numPalpites} palpites!';  
    playagain.hidden = false;}}
```

*Atualize o estado do jogo com base no que o usuário
adivinhou.*

*Retorna true se o estado foi atualizado e false caso
contrário.*

```

updateForGuess(guess) {
    Se for um número e estiver no intervalo certo if
        ((palpite > this.low) && (palpite < this.high)) {
            Atualize o objeto de estado com base neste guess if
                (palpite < this.secret) this.low = adivinhe; senão
                se (palpite > this.secret) this.high = palpite;
                this.guess = adivinar; this.numGuesses++; retornar
                verdadeiro;} else { // Um palpite inválido:
                notificar o usuário, mas não
                estado de atualização
                alert('Insira um número maior que ${this.low} e menor que ${this.high}');
            return false;}}
}

```

Com a classe GameState definida, fazer o jogo funcionar é apenas uma questão de inicializar, atualizar, salvar e renderizar o objeto de estado nos momentos apropriados.

Quando somos carregados pela primeira vez, tentamos obter o estado do jogo da URL // e, se isso falhar, iniciamos um novo jogo. Portanto, se o usuário marcar um jogo, esse jogo poderá ser restaurado a partir do URL. Mas se carregarmos uma página com // sem parâmetros de consulta, obteremos apenas um novo game.let gamestate = GameState.fromURL(window.location)||GameState.newGame();

Salve este estado inicial do jogo no browserhistory, mas use// replaceState em vez de pushState() para esta pagehistory.replaceState(gamestate, "", gamestate.toURL());

*Exibe este
stategamestate.render() inicial;*

```

Quando o usuário adivinhar, atualize o estado do jogo com base
em seu palpite// e, em seguida, salve o novo estado no
histórico do navegador e renderize o novo
estadodocumento.querySelector("#input").onchange = (evento) =>
{
  if (gamestate.updateForGuess(parseInt(event.target.value))){}

  history.pushState(gamestate, "",
    gamestate.toURL());}gamestate.render();};

Se o usuário voltar ou avançar no histórico, obteremos um
evento popstate // no objeto window com uma cópia do
objeto state que salvamos com // pushState. Quando isso
acontecer, renderize o novo estado.window.onpopstate =
(evento) => {

  gamestate = GameState.fromStateObject(event.state);
  Restaurar o estado
    gamestate.render();                                     e
  exiba};
</script>
</body>
</html>

```

15.11 Rede

Toda vez que você carrega uma página da Web, o navegador faz solicitações de rede - usando os protocolos HTTP e HTTPS - para um arquivo HTML, bem como as imagens, fontes, scripts e folhas de estilo das quais o arquivo depende. Mas além de poder fazer solicitações de rede em resposta às ações do usuário, os navegadores da Web também expõem APIs JavaScript para rede.

Esta seção aborda três APIs de rede:

- O método fetch() define uma API baseada em promessa para fazer solicitações HTTP e HTTPS. A API fetch() simplifica as solicitações GET básicas, mas possui um conjunto de recursos abrangente que também oferece suporte a praticamente qualquer caso de uso HTTP possível. A API Server-Sent
- Events (ou SSE) é uma interface conveniente e baseada em eventos para técnicas de "sondagem longa" HTTP, onde o servidor web mantém a conexão de rede aberta para que possa enviar dados ao cliente sempre que quiser. WebSockets é um protocolo de rede que não é HTTP, mas é projetado para interoperar com HTTP. Ele define a API de passagem de mensagens assíncrona em que clientes e servidores podem enviar e receber mensagens uns dos outros de maneira semelhante aos soquetes de rede TCP.

15.11.1 fetch()

Para solicitações HTTP básicas, o uso de fetch() é um processo de três etapas:

1. Chame fetch(), passando a URL cujo conteúdo você deseja recuperar.
2. Obtenha o objeto de resposta que é retornado de forma assíncrona por passo 1 quando a resposta HTTP começa a chegar e chame o método `body` deste objeto de resposta para solicitar o corpo da resposta.
3. Obtenha o objeto `body` que é retornado de forma assíncrona pela etapa 2 e processe-o como desejar. A API fetch() é totalmente baseada em promessas e há duas etapas assíncronas aqui, então você normalmente espera duas chamadas `then()` ou duas expressões de espera ao usar `fetch()`. (E se você tiver

esqueceu o que são, você pode querer reler o Capítulo 13 antes de continuar com esta seção.)

Veja como é uma solicitação `fetch()` se você estiver usando `then()` e esperar que a resposta do servidor à sua solicitação seja formatada em JSON:

```
fetch("/api/usuários/atual")           Fazer um HTTP (ou  
HTTPS) GET  
.then(response => response.json()) // Analisa seu corpo como  
um objeto JSON  
.then(currentUser => {             Em seguida, processe isso  
objeto analisado  
    displayUserInfo(currentUser);});
```

Aqui está uma solicitação semelhante feita usando as palavras-chave `async` e `await` para uma API que retorna uma string simples em vez de um objeto JSON:

```
função assíncrona isServiceReady() {  
let resposta = await fetch("/api/serviço/status");  
let corpo = await resposta.text(); corpo de retorno  
== "pronto";}
```

Se você entender esses dois exemplos de código, saberá 80% do que precisa saber para usar a API `fetch()`. As subseções a seguir demonstrarão como fazer solicitações e receber respostas um pouco mais complicadas do que as mostradas aqui.

ADEUS XMLHTTPREQUEST

A API `fetch()` substitui a API `XMLHttpRequest` barroca e enganosamente chamada (que não tem nada a ver com XML). Você ainda pode ver `XHR` (como é frequentemente abreviado) no código existente, mas não há razão hoje para usá-lo em um novo código, e ele não está documentado neste capítulo. No entanto, há um exemplo de `XMLHttpRequest` neste livro, e você pode consultar §13.1.3 se quiser ver um exemplo de rede JavaScript de estilo antigo.

CÓDIGOS DE STATUS HTTP, CABEÇALHOS DE RESPOSTA E ERROS DE REDE

O processo fetch() de três etapas mostrado em §15.11.1 elimina todo o código de tratamento de erros. Aqui está uma versão mais realista:

```
fetch("/api/usuários/atual")  Fazer um HTTP (ou HTTPS) GET
pedir.
    .then(resposta => {          Quando obtivermos uma resposta,
primeiro verifique
        if (resposta.ok &&      para um código de sucesso e o
tipo esperado.
            response.headers.get("Tipo de conteúdo") ===
"aplicativo/json") {
                retornar response.json(); Retornar uma promessa para
o corpo.
            } else {
                lançar novo Erro(      Ou lance um erro.
                    'Status de resposta inesperada
${response.status} ou tipo de conteúdo'
                );
            }
        then(currentUser => {
            Quando o response.json()
Promessas são resolvidas
            displayUserInfo(currentUser); fazer algo com
o corpo analisado.
        }).catch(erro =>
        {
            Ou se algo deu errado,
basta registrar o erro.
            Se o navegador do usuário estiver offline, fetch() em si
rejeitará.
            Se o servidor retornar uma resposta ruim, lançaremos
um erro acima.
            console.log("Erro ao buscar o usuário atual:",
erro);
        });
    };
};});
```

A promessa retornada por `fetch()` é resolvida para um objeto `Response`. A propriedade `status` desse objeto é o código de status HTTP, como `200` para solicitações bem-sucedidas ou `404` para respostas "Não encontrado". (`statusText` fornece o texto padrão em inglês que acompanha o código de status numérico.) Convenientemente, a propriedade `ok` de uma resposta é verdadeira se o `status` for `200` ou qualquer código entre `200` e `299` e é falsa para qualquer outro código.

`fetch()` resolve sua promessa quando a resposta do servidor começa a chegar, assim que o status HTTP e os cabeçalhos de resposta estão disponíveis, mas normalmente antes que o corpo completo da resposta chegue. Mesmo que o corpo ainda não esteja disponível, você pode examinar os cabeçalhos nesta segunda etapa do processo de busca. A propriedade `headers` de um objeto `Response` é um objeto `Headers`. Use seu método `has()` para testar a presença de um cabeçalho ou use seu método `get()` para obter o valor de um cabeçalho. Os nomes de cabeçalho HTTP não diferenciam maiúsculas de minúsculas, portanto, você pode passar nomes de cabeçalho em minúsculas ou maiúsculas para essas funções.

O objeto `Headers` também é iterável se você precisar fazer isso:

```
fetch(url).then(resposta => {
  for(let [nome,valor] de response.headers) {
    console.log('${nome}: ${valor}');}});
```

Se um servidor web responder à sua solicitação `fetch()`, a promessa que foi retornada será cumprida com um objeto `Response`, mesmo que a resposta do servidor tenha sido um erro `404 Not Found` ou um `500 Internal Server Error`. `fetch()` só rejeita a promessa que retorna se não puder entrar em contato

o servidor web em tudo. Isso pode acontecer se o computador do usuário estiver offline, o servidor não responder ou a URL especificar um nome de host que não existe. Como essas coisas podem acontecer em qualquer solicitação de rede, é sempre uma boa ideia incluir uma cláusula `.catch()` sempre que você fizer uma chamada `fetch()`.

DEFININDO PARÂMETROS DE SOLICITAÇÃO às vezes que você deseja passar parâmetros extras junto com o URL quando você faz uma solicitação. Isso pode ser feito adicionando pares de nome/valor no final de uma URL após um `?`. As classes `URL` e `URLSearchParams` (que foram abordadas em §11.9) facilitam a construção de URLs neste formato, e a função `fetch()` aceita objetos de URL como seu primeiro argumento, para que você possa incluir parâmetros de solicitação em uma solicitação `fetch()` como esta:

```
Função assíncrona search(term) {  
  let url = new URL("/api/search");  
  url.searchParams.set("q", termo); let resposta = await  
  fetch(url); if (!response.ok) throw new  
  Error(response.statusText); let resultsArray = await  
  response.json(); return resultsArray;}
```

DEFININDO CABEÇALHOS DE SOLICITAÇÃO Some vezes você precisa definir cabeçalhos em suas solicitações `fetch()`. Se você estiver fazendo solicitações de API Web que exigem credenciais, por exemplo, talvez seja necessário incluir um cabeçalho `Authorization` que contenha essas credenciais. Para fazer isso, você pode usar a versão de dois argumentos de `fetch()`. Como antes, o primeiro argumento é uma string ou URL

que especifica a URL a ser buscada. O segundo argumento é um objeto que pode fornecer opções adicionais, incluindo cabeçalhos de solicitação:

```
let authHeaders = new Headers(); // Não use  
autenticação básica a menos que seja sobre um  
HTTPSConection.authHeaders.set("Authorization",  
  
    'Básico ${btoa('${nome de usuário}:${senha}')}' );  
fetch("/api/users/", { cabeçalhos: authHeaders })  
    .then(resposta => response.json())  
    .then(usersList => displayAllUsers(usersList));  
manuseio omitido...  
    .then(usersList => displayAllUsers(usersList));
```

Existem várias outras opções que podem ser especificadas no secondargument para fetch(), e veremos isso novamente mais tarde. Uma alternativa parapassar dois argumentos para fetch() é passar os mesmos dois argumentos para o construtor Request() e, em seguida, passar o objeto resultingRequest para fetch():

```
let solicitação = new Solicitação(url, { cabeçalhos  
}); fetch(request).then(resposta => ...);
```

ANALISANDO CORPOS DE RESPOSTA No processo fetch() de três etapas que demonstramos, a segunda etapa termina chamando os métodos json() ou text() do objeto Response e retornando o objeto Promise que esses métodos retornam. Em seguida, a terceira etapa começa quando essa promessa é resolvida com o corpo da resposta analisado como um objeto JSON ou simplesmente como uma string detexto.

Esses são provavelmente os dois cenários mais comuns, mas não são as únicas maneiras de obter o corpo da resposta de um servidor web. Em

além de json() e text(), o objeto Response também tem estes métodos:

arrayBuffer()

Esse método retorna uma promessa que é resolvida para um ArrayBuffer. Isso é útil quando a resposta contém dados binários. Você pode usar o ArrayBuffer para criar uma matriz tipada (§11.2) ou um objeto DataView (§11.2.5) a partir do qual você pode ler os dados binários.

blob()

Esse método retorna uma Promise que é resolvida para um objeto Blob. Os blobs não são abordados em detalhes neste livro, mas o nome significa "Objeto Grande Binário" e são úteis quando você espera grandes quantidades de dados binários. Se você solicitar o corpo da resposta como aBlob, a implementação do navegador poderá transmitir os dados de resposta para um arquivo temporário e, em seguida, retornar um objeto Blob que representa esse arquivo temporário. Os objetos Blob, portanto, não permitem acesso aleatório ao corpo da resposta da maneira que um ArrayBuffer faz. Depois de ter um Blob, você pode criar uma URL que se refira a ele com URL.createObjectURL() ou pode usar a API FileReader baseada em eventos para obter de forma assíncrona o conteúdo do Blob como uma cadeia de caracteres ou um ArrayBuffer. No momento da redação deste artigo, alguns navegadores também definem métodos text() e arrayBuffer() baseados em promessas que fornecem uma rota mais direta para obter o conteúdo de um Blob.

formData()

Esse método retorna uma Promise que é resolvida para um objeto FormData. Você deve usar esse método se espera que o corpo da resposta seja codificado no formato "multipart/form-data". Esse formato é comum em solicitações POST feitas a um servidor, mas respostas incomuns no servidor, portanto, esse método não é usado com frequência.

CORPOS DE RESPOSTA DE STREAMING Em adição aos cinco métodos de resposta que retornam de forma assíncrona alguma forma do corpo de resposta completo para você, há também uma opção para transmitir o corpo da resposta, o que é útil se houver algum tipo de processamento que você possa fazer nas partes do corpo da resposta à medida que elas chegam pela rede. Mas transmitir a resposta também é útil se você quiser exibir uma barra de progresso para que o usuário possa ver o progresso do download.

A propriedade body de um objeto Response é um objeto ReadableStream. Se você já chamou um método de resposta como text() ou json() que lê, analisa e retorna o corpo, então bodyUsed será true para indicar que o fluxo do corpo já foi lido. Se bodyUsed for false, no entanto, o fluxo ainda não foi lido. Nesse caso, você pode chamar getReader() em response.body para obter um objeto Streamreader e, em seguida, usar o método read() desse objeto leitor para ler de forma assíncrona pedaços de texto do fluxo. O método read() retorna uma promessa que é resolvida para um objeto com as propriedades done e value. done será verdadeiro se todo o corpo tiver sido read or se o fluxo foi fechado. E o valor será o próximo pedaço, como um Uint8Array, ou indefinido se não houver mais pedaços.

Essa API de streaming é relativamente simples se você usar `async` and `await`, mas é surpreendentemente complexa se você tentar usá-la com `rawPromises`. O exemplo 15-10 demonstra a API definindo a função `astreamBody()`. Suponha que você queira baixar um arquivo JSON grande e relatar o progresso do download para o usuário. Você não pode fazer isso

com o método json() do objeto Response, mas você pode fazer isso com a função streamBody(), assim (supondo que uma função updateProgress() seja definida para definir o atributo value em um <progress> elemento HTML):

```
fetch('big.json')
  .then(resposta => streamBody(resposta,
    updateProgress)).then(bodyText =>
  JSON.parse(bodyText)).then(handleBigJSONObject);
```

A função streamBody() pode ser implementada conforme mostrado no Exemplo 15-10.

Exemplo 15-10. Streaming do corpo da resposta de uma solicitação fetch()

```
/**  
 * Uma função assíncrona para transmitir o corpo do  
 objeto aResponse  
 * obtido de uma solicitação fetch(). Passe o objeto Response  
 como o primeiro  
 * seguido por dois retornos de chamada opcionais.** Se  
 você especificar uma função como o segundo argumento,  
 thatreportProgress  
  
 * O retorno de chamada será chamado uma vez para cada  
 parte recebida. O primeiro  
 * é o número total de bytes recebidos até o momento. O  
 segundo  
 * é um número entre 0 e 1 especificando o quanto completo o  
 download  
 * É. Se o objeto Response não tiver cabeçalho "Content-  
 Length", no entanto,  
 * este segundo argumento sempre será NaN.** Se você quiser  
 processar os dados em partes à medida que eles chegam,  
 especifique uma função * como o terceiro argumento. Os  
 pedaços serão passados, como Uint8Array  
  
 * objetos, para este retorno de chamada processChunk.**  
 streamBody() retorna uma promessa que resolve para uma  
 string. Se
```

```

um processChunk
* retorno de chamada foi fornecido, então esta string é a
concatenação dos valores
* retornado por esse retorno de chamada. Caso contrário,
a string é a concatenação de
* os valores dos chunks convertidos em strings
UTF-8.*/async function streamBody(response,
reportProgress,processChunk) {

Quantos bytes estamos esperando, ou NaN se nenhum cabeçalho
expectedBytes = parseInt(response.headers.get("Content-
Length"));

    let bytesRead = 0;                                Quantos bytes
recebido até agora
    let leitor = resposta.corpo.getReader(); Ler bytes com
esta função
    Seja decodificador = Novo TextDecode("UTF-8"); Para conversão
bytes para texto
    seja corpo = "";                                Texto lido assim
longe

while(true) { // Loop até sairmos abaixo

    let {done, value} = await reader.read(); Leia um
pedaço

        if (valor) { // Se obtivermos
uma matriz de bytes:
            if (processChunk) {                      Processo
os bytes se
                let processado = processChunk(valor); um
o retorno de chamada foi passado.
                if (processado) {
                    corpo += processado;}}
                else {

```

Caso contrário, converta bytes

```

                    corpo += decoder.decode(valor, {stream: true}); para texto.
                }

```

o retorno de chamada de progresso foi *Se um*

```

        bytesRead += valor.comprimento;    Passado
então chame isso
        reportProgress(bytesRead, bytesRead /
expectedBytes);
    } } if
        (feito) {
            Se isso for
o último pedaço,
            quebrar;
        Saia do
laço
    }
}

    corpo de      Retornar o corpo do texto que acumulamos
}    retorno;

```

Essa API de streaming é nova no momento da redação deste artigo e espera-se que evolua. Em particular, há planos para tornar os objetos ReadableStream de forma assíncrona iteráveis para que possam ser usados com for/awaitloops (§13.4.1).

ESPECIFICANDO O MÉTODO REQUEST E REQUESTBODY Na cada um dos exemplos `fetch()` mostrados até agora, fizemos uma solicitação HTTP (ou HTTPS) GET. Se você quiser usar um método de solicitação diferente (como POST, PUT ou DELETE), basta usar a versão de dois argumentos de `fetch()`, passando um objeto Options com um parâmetro de método:

```

fetch(url, { method: "POST" }).then(r
=>r.json()).then(handleResponse);

```

As solicitações POST e PUT normalmente têm um corpo de solicitação contendo dados a serem enviados ao servidor. Contanto que a propriedade `method` não esteja definida como

"GET" ou "HEAD" (que não suportam corpos de solicitação), você pode especificar um corpo de solicitação definindo a propriedade body do objeto Options:

```
fetch(url, {  
  método: "POST", corpo:  
  "olá mundo"})
```

Quando você especifica um corpo de solicitação, o navegador adiciona automaticamente um cabeçalho "Content-Length" apropriado à solicitação. Quando o corpo é uma cadeia de caracteres, como no exemplo anterior, o navegador padroniza o cabeçalho "Content-Type" como "text/plain; charset=UTF-8." Talvez seja necessário substituir esse padrão se especificar um corpo de string de algum tipo mais específico, como "text/html" ou "application/json":

```
fetch(url, {  
  método: "POST", cabeçalhos: novos  
  cabeçalhos({ "Content-  
  Type": "application/json" }),  
  corpo: JSON.stringify(requestBody) })
```

A propriedade body do objeto de opções fetch() não precisa ser uma string. Se você tiver dados binários em uma matriz tipada ou em um objeto DataView ou em um ArrayBuffer, poderá definir a propriedade body como esse valor e especificar um cabeçalho "Content-Type" apropriado. Se você tiver dados binários no formato Blob, basta definir o corpo como o Blob. Os blobs têm uma propriedade type que especifica seu tipo de conteúdo e o valor dessa propriedade é usado como o valor padrão do cabeçalho "Content-Type".

Com solicitações POST, é um pouco comum passar um conjunto de

name/value no corpo da solicitação (em vez de codificá-los na parte da consulta da URL). Existem duas maneiras de fazer isso:

Você pode especificar seus nomes e valores de parâmetro com `withURLSearchParams` (que vimos anteriormente nesta seção e que está documentado em §11.9) e, em seguida, passar o objeto `URLSearchParams` como o valor da propriedade `body`. Se você fizer isso, o corpo será definido como uma string que se parece com a parte de consulta de um URL, e o cabeçalho "Content-Type" será automaticamente definido como "application/x-www-form-urlencoded; charset=UTF-8." Se, em vez disso, você especificar seus nomes e valores de parâmetro com um objeto `FormData`, o corpo usará uma codificação multipartencoding mais detalhada e "Content-Type" será definido como "multipart/form-data; limite=..." com uma cadeia de caracteres de limite exclusiva que corresponde ao corpo. O uso de um objeto `FormData` é particularmente útil quando os valores que você deseja carregar são longos ou são objetos `File` ou `Blob` que podem ter seu próprio "Content-Type". Os objetos `FormData` podem ser criados e inicializados com valores ignorando um `<form>` elemento para o construtor `FormData()`. Mas você também pode criar corpos de solicitação "multipart/form-data" invocando o construtor `FormData()` sem argumentos e inicializando os pares de nome/valor que ele representa com os métodos `theSet()` e `append()`.

UPLOAD DE ARQUIVO COM FETCH() O upload de arquivos do computador de um usuário para um servidor web é uma tarefa comum e pode ser realizado usando um objeto `FormData` como o corpo da solicitação. Uma maneira comum de obter um objeto `File` é exibir um elemento `<input type="file">` em sua página da Web e ouvir eventos "change" nesse elemento. Quando ocorre um evento de "alteração", os arquivos

do elemento de entrada deve conter pelo menos um objeto File. Os objetos de arquivo também estão disponíveis por meio da API HTML de arrastar e soltar. ThatAPI não é abordada neste livro, mas você pode obter arquivos da matriz dataTransfer.files do objeto de evento passado para um eventlistener para eventos de "descartar".

Lembre-se também de que os objetos File são um tipo de Blob e, às vezes, pode ser útil carregar Blobs. Suponha que você tenha escrito um aplicativo da Web que permite ao usuário criar desenhos em um <canvas>elemento. Você pode carregar os desenhos do usuário como arquivos PNG com o seguinte código:

```
A função canvas.toBlob() é baseada em retorno de
chamada.// Este é um wrapper baseado em promessa
para a função it.async getCanvasBlob(canvas) {
    return new Promise((resolve, reject) => {
        canvas.toBlob(resolve);});}
```

```
Veja como carregamos um arquivo PNG de uma
função canvasasync uploadCanvasImage(canvas) {
let pngblob = await getCanvasBlob(canvas); let formdata =
new FormData(); FormData.Set("CanvasImage", PngBlob); let
resposta = await fetch("/upload", { método: "POST", corpo:
formdata });

let body = await response.json();}
```

SOLICITAÇÕES DE ORIGEM CRUZADAModo tempo em que o `fetch()` é usado por aplicativos da Web para solicitar dados de seu próprio servidor da Web. Solicitações como essas são conhecidas como mesma origem

requests porque a URL passada para fetch() tem a mesma origem (protocolo mais nome do host mais porta) que o documento que contém o script que está fazendo a solicitação.

Por motivos de segurança, os navegadores da Web geralmente não permitem (embora haja exceções para imagens e scripts) solicitações de rede entre origens. No entanto, o Compartilhamento de Recursos entre Origens, ou CORS, permite solicitações de origem cruzada segura. Quando fetch() é usado com um cross-originURL, o navegador adiciona um cabeçalho "Origin" à solicitação (e não permite que ele seja substituído por meio da propriedade headers) para notificar o servidor web de que a solicitação está vindo de um documento com uma origem diferente. Se o servidor responder à solicitação com um cabeçalho "Access-Control-Allow-Origin" apropriado, a solicitação continuará. Caso contrário, se o servidor não permitir explicitamente a solicitação, a Promise retornada por fetch() será rejeitada.

ABORTANDO UMA SOLICITAÇÃO *vezes você pode querer abortar uma solicitação fetch() que você já emitiu, talvez porque o usuário clicou em um botão Cancelar ou a solicitação está demorando muito. A API de busca permite que as solicitações sejam anuladas usando as classes AbortController e AbortSignal. (Essas classes definem um mecanismo de anulação genérico adequado para uso por outras APIs também.)*

Se você quiser ter a opção de abortar uma solicitação fetch(), crie um objeto AbortController antes de iniciar a solicitação. A propriedade signal do objeto controlador é um objeto AbortSignal. Passe este objeto de sinal como o valor da propriedade de sinal do

options que você passa para fetch(). Feito isso, você pode chamar o método abort() do objeto controlador para abortar a solicitação, o que fará com que todos os objetos Promise relacionados à solicitação de busca rejeitem com uma exceção.

Aqui está um exemplo de como usar o mecanismo AbortController para impor um tempo limite para solicitações de busca:

```
Esta função é como fetch(), mas adiciona suporte para
atimeout// propriedade no objeto options e aborta a busca
se ela não estiver completa// dentro do número de
milissegundos especificado por thatproperty.function
fetchWithTimeout(url, options={} {

    if (opções.timeout) {   Se a propriedade timeout existir
        e é diferente de zero
            let controlador = new AbortController(); Crie um
            controlador
            opções.sinal = controlador.sinal;           Defina o
            Propriedade do sinal
            Inicie um temporizador que enviará o sinal de aborto
            após o especificado
            número de milissegundos se passaram. Observe que nós
            nunca cancelo
            este cronômetro. Chamar abort() depois que a busca for
            completo tem
            no effect.setTimeout(() => {
                controller.abort(); },
            options.timeout);
    } // Agora é só executar um
    fetchreturn fetch(url, options);}


```

OPÇÕES DE SOLICITAÇÃO DIVERSOS Você viu que um objeto Options pode ser passado como o segundo argumento para fetch() (ou como o segundo argumento para o Request())

constructor) para especificar o método de solicitação, os cabeçalhos de solicitação e o corpo da solicitação. Ele também suporta várias outras opções, incluindo estas:

cache

Use essa propriedade para substituir o comportamento de cache padrão do navegador. O cache HTTP é um tópico complexo que está além do escopo deste livro, mas se você souber algo sobre como ele funciona, poderá usar os seguintes valores legais de cache:

"padrão"

Esse valor especifica o comportamento de cache padrão. As respostas novas no cache são servidas diretamente do cache e as respostas obsoletas são revalidadas antes de serem servidas.

"sem loja"

Esse valor faz com que o navegador ignore seu cache. O cache não é verificado quanto a correspondências quando a solicitação é feita e não é atualizado quando a resposta chega.

"recarregar"

Esse valor informa ao navegador para sempre fazer uma solicitação de rede normal, ignorando o cache. Quando a resposta chega, no entanto, ela é armazenada no cache.

"sem cache"

Esse valor (nomeado de forma enganosa) informa ao navegador para não fornecer novos valores do cache. Os valores em cache novos ou obsoletos são revalidados antes de serem retornados.

"Ocultação da força"

Esse valor informa ao navegador para fornecer respostas do cachê mesmo que estejam obsoletas.

redirecionar

Essa propriedade controla como o navegador lida com respostas de redirecionamento do servidor. Os três valores legais são:

"seguir"

Esse é o valor padrão e faz com que o navegador sigare direcionamentos automaticamente. Se você usar esse padrão, os objetos Response que você obtém com fetch() nunca devem ter um status no intervalo de 300 a 399.

"erro"

Esse valor faz com que fetch() rejeite sua promessa retornada se o servidor retornar uma resposta de redirecionamento.

"manual"

Esse valor significa que você deseja manipular manualmente as respostas de redirecionamento, e a Promise retornada por fetch() pode resolver para um objeto Response com um status no intervalo de 300 a 399. Nesse caso, você terá que usar o cabeçalho "Local" doResposta para seguir manualmente o redirecionamento.

Referrer

Você pode definir essa propriedade como uma string que contém uma URL relativa para especificar o valor do cabeçalho HTTP "Referer" (que é historicamente incorreto com três Rs em vez de quatro). Se você definir thisproperty como a string vazia, o cabeçalho "Referer" será omitido da solicitação.

15.11.2 Eventos enviados pelo servidor

Uma característica fundamental do protocolo HTTP sobre o qual a web é

construído é que os clientes iniciam solicitações e os servidores respondem a essas solicitações. Alguns aplicativos da web acham útil, no entanto, que seus servidores enviem notificações quando ocorrem eventos. Isso não é natural para o HTTP, mas a técnica que foi desenvolvida é que o cliente faça uma solicitação ao servidor e, em seguida, nem o cliente nem o servidor fechem a conexão. Quando o servidor tem algo para contar ao cliente, ele grava dados na conexão, mas a mantém aberta. O efeito é como se o cliente fizesse uma solicitação de rede e o servidor respondesse de maneira lenta e intermitente com pausas significativas entre picos de atividade. Conexões de rede como essa geralmente não ficam abertas para sempre, mas se o cliente detectar que a conexão foi fechada, ele pode simplesmente fazer outra solicitação para reabrir a conexão.

Essa técnica para permitir que os servidores enviem mensagens aos clientes é surpreendentemente eficaz (embora possa ser cara no lado do servidor porque o servidor deve manter uma conexão ativa com todos os seus clientes). Por ser um padrão de programação útil, o JavaScript do lado do cliente oferece suporte a ele com a API EventSource. Para criar esse tipo de conexão de solicitação de longa duração com um servidor web, basta passar uma URL para o construtor EventSource(). Quando o servidor grava dados (formatados corretamente) na conexão, o objeto EventSource os converte em eventos que você pode escutar:

```
let ticker = new EventSource("stockprices.php");
ticker.addEventListener("bid", (evento) => {
  displayNewBid(event.data);}
```

O objeto de evento associado a um evento de mensagem tem uma propriedade de dados que contém qualquer cadeia de caracteres que o servidor enviou como carga útil para esse evento.

O objeto de evento também tem uma propriedade `type`, como todos os objetos de evento, que especifica o nome do evento. O servidor determina o tipo de eventos que são gerados. Se o servidor omitir um nome de evento nos dados que ele grava, o tipo de evento será padronizado como "mensagem".

O protocolo Server-Sent Event é simples. O cliente inicia uma conexão com o servidor (quando cria o objeto `EventSource`) e o servidor mantém essa conexão aberta. Quando ocorre um evento, o servidor grava linhas de texto na conexão. Um evento que passa por cima do fio pode ter esta aparência, se os comentários forem omitidos:

```
Evento: Licitação Define o tipo do objeto de evento  
dados: G00G Define a propriedade Data  
Dados: 999 anexa uma nova linha e mais dados// uma  
linha em branco marca o fim do evento
```

Existem alguns detalhes adicionais no protocolo que permitem que os eventos recebam IDs e permitem que um cliente de reconexão informe ao servidor qual foi o ID do último evento recebido, para que um servidor possa reenviar quaisquer eventos perdidos. Esses detalhes são invisíveis do lado do cliente, no entanto, e não são discutidos aqui.

Uma aplicação óbvia para eventos enviados pelo servidor é para colaborações multiusuário, como bate-papo online. Um cliente de chat pode usar `fetch()` para postar mensagens na sala de chat e assinar o fluxo de conversa com um objeto `EventSource`. O exemplo 15-11 demonstra como é fácil escrever um cliente de chat como este com `EventSource`.

Exemplo 15-11. Um cliente de bate-papo simples usando EventSource

```
<html><head><title>Bate-  
papo</title></head> <body>SSE
```

```
<!-- A interface do usuário do chat é apenas um único campo de entrada de texto --><!-- Novas mensagens de chat serão inseridas antes deste campo de entrada --><input id="input" style="width:100%; padding:10px; border:solidblack 2px"/><script> Cuide de alguns detalhes da interface do usuáriiolet nick = prompt("Digite seu apelido");// Obter o apelido do usuário input = document.getElementById("input"); // Encontre o inputfieldinput.focus()// Defina keyboardfocus
```

Registre-se para notificação de novas mensagens usando EventSourcelet

```
chat = new EventSource("/chat");
chat.addEventListener("chat", event => {// Quando uma mensagem de bate-papo chega
    let div = document.createElement("div"); Crie um <div>
    div.append(evento.data); Adicionar texto de
    a mensagem
    entrada.append(div); E adicione div
    Antes da entrada
    entrada.scrollIntoView(); Garantir a entrada
    ELT é
    visível});
```

Publique as mensagens do usuário no servidor usando fetchinput.addEventListener("change", ()=>{// Quando os ataques do usuário retornam

```
fetch("/chat", {// Inicie uma solicitação HTTP para esta url.
```

```
method: "POST",// Torná-lo um POSTrequest com corpo
        body: nick + ": " + input.value // definido para o
        entalhe e entrada.
    }).catch(e =>
        console.error); Ignore a resposta, mas
    registre todos os erros.
    input.value = "";
}); Limpar a entrada
</script>
</body>
</html>
```

O código do lado do servidor para este programa de bate-papo não é muito mais

complicado do que o código do lado do cliente. O exemplo 15-12 é um servidor NodeHTTP simples. Quando um cliente solicita a URL raiz "/", ele envia o código do chatclient mostrado no Exemplo 15-11. Quando um cliente faz uma solicitação GET para a URL "/chat", ele salva o objeto de resposta e mantém essa conexão aberta. E quando um cliente faz uma solicitação POST para "/chat", ele usa o corpo da solicitação como uma mensagem de bate-papo e a escreve, usando o formato "text/event-stream" para cada um dos objetos de resposta salvos. O código do servidor escuta na porta 8080, então, depois de executá-lo com o Node, aponte seu navegador para <http://localhost:8080> para se conectar e começar conversando consigo mesmo.

Exemplo 15-12. Um servidor de bate-papo de eventos enviados pelo servidor

Este é um JavaScript do lado do servidor, destinado a ser executado com o NodeJS.// Ele implementa uma sala de bate-papo muito simples e completamente anônima.// POSTE novas mensagens em /chat ou OBTENHA um fluxo de texto/evento de mensagens// da mesma URL. Fazer uma solicitação GET para / retorna um arquivo HTML simples// que contém a interface do usuário do bate-papo do lado do cliente.
const http = require("http"); const fs = require("fs"); const url = require("url");

O arquivo HTML do cliente de bate-papo. Usado abaixo.
const clientHTML = fs.readFileSync("chatClient.html");

Uma matriz de objetos ServerResponse que vamos enviar eventos para deixar clientes = [];

Crie um novo servidor e ouça na porta 8080.// Conecte-se ao <http://localhost:8080/> para usá-lo.
let server = new http. Servidor();
server.listen(8080);

Quando o servidor receber uma nova solicitação, execute esta função
server.on("request", (request, response) => {

```
Analise o URLlet pathname =
url.parse(request.url).pathname;
```

Se a solicitação for para "/", envie o chatUI do lado do cliente.

```
if (nome do caminho === "/") { Uma solicitação para a interface do usuário do chat
    response.writeHead(200, {"Tipo de conteúdo": "texto/html"}).end(clientHTML);
}// Caso contrário, envie um erro 404 para qualquer caminho diferente de "/chat" ou para

qualquer método diferente de "GET" e "POST" else if (pathname !== "/chat" ||
    (request.method !== "GET" && request.method !== "POST")) {
    resposta.writeHead(404).end();}Se a solicitação /chat for um GET, um cliente estará se conectando.
```

```
else if (request.method === "GET") {
    acceptNewClient(solicitação, resposta);}Caso contrário, a solicitação /chat é um POST de uma nova mensagem else {
    broadcastNewMessage(solicitação,
    resposta);}});
```

Isso lida com solicitações GET para o endpoint /chat que são geradas quando// o cliente cria um novo objeto EventSource (ou quando theEventSource// se reconecta automaticamente).function acceptNewClient(request, response) {

Lembre-se do objeto de resposta para que possamos enviar mensagens futuras para ele
 clientes.push(resposta);

Se o cliente fechar a conexão, remova o

```
response do array de
clientsRequest.connection.on("end", () => {
    clientes.splice(clientes.indexOf(resposta), 1);
```

```
resposta.end();});
```

Defina cabeçalhos e envie um evento de bate-papo inicial para apenas este cliente

```
resposta.writeHead(200, {  
  "Tipo de conteúdo": "texto/fluxo de  
  eventos", "Conexão": "keep-alive", "Cache-Control":  
  "sem cache"}); response.write("evento: chat\n"data:  
  Conectado\n\n");
```

Observe que intencionalmente não chamamos response.end() aqui.

Manter a conexão aberta é o que faz Server-SentEvents funcionar.}

Essa função é chamada em resposta a solicitações POST para o/ponto de extremidade de chat// que os clientes enviam quando os usuários digitam uma nova mensagem. função assíncrona broadcastNewMessage(request, response) { Primeiro, leia o corpo da solicitação para obter a mensagem do usuário

```
request.setEncoding("utf8"); seja  
corpo = ""; for await (let pedaço  
de solicitação) {  
  corpo += pedaço;}
```

Depois de ler o corpo, envie uma resposta vazia e feche a conexão

```
resposta.writeHead(200).end();
```

Formatar a mensagem em formato de texto/fluxo de eventos, prefixando cada

```
linha com "data: "let mensagem = "data: " +  
body.replace("\n", "\ndata: ");
```

*Dê aos dados da mensagem um prefixo que os defina como um evento de "chat"
e dê a ele um sufixo de nova linha dupla que marca o fim do evento.*

```
let evento = 'evento: chat\n${mensagem}\n\n';
```

```
Agora envie este evento para todos os clientes  
ouvintesclients.forEach(client =>  
  client.write(event));}
```

15.11.3 WebSockets

A API WebSocket é uma interface simples para um protocolo de rede complexo e poderoso. Os WebSockets permitem que o código JavaScript no navegador troque facilmente mensagens de texto e binárias com um servidor. Assim como acontece com os eventos enviados pelo servidor, o cliente deve estabelecer a conexão, mas uma vez que a conexão é estabelecida, o servidor pode enviar mensagens de forma assíncrona para o cliente. Ao contrário do SSE, as mensagens binárias são suportadas e as mensagens podem ser enviadas em ambas as direções, não apenas do servidor para o cliente.

O protocolo de rede que habilita o WebSockets é um tipo de extensão para HTTP. Embora a API WebSocket seja uma reminiscência de soquetes de rede de baixo nível, os pontos de extremidade de conexão não são identificados pelo endereço IP e pela porta. Em vez disso, quando você deseja se conectar a um serviço usando o protocolo WebSocket, especifique o serviço com uma URL, assim como faria para um serviço da Web. No entanto, as URLs do WebSocket começam com `wss://` instead de `https://`. (Os navegadores normalmente restringem WebSockets para funcionar apenas em páginas carregadas em `https://`connections seguras).

Para estabelecer uma conexão WebSocket, o navegador primeiro estabelece uma conexão HTTP e envia ao servidor um cabeçalho `Upgrade: websocket` solicitando que a conexão seja alternada do protocolo HTTP para o protocolo WebSocket. O que isso significa é que, para usar WebSockets em seu JavaScript do lado do cliente, você precisará ser

trabalhando com um servidor web que também fala o protocolo WebSocket, e você precisará ter o código do lado do servidor escrito para enviar e receber dados usando esse protocolo. Se o seu servidor estiver configurado dessa forma, esta seção explicará tudo o que você precisa saber para lidar com a extremidade do lado do cliente da conexão. Se o seu servidor não suportar o protocolo WebSocket, considere usar Eventos enviados pelo servidor (§15.11.2) em vez disso.

**CRIANDO, CONECTANDO E
DESCONECTANDO WEBSOCKET** Se você quiser se comunicar com um servidor habilitado para WebSocket, crie um objeto WebSocket, especificando a URL `wss://` que identifica o servidor e o serviço que você deseja usar:

```
let soquete = novo WebSocket("wss://example.com/stockticker");
```

Quando você cria um WebSocket, o processo de conexão é iniciado automaticamente. Mas um WebSocket recém-criado não será conectado quando for retornado pela primeira vez.

A propriedade `readyState` do soquete especifica em qual estado a conexão está. Essa propriedade pode ter os seguintes valores:

WebSocket.CONNECTING

Este WebSocket está se conectando.

WebSocket.OPEN

Este WebSocket está conectado e pronto para comunicação.

WebSocket.CLOSING

Essa conexão WebSocket está sendo fechada.

WebSocket.FECHADO

Este WebSocket foi fechado; nenhuma outra comunicação é possível. Esse estado também pode ocorrer quando a tentativa de conexão inicial falha.

Quando um WebSocket faz a transição do estado CONNECTING para o OPENstate, ele dispara um evento "open" e você pode escutar esse evento definindo a propriedade onopen do WebSocket ou chamando addEventListener() nesse objeto.

Se ocorrer um protocolo ou outro erro para uma conexão WebSocket, o objeto WebSocket disparará um evento de "erro". Você pode definir onerror para definir um manipulador ou, alternativamente, usar addEventListener().

Quando terminar de usar um WebSocket, você pode fechar a conexão chamando o método close() do objeto WebSocket. Quando um WebSocket muda para o estado CLOSED, ele dispara um evento "close" e você pode definir a propriedade onclose para escutar esse evento.

ENVIANDO MENSAGENS POR UM WEBSOCKET Para enviar uma mensagem para o servidor na outra extremidade de uma conexão WebSocket, basta invocar o método send() do objeto WebSocket. send() espera um único argumento de mensagem, que pode ser a string, Blob, ArrayBuffer, matriz tipada ou objeto DataView.

O método send() armazena em buffer a mensagem especificada a ser transmitida e retorna antes que a mensagem seja realmente enviada. O

`bufferedAmount` do objeto `WebSocket` especifica o número de bytes que são armazenados em buffer, mas ainda não foram enviados. (Surpreendentemente, os `WebSockets` não disparam nenhum evento quando esse valor atinge 0.)

RECEBENDO MENSAGENS DE UM WEBSOCKET Para receber mensagens de um servidor por meio de um `WebSocket`, registre um manipulador de eventos para eventos de "mensagem", definindo a propriedade `onmessage` do objeto `WebSocket` ou chamando `addEventListener()`. O objeto associado a um evento "message" é uma instância `MessageEvent` com uma propriedade de dados que contém a mensagem do servidor. Se o servidor enviou texto codificado em UTF-8, a `event.data` será uma cadeia de caracteres contendo esse texto.

Se o servidor enviar uma mensagem que consiste em dados binários em vez de texto, a propriedade `data` será (por padrão) um objeto `Blob` que representa esses dados. Se você preferir receber mensagens binárias como `ArrayBuffers` em vez de `Blobs`, defina a propriedade `binaryType` do objeto `WebSocket` como a cadeia de caracteres "arraybuffer".

Há várias APIs da Web que usam mensagens de alteração forex de objetos `MessageEvent`. Algumas dessas APIs usam o algoritmo de clone estruturado ([consulte "O algoritmo de clone estruturado"](#)) para permitir estruturas de dados complexas como a carga útil da mensagem. `WebSockets` não é uma dessas APIs: as mensagens trocadas por um `WebSocket` são uma única cadeia de caracteres Unicode ou uma única cadeia de caracteres de bytes (representada como um `Blob` ou um `ArrayBuffer`).

NEGOCIAÇÃO DE PROTOCOLO

O protocolo WebSocket permite a troca de texto e mensagens binárias, mas nada diz sobre a estrutura ou o significado dessas mensagens. Os aplicativos que usam WebSockets devem construir seu próprio protocolo de comunicação sobre esse mecanismo simples de troca de mensagens. O uso de URLs wss:// ajuda nisso: cada URL normalmente terá suas próprias regras de como as mensagens devem ser trocadas. Se você escrever código para conectar-se a `wss://example.com/stockticker`, provavelmente saberá que receberá mensagens sobre os preços das ações.

Os protocolos tendem a evoluir, no entanto. Se um protocolo hipotético de cotação de ações for atualizado, você poderá definir um novo URL e conectar-se ao serviço atualizado, pois `wss://example.com/stockticker/v2`. URL-based controle de versão nem sempre é suficiente. Com protocolos complexos que evoluíram ao longo do tempo, você pode acabar com servidores implantados que suportam várias versões do protocolo e clientes implantados que suportam um conjunto diferente de versões de protocolo.

Antecipando essa situação, o protocolo WebSocket e a API incluem um recurso de negociação de protocolo no nível do aplicativo. Quando você chama o construtor `WebSocket()`, a URL `wss://` é o primeiro argumento, mas você também pode passar uma matriz de strings como o segundo argumento. Se você fizer isso, estará especificando uma lista de protocolos de aplicativos que você sabe como manipular e pedindo ao servidor para escolher um. Durante o processo de conexão, o servidor escolherá um dos protocolos (ou falhará com um erro se não suportar nenhuma das opções do cliente). Depois que a conexão for estabelecida, a propriedade `protocol` do objeto `WebSocket` especifica qual versão de protocolo o servidor escolheu.

15.12 Armazenamento

Os aplicativos da Web podem usar APIs do navegador para armazenar dados localmente no computador do usuário. Esse armazenamento do lado do cliente serve para dar uma memória ao navegador da web. Os aplicativos da Web podem armazenar as preferências do usuário, por exemplo, ou até mesmo armazenar seu estado completo, para que possam retomar exatamente de onde você parou no final de sua última visita. O armazenamento do lado do cliente é segregado por origem, portanto, as páginas de um site não podem ler os dados armazenados por páginas de outro site. Mas duas páginas do mesmo site podem compartilhar armazenamento e usá-lo como um mecanismo de comunicação. A entrada de dados em um formulário em uma página pode ser exibida em uma tabela em outra página, por exemplo. Os aplicativos da Web podem escolher o tempo de vida dos dados que armazenam: os dados podem ser armazenados temporariamente para que sejam retidos apenas até que a janela feche ou o navegador saia, ou podem ser salvos no computador do usuário e armazenados permanentemente para que estejam disponíveis meses ou anos depois.

Há várias formas de armazenamento do lado do cliente:

Armazenamento na Web

A API de Armazenamento na Web consiste nos objetos `localStorage` e `sessionStorage`, que são essencialmente objetos persistentes que mapeiam chaves de cadeia de caracteres para valores de cadeia de caracteres. O armazenamento na Web é muito fácil de usar e é adequado para armazenar grandes (mas não enormes) quantidades de dados.

Bolinhos

Os cookies são um antigo mecanismo de armazenamento do lado do cliente que foi projetado para uso por scripts do lado do servidor. Uma API JavaScript estranha torna os cookies programáveis no lado do cliente, mas eles são difíceis de usar e adequados apenas para armazenar pequenas quantidades de dados textuais. Além disso, qualquer

os dados armazenados como cookies são sempre transmitidos ao servidor com cada solicitação HTTP, mesmo que os dados sejam apenas de interesse do cliente.

DB indexado

IndexedDB é uma API assíncrona para um banco de dados de objetos que oferece suporte à indexação.

ARMAZENAMENTO, SEGURANÇA E PRIVACIDADE

Os navegadores da Web geralmente se oferecem para lembrar as senhas da Web para você e as armazenam com segurança de forma criptografada no dispositivo. Mas nenhuma das formas de armazenamento de dados do lado do cliente descritas neste capítulo envolve criptografia: você deve presumir que tudo o que seus aplicativos da Web salvam reside no dispositivo do usuário de forma não criptografada. Os dados armazenados são, portanto, acessíveis a usuários curiosos que compartilham acesso ao dispositivo e a softwares maliciosos (como spyware) que existem no dispositivo. Por esse motivo, nenhuma forma de armazenamento do lado do cliente deve ser usada para senhas, números de contas financeiras ou outras informações confidenciais semelhantes.

15.12.1 localStorage e sessionStorage

As propriedades `localStorage` e `sessionStorage` do objeto `Window` referem-se a objetos `Storage`. Um objeto `Storage` se comporta como um objeto JavaScript normal, exceto que:

Os valores de propriedade de objetos `Storage` devem ser cadeias de caracteres. As propriedades armazenadas em um objeto `Storage` persistem. Se você definir uma propriedade do objeto `localStorage` e, em seguida, o usuário recarregar a página, o valor salvo nessa propriedade ainda estará disponível para o seu programa. Você pode usar o objeto `localStorage` assim, por exemplo:

```
let nome = localStorage.username;           Consultar um  
valor.  
if  
(!nome) {  
    nome = prompt("Qual é o seu nome?");  Peça ao usuário um
```

```
pergunta.
localStorage.username = nome;           Armazene o
resposta
.}
```

Você pode usar o operador delete para remover propriedades fromlocalStorage e sessionStorage, e você pode usar afor/in loop ou Object.keys() para enumerar as propriedades de um objeto Storage. Se você deseja remover todas as propriedades de um objeto de armazenamento, chame o método clear():

```
localStorage.clear();
```

Os objetos de armazenamento também definem os métodos getItem(), setItem() e deleteItem(), que você pode usar em vez do acesso directproperty e do operador delete, se desejar.

Lembre-se de que as propriedades dos objetos Storage só podem armazenar strings. Se você deseja armazenar e recuperar outros tipos de dados, terá que codificá-los e decodificá-los você mesmo.

Por exemplo:

Se você armazenar um número, ele será automaticamente convertido em uma string.// Não se esqueça de analisá-lo ao recuperá-lo de storage.localStorage.x = 10; let x = parseInt(localStorage.x);

Converta uma data em uma string ao definir e analise-a ao obter localStorage.lastRead = (new Date()).toUTCString(); let lastRead = new Date(Date.parse(localStorage.lastRead));

JSON faz uma codificação conveniente para qualquer primitivo ou dados

```
structurelocalStorage.data =  
JSON.stringify(dados);  
dados do storelet =  
JSON.parse(localStorage.data);  
decifrar.
```

*Codificar e
Recuperar e
decifrar.*

TEMPO DE VIDA E ESCOPO DO ARMAZENAMENTO A DIFERENÇA ENTRE LOCALStorage e sessionStorage envolve o tempo de vida e o escopo do armazenamento. Os dados armazenados por meio do armazenamento local são permanentes: eles não expiram e permanecem armazenados no dispositivo do usuário até que um aplicativo da web o exclua ou o usuário peça ao navegador (por meio de alguma interface do usuário específica do navegador) para excluí-lo.

localStorage tem como escopo a origem do documento. Conforme explicado em "A política de mesma origem", a origem de um documento é definida por seu protocolo, nome do host e porta. Todos os documentos com a mesma origem compartilham os mesmos dados localStorage (independentemente da origem dos scripts que realmente acessam localStorage). Eles podem ler os dados uns dos outros e podem substituir os dados uns dos outros. Mas documentos com origens diferentes nunca podem ler ou substituir os dados uns dos outros (mesmo que ambos estejam executando um script do mesmo servidor de terceiros).

Observe que localStorage também tem como escopo a implementação do navegador. Se você visitar um site usando o Firefox e depois visitar novamente usando o Chrome (por exemplo), todos os dados armazenados durante a primeira visita não estarão acessíveis durante a segunda visita.

Os dados armazenados por meio da sessão O armazenamento tem um tempo de vida diferente dos dados armazenados por meio do localStorage: ele tem o mesmo tempo de vida que a janela de nível superior ou a guia do navegador na qual o script que os armazenou está

Executando. Quando a janela ou guia é fechada permanentemente, todos os dados armazenados por meio de sessionStorage são excluídos. (Observe, no entanto, que os navegadores modernos têm a capacidade de reabrir guias fechadas recentemente e restaurar a última sessão de navegação, portanto, o tempo de vida dessas guias e sua sessão associadaArmazenamento pode ser mais longo do que parece.)

Como localStorage, sessionStorage tem como escopo a origem do documento para que documentos com origens diferentes nunca compartilhem sessionStorage. Mas sessionStorage também tem escopo por janela. Se um usuário tiver duas guias do navegador exibindo documentos da mesma origem, essas duas guias terão dados separadossessionStorage: os scripts em execução em uma guia não podem ler ou substituir os dados gravados por scripts na outra guia, mesmo que ambas as guias estejam visitando exatamente a mesma página e estejam executando exatamente os mesmos scripts.

EVENTOS DE ARMAZENAMENTOQuando os dados armazenados no localStorage são alterados, o navegador aciona um evento de "armazenamento" em qualquer outro objeto Window para o qual esses dados são visíveis (mas não na janela que fez a alteração). Se um navegador tiver duas guias abertas para páginas com a mesma origem e uma dessas páginas armazenar um valor em localStorage, a outra guia receberá um evento de "armazenamento".

Registre um manipulador para eventos de "armazenamento" definindo window.onstorage ou chamando window.addEventListener() com o tipo de evento "armazenamento".

O objeto de evento associado a um evento de "armazenamento" tem algumas propriedades importantes:

chave

O nome ou a chave do item que foi definido ou removido.
Se o método clear() foi chamado, essa propriedade será nula.

novoValor

Mantém o novo valor do item, se houver. Se removeItem() foi chamado, essa propriedade não estará presente.

oldValue

Contém o valor antigo de um item existente que foi alterado ou excluído. Se uma nova propriedade (sem valor antigo) for adicionada, essa propriedade não estará presente no objeto de evento.

área de armazenamento

O objeto Storage que foi alterado. Geralmente é o objeto localStorage.

URL

A URL (como uma cadeia de caracteres) do documento cujo script fez essa alteração de armazenamento.

Observe que localStorage e o evento "storage" podem servir como um mecanismo de transmissão pelo qual um navegador envia uma mensagem para todas as janelas que estão visitando o mesmo site no momento. Se um usuário solicitar que um site pare de executar animações, por exemplo, o site poderá armazenar essa preferência no localStorage para que possa honrá-la em visitas futuras. E ao armazenar a preferência, ele gera um evento que

permite que outras janelas que exibem o mesmo site também honrem a solicitação.

Como outro exemplo, imagine um aplicativo de edição de imagens baseado na web que permite ao usuário exibir paletas de ferramentas em janelas separadas. Quando o usuário seleciona uma ferramenta, o aplicativo usa localStorage para salvar o estado atual e gerar uma notificação para outras janelas de que uma nova ferramenta foi selecionada.

15.12.2 Biscoitos

Um cookie é uma pequena quantidade de dados nomeados armazenados pelo navegador da web e associados a uma determinada página da web ou site. Os cookies foram projetados para programação do lado do servidor e, no nível mais baixo, são implementados como uma extensão do protocolo HTTP. Os dados de cookies são transmitidos automaticamente entre o navegador da web e o servidor da web, portanto, os scripts do lado do servidor podem ler e gravar valores de cookies armazenados no cliente. Esta seção demonstra como os scripts do lado do cliente também podem manipular cookies usando a propriedade cookie do objeto Document.

POR QUE "COOKIE"?

O nome "cookie" não tem muito significado, mas não é usado sem precedentes. Nos anais da história da computação, o termo "cookie" ou "cookie mágico" tem sido usado para se referir a um pequeno pedaço de dados, particularmente um pedaço de dados privilegiados ou secretos, semelhante a uma senha, que prova que a identidade ou permite o acesso. Em JavaScript, os cookies são usados para salvar o estado e podem estabelecer uma espécie de identidade para um navegador da web. No entanto, os cookies em JavaScript não usam nenhum tipo de criptografia e não são seguros de forma alguma (embora transmiti-los por meio de uma conexão https: ajude).

A API para manipular cookies é antiga e enigmática. Não há métodos envolvidos: os cookies são consultados, definidos e excluídos lendo

e escrever a propriedade cookie do objeto Document usando cadeias de caracteres especialmente formatadas. O tempo de vida e o escopo de cada cookie podem ser especificados individualmente com atributos de cookie. Esses atributos também são especificados com cadeias de caracteres especialmente formatadas definidas na mesma cookiepropriedade.

As subseções a seguir explicam como consultar e definir valores e atributos de cookies.

LENDÔ COOKIES Quando você lê a propriedade `document.cookie`, ela retorna uma string que contém todos os cookies que se aplicam ao documento atual. A cadeia de caracteres é uma lista de pares de nome/valor separados entre si por ponto e vírgula e um espaço. O valor do cookie é apenas o valor em si e não inclui nenhum dos atributos que podem estar associados a esse cookie. (Falaremos sobre atributos a seguir.) Para usar a propriedade `document.cookie`, você normalmente deve chamar o método `split()` para dividi-lo em pares individuais de nome/valor.

Depois de extrair o valor de um cookie da `cookieproperty`, você deve interpretar esse valor com base em qualquer formato ou codificação usada pelo criador do cookie. Você pode, por exemplo, passar o valor do cookie para `decodeURIComponent()` e depois para `JSON.parse()`.

O código a seguir define uma função `getCookie()` que analisa a propriedade `document.cookie` e retorna um objeto cujas propriedades especificam os nomes e valores dos cookies do documento:

```

Retorna os cookies do documento como um objeto
Map.// Suponha que os valores dos cookies
estejam codificados
withencodeURIComponent().function getCookies() {
    let cookies = new Map(); O objeto que retornaremos
    let all = document.cookie; Pegue todos os biscoitos em um grande
corda
let list = all.split("; "); Dividir em pares de
nome/valor individual
    for(let cookie da lista) { Para cada cookie nesse
lista
        if (!cookie.includes("=")) continuar; Ignorar se houver
é no = sinal
        let p = cookie.indexOf("=");
Encontre o
primeiro = sinal
        let nome = cookie.substring(0, p); Obter biscoito
nome
        let valor = cookie.substring(p+1); Obter biscoito
valor
        valor = decodeURIComponent(valor); Decodifique o
valor
        cookies.set(nome, valor); Lembrar
Nome e valor do cookie
    }
    return cookies;
}

```

ATRIBUTOS DE COOKIE: TEMPO DE VIDA E ESCOPOEm adição a um nome e um valor, cada cookie tem atributos opcionais que controlam seu tempo de vida e escopo. Antes de podermos descrever como definir cookies com JavaScript, precisamos explicar os atributos do cookie.

Os cookies são transitórios por padrão; Os valores que eles armazenam duram a duração da sessão do navegador da Web, mas são perdidos quando o usuário sai do navegador. Se você quiser que um cookie dure além de uma única sessão de navegação, você deve informar ao navegador quanto tempo (em segundos) você gostaria que ele retivesse o cookie especificando um atributo max-age. Se você especificar um

vitalício, o navegador armazenará cookies em um arquivo e os excluirá somente quando expirarem.

A visibilidade do cookie é definida pela origem do documento, como localStorage e sessionStorage, mas também pelo caminho do documento. Esse escopo é configurável por meio de atributos de cookie, caminho e domínio. Por padrão, um cookie é associado e acessível à página da web que o criou e a quaisquer outras páginas da web no mesmo diretório ou em quaisquer subdiretórios desse diretório. Se a página da web example.com/catalog/index.html creates um cookie, por exemplo, esse cookie também estará visível toexample.com/catalog/order.html and example.com/catalog/widgets/index.html, mas não estará visível toexample.com/about.html.

Esse comportamento de visibilidade padrão geralmente é exatamente o que você deseja. Às vezes, porém, você desejará usar valores de cookie em todo o site, independentemente de qual página cria o cookie. Por exemplo, se o usuário inserir seu endereço de correspondência em um formulário em uma página, você pode querer salvar esse endereço para usar como padrão na próxima vez que retornar à página e também como padrão em um formulário totalmente não relacionado em outra página onde ele é solicitado a inserir um endereço de cobrança. Para permitir esse uso, especifique um caminho para o cookie. Em seguida, qualquer página da Web do mesmo servidor da Web cujo URL comece com o prefixo de caminho especificado pode compartilhar o cookie. Por exemplo, se um conjunto de cookies byexample.com/catalog/widgets/index.html tiver seu caminho definido como "/catalog", esse cookie também ficará visível para example.com/catalog/order.html. Ou, se thepath estiver definido como "/", o cookie ficará visível para qualquer página no example.com domínio, dando ao cookie um escopo semelhante ao de localStorage.

Por padrão, os cookies têm como escopo a origem do documento. No entanto, sites grandes podem querer que os cookies sejam compartilhados entre subdomínios. Por exemplo, o servidor em `order.example.com` pode precisar ler os valores de cookie definidos a partir de `catalog.example.com`. É aqui que entra o `domainattribute`. Se um cookie criado por um `oncatalog.example.com` de página definir seu atributo de caminho como "/" e seu atributo de domínio como ".example.com", esse cookie estará disponível para todas as páginas da Web em `catalog.example.com`, `orders.example.com` e qualquer outro servidor no domínio `example.com`. Observe que você não pode definir o domínio de um cookie para um domínio diferente de um domínio pai do seu servidor.

O atributo de cookie final é um atributo booleano chamado `secure` que especifica como os valores de cookie são transmitidos pela rede. Por padrão, os cookies são inseguros, o que significa que são transmitidos por uma conexão HTTP normal e insegura. Se um cookie for marcado como seguro, no entanto, ele será transmitido somente quando o navegador e o servidor estiverem conectados via HTTPS ou outro protocolo seguro.

LIMITAÇÕES DE COOKIES

Os cookies destinam-se ao armazenamento de pequenas quantidades de dados por scripts do lado do servidor, e esses dados são transferidos para o servidor sempre que um URL relevante é solicitado. O padrão que define cookies incentiva os fabricantes de navegadores a permitir um número ilimitado de cookies de tamanho irrestrito, mas não exige que os navegadores retenham mais de 300 cookies no total, 20 cookies por servidor web ou 4 KB de dados por cookie (tanto o nome quanto o valor contam para esse limite de 4 KB). Na prática, os navegadores permitem muito mais de 300 cookies no total, mas o limite de tamanho de 4 KB ainda pode ser aplicado por alguns.

ARMAZENANDO COOKIES Para associar um valor de cookie transitório ao documento atual, basta definir a propriedade `cookie` como uma string `name=value`. Por exemplo:

```
document.cookie  
='version=${encodeURIComponent(document.lastModified)}';
```

Na próxima vez que você ler a propriedade cookie, o par nome/valor armazenado será incluído na lista de cookies do documento. Os valores de cookie não podem incluir ponto e vírgula, vírgulas ou espaços em branco. Por esse motivo, você pode querer usar a função global principal do JavaScript encodeURIComponent() para codificar o valor antes de armazená-lo no cookie. Se você fizer isso, terá que usar a função correspondingdecodeURIComponent() ao ler o cookievalue.

Um cookie escrito com um simples par de nome/valor dura a sessão de navegação na web atual, mas é perdido quando o usuário sai do navegador. Para criar um cookie que possa durar entre as sessões do navegador, especifique seu tempo de vida (em segundos) com um atributo max-age. Você pode fazer isso definindo a propriedade cookie como uma string do formulário: name=value; max-age=segundos. A função a seguir define um cookie com um atributo opcional max-age:

Armazene o par nome/valor como um cookie, codificando o valuewith// encodeURIComponent() para escapar ponto e vírgula, vírgulas e espaços.// Se daysToLive for um número, defina o atributo max-age para que o cookie// expire após o número especificado de dias. Passe 0 paraexcluir um cookie.function setCookie(nome, valor, diasParaViver=nulo){

```
let cookie = '${nome}=${encodeURIComponent(valor)}';  
if (daysToLive !== null) {  
  cookie += '; max-  
  age=${daysToLive*60*60*24}';}  
document.cookie =  
cookie;
```

```
}
```

Da mesma forma, você pode definir os atributos de caminho e domínio de um cookie anexando strings do formato ;path=value ou ;domain=value para a string definida na propriedade document.cookie. Para definir a propriedade secure, basta anexar ;secure.

Para alterar o valor de um cookie, defina seu valor novamente usando o mesmo nome, caminho e domínio junto com o novo valor. Você pode alterar o tempo de vida de um cookie ao alterar seu valor especificando um atributo newmax-age.

Para excluir um cookie, defina-o novamente usando o mesmo nome, caminho e domínio, especificando um valor arbitrário (ou vazio) e um atributo max-age de 0.

15.12.3 DB indexado

A arquitetura de aplicativos da Web tradicionalmente apresenta HTML, CSS e JavaScript no cliente e um banco de dados no servidor. Você pode achar surpreendente, portanto, saber que a plataforma da web inclui um banco de dados simpleobject com uma API JavaScript para armazenar persistentemente objetos JavaScript no computador do usuário e recuperá-los conforme necessário.

O IndexedDB é um banco de dados de objetos, não um banco de dados relacional, e é muito mais simples do que os bancos de dados que suportam consultas SQL. No entanto, é mais poderoso, eficiente e robusto do que o armazenamento de chave/valor fornecido pelo localStorage. Como o localStorage,

Os bancos de dados IndexedDB têm como escopo a origem do containingdocument: duas páginas da Web com a mesma origem podem acessar os dados uma da outra, mas páginas da Web de origens diferentes não.

Cada origem pode ter qualquer número de bancos de dados IndexedDB. Cada um tem um nome que deve ser único dentro da origem. No IndexedDB API, um banco de dados é simplesmente uma coleção de armazenamentos de objetos nomeados. Como o nome indica, um armazenamento de objetos armazena objetos. Os objetos são serializados no armazenamento de objetos usando o algoritmo de clone estruturado (consulte "O Algoritmo de Clone Estruturado"), o que significa que os objetos armazenados podem ter propriedades cujos valores são Mapas, Conjuntos ou matrizes digitadas. Each deve ter uma chave pela qual possa ser classificado e recuperado do armazenamento. As chaves devem ser exclusivas - dois objetos no mesmo armazenamento podem não ter a mesma chave - e devem ter uma ordem natural para que possam ser classificadas. Strings JavaScript, números e objetos Date são chaves válidas. Um banco de dados IndexedDB pode gerar automaticamente uma chave exclusiva para cada objeto inserido no banco de dados. Muitas vezes, porém, os objetos que você insere em um armazenamento de objetos já terão uma propriedade que é adequado para uso como uma chave. Nesse caso, você especifica um "caminho de chave" para essa propriedade ao criar o armazenamento de objetos. Conceitualmente, um caminho de chave é um valor que informa ao banco de dados como extrair a chave de um objeto do objeto.

Além de recuperar objetos de um armazenamento de objetos por seu valor primarykey, talvez você queira ser capaz de pesquisar com base no valor de outras propriedades no objeto. Para poder fazer isso, é possível definir qualquer número de índices no armazenamento de objetos. (A capacidade de indexar um armazenamento de objetos explica o nome "IndexedDB".) Cada índice define uma chave secundária para os objetos armazenados. Esses índices geralmente não são

úniques e vários objetos podem corresponder a um único valor de chave.

O IndexedDB fornece garantias de atomicidade: consultas e atualizações para o banco de dados são agrupadas em uma transação para que todas sejam bem-sucedidas ou todas falhem juntas e nunca deixem o banco de dados em um estado indefinido e parcialmente atualizado. As transações no IndexedDB são mais simples do que em muitas APIs de banco de dados; vamos mencioná-los novamente mais tarde.

Conceitualmente, a API IndexedDB é bastante simples. Para consultar ou atualizar um banco de dados, primeiro abra o banco de dados desejado (especificando-o pelo nome). Em seguida, você cria um objeto de transação e usa esse objeto para procurar o armazenamento de objetos desejado no banco de dados, também por nome. Finalmente, você procura um objeto chamando o método `get()` do armazenamento de objetos ou armazena um novo objeto chamando `put()` (ou chamando `add()`, se quiser evitar sobreescriver objetos existentes).

Se você quiser procurar os objetos para um intervalo de chaves, crie um objeto `IDBRange` que especifique os limites superior e inferior do intervalo e passe-o para os métodos `getAll()` ou `openCursor()` do armazenamento de objetos.

Se você quiser fazer uma consulta usando uma chave secundária, procure o índice nomeado do armazenamento de objetos e, em seguida, chame os métodos `get()`, `getAll()` ou `openCursor()` do objeto de índice, passando uma chave única ou um objeto `IDBRange`.

Essa simplicidade conceitual da API IndexedDB é complicada, no entanto, pelo fato de que a API é assíncrona (para que os aplicativos da Web possam usá-la sem bloquear o thread principal da interface do usuário do navegador). DB indexado

foi definido antes que as promessas fossem amplamente suportadas, portanto, a API é baseada em eventos em vez de em promessas, o que significa que ela não funciona com `async` e `await`.

A criação de transações e a consulta de armazenamentos de objetos e índices são operações assíncronas. Mas abrir um banco de dados, atualizar um repositório de objetos e consultar um repositório ou índice são operações assíncronas. Todos esses métodos assíncronos retornam imediatamente um objeto de solicitação. O navegador aciona um evento de sucesso ou erro no objeto de solicitação quando a solicitação é bem-sucedida ou falha, e você pode definir manipuladores com as propriedades `onsuccess` e `onerror`. Dentro de um `onsuccesshandler`, o resultado da operação está disponível como a propriedade `result` do objeto de solicitação. Outro evento útil é o evento "complete" despachado em objetos de transação quando uma transação é concluída com êxito.

Um recurso conveniente dessa API assíncrona é que ela simplifica o gerenciamento de transações. A API IndexedDB força você a criar um objeto de transação para obter o armazenamento de objetos no qual você pode executar consultas e atualizações. Em uma API síncrona, você esperaria marcar explicitamente o final da transação chamando um método `commit()`. Mas com o IndexedDB, as transações são confirmadas automaticamente (se você não as anular explicitamente) quando todos os manipuladores de eventos `onsuccess` forem executados e não houver mais solicitações assíncronas pendentes que se refiram a essa transação.

Há mais um evento que é importante para a API IndexedDB. Quando você abre um banco de dados pela primeira vez ou quando incrementa o

número de versão de um banco de dados existente, o IndexedDB aciona um evento "upgradeneeded" no objeto de solicitação retornado pela chamada indexedDB.open(). O trabalho do manipulador de eventos para eventos "upgradeneeded" é definir ou atualizar o esquema para o novo banco de dados (ou a nova versão do banco de dados existente). Para bancos de dados IndexedDB, isso significa criar armazenamentos de objetos e definir índices nesses armazenamentos de objetos. E, de fato, a única vez que a API IndexedDB permite que você crie um armazenamento de objetos ou um índice é em resposta a um evento "upgradeneeded".

Com essa visão geral de alto nível do IndexedDB em mente, agora você deve ser capaz de entender o Exemplo 15-13. Esse exemplo usa IndexedDB para criar e consultar um banco de dados que mapeia códigos postais dos EUA (códigos postais) para cidades dos EUA. Ele demonstra muitos, mas não todos, os recursos básicos do IndexedDB. O exemplo 15-13 é longo, mas bem comentado.

Exemplo 15-13. Um banco de dados IndexedDB de códigos postais dos EUA

Esta função utilitária obtém de forma assíncrona o objeto de banco de dados (criando// e inicializando o banco de dados, se necessário) e o passa para o retorno de chamada.função withDB(retorno de chamada) {

```
let request = indexedDB.open("códigos postais", 1);
Solicitação vldo banco de dados
    request.onerror = console.error;  Registre todos os erros
    request.onsuccess = () => {  Ou chame isso quando terminar
        let db = request.result;  O resultado da solicitação
é o banco de dados
        retorno de chamada (db);  Invocar o retorno de chamada com
o banco de dados
    };
}
```

Se a versão 1 do banco de dados ainda não existir,
então este evento
manipulador será acionado. Isso é usado para criar e

*inicializar
armazenamentos e índices de objetos quando o banco de dados
é criado pela primeira vez ou para modificar
quando mudamos de uma versão do esquema de banco de dados para
outra.*

```
request.onupgradeneeded = () => {  
  initdb(request.result, callback); };
```

*withDB() chama esta função se o banco de dados ainda não tiver
sido inicializado.// Configuramos o banco de dados e o
preenchemos com dados, depois passamos o banco de dados para//
a função de retorno de chamada.//// Nosso banco de dados de
CEP inclui um armazenamento de objetos que contém objetos como
this://{//zipcode: "02134",//city: "Allston",//state:
"MA",//}//// Usamos a propriedade "zipcode" como a chave do
banco de dados e criamos um índice para// o nome da
cidade.function initdb(db, retorno de chamada) {*

*Crie o armazenamento de objetos, especificando um nome para o
armazenamento e*

*um objeto options que inclui o "caminho da
chave" especificando o*

```
nome da propriedade do campo-chave para esta loja.let  
store = db.createObjectStore("zipcodes", // nome da loja  
{ keyPath: "CEP" });
```

*Agora indexe o armazenamento de objetos pelo nome da
cidade, bem como por código postal.*

*Com este método, a string do caminho da chave é passada
diretamente como um
required em vez de como parte de um optionsObject.*

```
store.createIndex("cidades", "cidade");
```

*Agora obtenha os dados com os quais vamos inicializar o banco
de dados.*

O arquivo de dados zipcodes.json foi gerado a partir de dados licenciados pela CC de www.geonames.org:<https://download.geonames.org/export/zip/US.zip>

```
fetch("zipcodes.json")  
pedir  
    .then(resposta => response.json()) Fazer um HTTP GET  
    Analisar o corpo  
como JSON  
    .then(zipcodes => {// Obter CEP de 40K  
arquivo  
        Para inserir dados de CEP no  
banco de dados precisamos de um  
transação. Para criar nossa transação  
objeto, precisamos  
para especificar quais armazenamentos de objetos usaremos  
(só temos  
um) e precisamos dizer a ele que estaremos fazendo  
grava no  
banco de dados, não apenas lê:  
let transaction = db.transaction(["zipcodes"],  
"readwrite");  
transaction.onerror = console.error;  
  
Obter nosso armazenamento de objetos do transaction  
let store = transaction.objectStore("zipcodes");  
  
A melhor parte da API IndexedDB é que  
Armazenamentos de objetos  
são *realmente* simples. Veja como adicionamos (ou  
atualização) nossos registros:  
for(let registro de códigos postais) { store.put(record); }  
  
Quando a transação for concluída com êxito, o  
base de dados  
é inicializado e pronto para uso, para que possamos chamar  
o  
callback que foi originalmente passada para  
withDB()  
    transaction.oncomplete = () => { callback(db); };});  
}  
  
Dado um código postal, use a API IndexedDB para pesquisar de  
forma assíncrona a cidade// com esse código postal e passá-  
lo para o retorno de chamada especificado,
```

```
ou passe null if// nenhuma cidade
for encontrada.function
lookupCity(zip, callback) {
    withDB(db => {
        Crie um objeto de transação somente leitura para isso
consulta. O
        argumento é uma matriz de armazenamentos de objetos que
para precisaremos
usar. let transação = db.transaction(["códigos postais"]);

        Obtenha o armazenamento de objetos do transactionlet
zipcodes = transaction.objectStore("zipcodes");

        Agora solicite o objeto que corresponde ao especificado
CEP.
As linhas acima eram síncronas, mas esta é
Async.
        let solicitação = zipcodes.get(zip);
        request.onerror = console.error;      Erros de log
        request.onsuccess = () => {           Ou chame isso
            função no sucesso
                let registro =                      Esta é a consulta
            resultado  solicitação.resultado;
                if (record) { // Se encontramos uma correspondência, passe-a
            o retorno  para
            de chamada  callback(`${record.city}, ${record.state}`);} else
                { // Caso contrário, diga ao retorno de chamada que
            nós falhamos
            callback(null);}}};});}
```

Dado o nome de uma cidade, use a API IndexedDB para de forma
assíncrona// procure todos os registros de CEP de todas as
cidades (em qualquer estado) que possuam// esse nome
(diferencia maiúsculas de minúsculas).function
lookupZipcodes(city, callback) {

```
    withDB(db => {
        Como acima, criamos uma transação e obtemos o objeto
loja
```

```
let transação = db.transaction(["códigos postais"]);
let store = transaction.objectStore("códigos postais");

Desta vez, também obtemos o índice da cidade do objeto
loja
let índice = store.index("cidades");

Solicite todos os registros correspondentes no índice com o
especificado
nome da cidade, e quando os pegamos os passamos para o
Callback.
Se esperássemos mais resultados, poderíamos usar
openCursor() em vez disso.
let solicitação = index.getAll(cidade);
request.onerror = console.error; request.onsuccess =
() => { callback(request.result);
};

});}
});}
```

15.13 Threads de trabalho e mensagens

Um dos recursos fundamentais do JavaScript é que ele é single-threaded: um navegador nunca executará dois manipuladores de eventos ao mesmo tempo e nunca acionará um cronômetro enquanto um manipulador de eventos estiver em execução, por exemplo. Atualizações simultâneas no estado do aplicativo ou no documento simplesmente não são possíveis, e os programadores do lado do cliente não precisam pensar ou mesmo entender a programação simultânea. Um corolário é que as funções JavaScript do lado do cliente não devem ser executadas por muito tempo; caso contrário, eles amarrarão o loop de eventos e o navegador da web não responderá à entrada do usuário. Esta é a razão pela qual `fetch()` é uma função assíncrona, por exemplo.

Os navegadores da Web relaxam com muito cuidado o requisito de thread único com a classe `Worker`: as instâncias dessa classe representam threads que são executadas

simultaneamente com o thread principal e o loop de eventos. Os trabalhadores vivem em um ambiente de execução independente com um objeto global completamente independente e sem acesso aos objetos Window ou Document. Os trabalhadores podem se comunicar com o thread principal somente por meio de uma passagem de mensagem assíncrona. Isso significa que as modificações simultâneas do DOM permanecem impossíveis, mas também significa que você pode escrever funções de longa duração que não interrompem o loop de eventos e travam o navegador. Criar um novo trabalhador não é uma operação pesada como abrir uma nova janela do navegador, mas os trabalhadores também não são "fibras" de peso mosca, e não faz sentido criar novos trabalhadores para realizar operações triviais. Aplicativos da Web complexos podem achar útil criar dezenas de trabalhadores, mas é improvável que um aplicativo com centenas ou milhares de trabalhadores seja prático.

Os workers são úteis quando seu aplicativo precisa executar tarefas computacionalmente intensivas, como processamento de imagens. Usar um trabalhador move tarefas como essa para fora do thread principal para que o navegador não pare de responder. E os trabalhadores também oferecem a possibilidade de dividir o trabalho entre vários tópicos. Mas os trabalhadores também são úteis quando você precisa executar cálculos moderadamente intensivos frequentes. Suponha, por exemplo, que você esteja implementando um editor de código simples no navegador e queira incluir realce de sintaxe. Para obter o realce correto, você precisa analisar o código em cada pressionamento de tecla. Mas se você fizer isso no thread principal, é provável que o código de análise impedirá que os manipuladores de eventos que respondem aos pressionamentos de tecla do usuário sejam executados prontamente e a experiência de digitação do usuário será lenta.

Como acontece com qualquer API de threading, há duas partes na API de trabalho. O

o primeiro é o objeto Worker: é assim que um worker se parece do lado de fora, para o thread que o cria. O segundo é theWorkerGlobalScope: este é o objeto global para um novo trabalhador e é a aparência de um thread de trabalho, por dentro, para si mesmo.

As seções a seguir abordam Worker e WorkerGlobalScope e também explicam a API de passagem de mensagens que permite que os trabalhadores se comuniquem com o thread principal e entre si. A mesma API de comunicação é usada para trocar mensagens entre um documento e <iframe>os elementos contidos no documento, e isso também é abordado nas seções a seguir.

15.13.1 Objetos de trabalho

Para criar um novo worker, chame o construtor Worker(), passando uma URL que especifica o código JavaScript que o worker deve executar:

```
let triturador de dados = novo trabalhador("utils/cruncher.js");
```

Se você especificar uma URL relativa, ela será resolvida em relação à URL do documento que contém o script que chamou o construtor Worker(). Se você especificar uma URL absoluta, ela deverá ter a mesma origem (mesmo protocolo, host e porta) que contém o documento.

Depois de ter um objeto Worker, você pode enviar dados para ele withpostMessage(). O valor que você passar para postMessage() será copiado usando o algoritmo de clone estruturado (consulte "O Algoritmo de Clone Estruturado"), e a cópia resultante será entregue ao trabalhador por meio de um evento de mensagem:

```
dataCruncher.postMessage("/api/dados/para/crunch");
```

Aqui estamos apenas passando uma única mensagem de string, mas você também pode usar objetos, matrizes, matrizes digitadas, mapas, conjuntos e assim por diante. Você pode receber mensagens de um trabalhador ouvindo eventos de "mensagem" no objeto Worker:

```
dataCruncher.onmessage = function(e) {  
    let stats = e.data; A mensagem é a propriedade de dados do evento  
    console.log('Média: ${stats.mean}');}
```

Como todos os destinos de eventos, os objetos Worker definem os métodos standard `addEventListener()` e `removeEventListener()`, e você pode usá-los no lugar do `onmessage`.

Além de `postMessage()`, os objetos de trabalho têm apenas um outrométodo, `terminate()`, que força um thread de trabalho a parar de funcionar.

15.13.2 O Objeto Global nos Trabalhadores

Ao criar um novo worker com o construtor `Worker()`, você especifica a URL de um arquivo de código JavaScript. Esse código é executado em um ambiente de execução JavaScript novo e puro, isolado do script que criou o trabalhador. O objeto global para esse novo ambiente de execução é um objeto `WorkerGlobalScope`. Um `WorkerGlobalScope` é algo mais do que o objeto global JavaScript principal, mas menos do que um objeto `Window` completo do lado do cliente.

O objeto WorkerGlobalScope tem um método postMessage() e uma propriedade do manipulador de eventos onmessage que são exatamente como as do objeto Worker, mas funcionam na direção oposta: chamar postMessage() dentro de um worker gera um evento de mensagem fora do worker, e as mensagens enviadas de fora do worker são transformadas em eventos e entregues ao manipulador onmessage. Como theWorkerGlobalScope é o objeto global para um worker, postMessage() e onmessage parecem uma função global e uma variável global para o código do worker.

Se você passar um objeto como o segundo argumento para o construtor Worker() e se esse objeto tiver uma propriedade name, o valor dessa propriedade se tornará o valor da propriedade name no objeto global do worker. Um worker pode incluir esse nome em qualquer mensagem impressa com console.warn() ou console.error().

A função close() permite que um worker se encerre e tem efeito semelhante ao método terminate() de um objeto Worker.

Como WorkerGlobalScope é o objeto global para workers, ele tem todas as propriedades do objeto global JavaScript principal, como o JSONobject, a função isNaN() e o construtor Date(). Além disso, no entanto, WorkerGlobalScope também tem as seguintes propriedades do objeto Window do lado do cliente:

- self é uma referência ao próprio objeto global.
WorkerGlobalScope não é um objeto Window e não define uma propriedade window.

Os métodos de temporizador setTimeout(), clearTimeout(), setInterval() e clearInterval(). Uma propriedade de localização que descreve a URL que foi passada para o construtor Worker(). Essa propriedade refere-se a um objeto Location, assim como a propriedade location de uma Window. O objeto Location tem propriedades href, protocol, host, hostname, port, pathname, search e hash. No entanto, em um trabalhador, essas propriedades são somente leitura. Uma propriedade navigator que se refere a um objeto compropriedades como as do objeto Navigator de uma janela. O objeto Navigator de um trabalhador tem as propriedades appName, appVersion, platform, userAgent e onLine. Os métodos de destino de evento usuais addEventListener() e removeEventListener(). Por fim, o objeto WorkerGlobalScope inclui APIs JavaScript importantes do lado do cliente, incluindo o objeto Console, a função fetch() e a API IndexedDB. WorkerGlobalScope também inclui o construtor Worker(), o que significa que os threads de trabalho podem criar seus próprios trabalhadores.

15.13.3 Importando código para um trabalhador

Os workers foram definidos em navegadores da web antes que o JavaScript tivesse um module system, portanto, os workers têm um sistema exclusivo para incluir código adicional. WorkerGlobalScope define importScripts() como uma função global à qual todos os trabalhadores têm acesso:

Antes de começarmos a trabalhar, carregue as classes e utilitários precisamos importScripts("utils/Histogram.js", "utils/BitSet.js");

`importScripts()` recebe um ou mais argumentos de URL, cada um dos quais deve se referir a um arquivo de código JavaScript. As URLs relativas são resolvidas em relação à URL que foi passada para o construtor `Worker()` (não em relação ao documento que o contém). `importScripts()` carrega e executa esses arquivos de forma síncrona um após o outro, na ordem em que foram especificados. Se carregar um script causar um erro de rede ou se a execução lançar um erro de qualquer tipo, nenhum dos scripts subsequentes será carregado ou executado. Um script carregado com `importScripts()` pode chamar `importScripts()` para carregar os arquivos dos quais depende. Observe, no entanto, que `importScripts()` não tenta acompanhar quais scripts já foram carregados e não faz nada para evitar ciclos de dependência.

`importScripts()` é uma função síncrona: ela não retorna até que todos os scripts tenham sido carregados e executados. Você pode começar a usar os scripts carregados assim que `importScripts()` retornar: não há necessidade de um retorno de chamada, manipulador de eventos, método `then()` ou `await`. Depois de internalizar a natureza assíncrona do JavaScript do lado do cliente, é estranho voltar à programação simples e síncrona novamente. Mas essa é a beleza dos threads: você pode usar uma chamada de função de bloqueio em um worker sem bloquear o loop de eventos no thread principal e sem bloquear os cálculos que estão sendo executados simultaneamente em outros workers.

MÓDULOS EM WORKERS

Para usar módulos em workers, você deve passar um segundo argumento para o construtor `Worker()`. Esse segundo argumento deve ser um objeto com uma propriedade `type` definida como a cadeia de caracteres "`module`". Passar a opção `type="module"` para o construtor `Worker()` é muito parecido com usar o atributo `type="module"` em uma tag HTML `<script>`: significa que o código deve ser interpretado como um módulo e

que as declarações de importação são permitidas.

Quando um worker carrega um módulo em vez de um script tradicional, o WorkerGlobalScope não define a função importScripts().

Observe que, desde o início de 2020, o Chrome é o único navegador compatível com módulos verdadeiros e import declarations em workers.

15.13.4 Modelo de execução de trabalhador

Os threads de trabalho executam seu código (e todos os scripts ou módulos importados) de forma síncrona de cima para baixo e, em seguida, entram em uma fase assíncrona na qual respondem a eventos e temporizadores. Se um worker registrar um manipulador de eventos de "mensagem", ele nunca será encerrado enquanto houver a possibilidade de que os eventos de mensagem ainda cheguem. Mas se um trabalhador não escutar as mensagens, ele será executado até que não haja mais tarefas pendentes (como promessas e temporizadores fetch()) e todos os retornos de chamada relacionados à tarefa tenham sido chamados. Depois que todos os retornos de chamada registrados forem chamados, não há como um trabalhador iniciar uma nova tarefa, portanto, é seguro para o threadexit, o que será feito automaticamente. Um worker também pode se desligar explicitamente chamando a função global close(). Observe que não há propriedades ou métodos no objeto Worker que especifiquem se um thread de trabalho ainda está em execução ou não, portanto, os workers não devem se fechar sem coordenar isso de alguma forma com seu thread pai.

ERROS EM WORKERS Se uma exceção ocorrer em um worker e não for capturada por nenhuma catch cláusula, um evento de "erro" será acionado no objeto global do worker. Se esse evento for manipulado e o manipulador chamar o método preventDefault() do objeto de evento, a propagação de erros será encerrada. Caso contrário, o evento "error" será acionado no Worker

objeto. Se preventDefault() for chamado lá, propagationends. Caso contrário, uma mensagem de erro será impressa no console do desenvolvedor e o manipulador onerror (§15.1.7) do objeto Window será invocado.

```
Lidar com erros de trabalho não detectados com um
manipulador dentro de theworker.self.onerror =
function(e) {
  console.log('Erro no trabalhador
em${e.filename}:${e.lineno}: ${e.message}');
  e.preventDefault();};
```

```
Ou lide com erros de trabalho não detectados com um
manipulador fora do worker.worker.onerror = function(e) {

  console.log('Erro no trabalhador
em${e.filename}:${e.lineno}: ${e.message}');
  e.preventDefault();};
```

Assim como o Windows, os trabalhadores podem registrar um manipulador a ser invocado quando aPromise for rejeitado e não houver nenhuma função .catch() para lidar com isso. Dentro de um trabalhador, você pode detectar isso definindo a função aself.onunhandledrejection ou usandoaddEventListener() para registrar um manipulador global para eventos "unhandledrejection". O objeto de evento passado para este manipulador terá uma propriedade promise cujo valor é o objeto Promise thatrejected e uma propriedade reason cujo valor é o que teria sido passado para uma função .catch().

15.13.5 postMessage(), MessagePorts e MessageChannels

O método postMessage() do objeto Worker e a função globalpostMesage() definida dentro de um worker funcionam invocando os métodos postMessage() de um par de objetos MessagePort que são criados automaticamente junto com o worker. O JavaScript do lado do cliente não pode acessar diretamente esses objetos MessagePort criados automaticamente, mas pode criar novos pares de portas conectadas com o construtor MessageChannel():

```
let canal = novo MessageChannel;           Crie um  
new channel.let myPort = channel.port1;// Tem duas portas  
let yourPort = channel.port2;// conectadas uma à outra.
```

```
myPort.postMessage("Você pode me ouvir?");      Uma mensagem  
postada em um will yourPort.onmessage = (e) =>  
console.log(e.data); recebido por outro.
```

Um MessageChannel é um objeto com propriedades port1 e port2 que se referem a um par de objetos MessagePort conectados. Um MessagePort é um objeto com um método postMessage() e uma propriedade onmessage eventhandler. Quando postMessage() é chamado em uma porta de um par conectado, um evento "message" é disparado na outra porta do par. Você pode receber esses eventos de "mensagem" definindo a propriedade onmessage ou usando addEventListener() para registrar um ouvinte para eventos de "mensagem".

As mensagens enviadas para uma porta são enfileiradas até que a propriedade onmessage seja definida ou até que o método start() seja chamado na porta. Isso evita que as mensagens enviadas por uma extremidade do canal sejam perdidas

do outro lado. Se você usar addEventListener() com aMessagePort, não se esqueça de chamar start() ou talvez nunca veja uma mensagem entregue.

Todas as chamadas postMessage() que vimos até agora receberam um argumento singlemessage. Mas o método também aceita um secondargument opcional. Esse segundo argumento é uma matriz de itens que devem ser transferidos para a outra extremidade do canal, em vez de ter uma cópia enviada pelo canal. Os valores que podem ser transferidos em vez de copiados sãoMessagePorts e ArrayBuffer. (Alguns navegadores também implementam outros tipos transferíveis, como ImageBitmap e OffscreenCanvas. Estes não são universalmente suportados, no entanto, e não são abordados neste livro.) Se o primeiro argumento para postMessage() incluir aMessagePort (aninhado em qualquer lugar dentro do objeto de mensagem), então thatMessagePort também deve aparecer no segundo argumento. Se você fizer isso, o MessagePort ficará disponível para a outra extremidade do canal e imediatamente se tornará não funcional do seu lado. Suponha que você tenha criado um trabalhador e queira ter dois canais para se comunicar com ele: um canal para troca de dados comum e um canal para mensagens de alta prioridade. No thread principal, você pode criar um MessageChannel e, em seguida, chamar postMessage() no worker para passar uma das MessagePorts para ele:

```
let trabalhador = new Trabalhador("worker.js"); let  
urgentChannel = new MessageChannel(); let urgentPort  
= urgentChannel.port1; worker.postMessage({ comando:  
"setUrgentPort", value:urgentChannel.port2 },  
  
[ urgentChannel.port2 ]);  
Agora podemos receber mensagens urgentes do trabalhador  
como esta
```

```
urgentPort.addEventListener("mensagem",
handleUrgentMessage); urgentPort.start(); // Comece a
receber mensagens // E envie mensagens urgentes como
estaurgentPort.postMessage("teste");
```

MessageChannels também são úteis se você criar dois workers e quiser permitir que eles se comuniquem diretamente entre si, em vez de exigir código no thread principal para retransmitir mensagens entre eles.

O outro uso do segundo argumento para postMessage() é transferir ArrayBuffers entre trabalhadores sem ter que copiá-los. Este é um aprimoramento de desempenho importante para grandes ArrayBuffers como aqueles usados para armazenar dados de imagem. Quando um ArrayBuffer é transferido por um MessagePort, o ArrayBuffer se torna inutilizável no thread original para que não haja possibilidade de acesso simultâneo ao seu conteúdo. Se o primeiro argumento para postMessage() incluir um ArrayBuffer ou qualquer valor (como um array tipado) que tenha um ArrayBuffer, então thatbuffer pode aparecer como um elemento de array no argumento secondpostMessage(). Se aparecer, será transferido sem cópia. Caso contrário, o ArrayBuffer será copiado em vez de transferido. O exemplo 15-14 demonstrará o uso dessa técnica de transferência com ArrayBuffers.

15.13.6 Mensagens entre origens com postMessage()

Há outro caso de uso para o método postMessage() no JavaScript do lado do cliente. Envolve janelas em vez de trabalhadores, mas há semelhanças suficientes entre os dois casos que descreveremos o método postMessage() do objeto Window aqui.

Quando um documento contém um <iframe> elemento, esse elemento atua como uma janela incorporada, mas independente. O objeto Element querepresenta o <iframe> tem uma propriedade contentWindow que é o objeto Window para o documento incorporado. E para scripts em execução nesse iframe aninhado, a propriedade window.parent refere-se ao contendo objeto Window. Quando duas janelas exibem documentos com a mesma origem, os scripts em cada uma dessas janelas têm acesso ao conteúdo da outra janela. Mas quando os documentos têm origens diferentes, a política de mesma origem do navegador impede que o JavaScriptem uma janela acesse o conteúdo de outra janela.

Para os trabalhadores, postMessage() fornece uma maneira segura para dois threads independentes se comunicarem sem compartilhar memória. Para janelas, postMessage() fornece uma maneira controlada para duas origens independentes trocarem mensagens com segurança. Mesmo que a política de mesma origem impeça que seu script veja o conteúdo de outra janela, você ainda pode chamar postMessage() nessa janela e, ao fazer isso, um evento de "mensagem" será acionado nessa janela, onde poderá ser visto pelos manipuladores de eventos nos scripts dessa janela.

O método postMessage() de uma janela é um pouco diferente do método postMessage() de um trabalhador, no entanto. O primeiro argumento ainda é uma mensagem arbitrária que será copiada pelo algoritmo de clone estruturado. Mas o segundo argumento opcional listando objetos a serem transferidos em vez de copiados torna-se um terceiro argumento opcional. O método postMessage() de uma janela recebe uma string como seu segundo argumento necessário. Esse segundo argumento deve ser um origin (um protocolo, nome de host e porta opcional) que especifica quem você

Espere receber a mensagem. Se você passar a cadeia de caracteres "<https://good.example.com>" como o segundo argumento, mas a janela na qual você está postando a mensagem realmente contiver conteúdo de "<https://malware.example.com>", a mensagem que você postou não será entregue. Se você estiver disposto a enviar sua mensagem para conteúdo de qualquer origem, poderá passar o curinga "*" como o segundo argumento.

O código JavaScript em execução dentro de uma janela ou <iframe> pode receber mensagens postadas nessa janela ou quadro definindo a propriedade onmessagedessa janela ou chamando addEventListener() para eventos de "mensagem". Assim como acontece com os trabalhadores, quando você recebe um evento "message" para uma janela, a propriedade data do objeto de evento é a mensagem que foi enviada. Além disso, no entanto, os eventos de "mensagem" entregues às janelas também definem as propriedades de origem e origem. A propriedade source especifica o objeto Window que enviou o evento, e você pode usar event.source.postMessage() para enviar uma resposta. A propriedade origin especifica a origem do conteúdo na sourcewindow. Isso não é algo que o remetente da mensagem possa forjar e, quando você recebe um evento de "mensagem", normalmente deseja: verifique se é de uma origem esperada.

15.14 Exemplo: O Conjunto de Mandelbrot

Este capítulo sobre JavaScript do lado do cliente culmina com um longo exemplo que demonstra o uso de workers e mensagens para paralelizar tarefas computacionalmente intensivas. Mas ele foi escrito para ser um aplicativo da Web envolvente do mundo real e também demonstra várias outras APIs demonstradas neste capítulo, incluindo a história

gestão; uso da classe `ImageData` com um `<canvas>`; e o uso de eventos de teclado, ponteiro e redimensionamento. Ele também demonstra importantes recursos básicos do JavaScript, incluindo geradores e um uso sofisticado de promessas.

O exemplo é um programa para exibir e explorar o Mandelbrotset, um fractal complexo que inclui belas imagens como a mostrada na Figura 15-16.

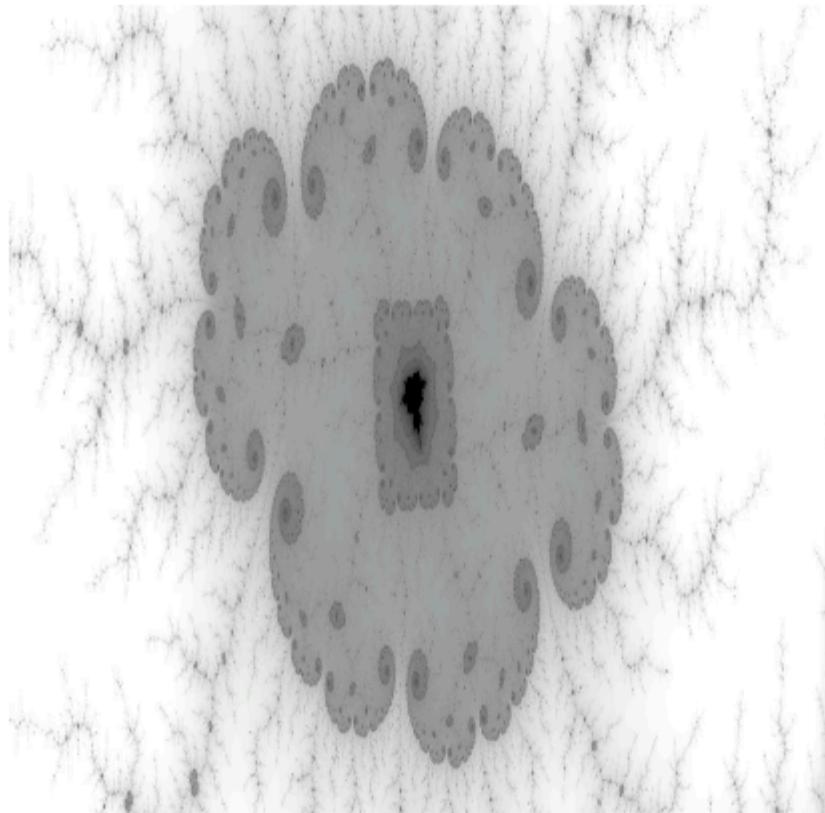


Figura 15-16. Uma parte do conjunto de Mandelbrot

O conjunto de Mandelbrot é definido como o conjunto de pontos no plano complexo, que, quando submetidos a um processo repetido de multiplicação complexa e adição, produzem um valor cuja magnitude permanece limitada. Os contornos do conjunto são surpreendentemente complexos e calcular quais pontos são membros do conjunto e quais não são é computacionalmente intensivo: para produzir uma imagem de 500×500 do conjunto de Mandelbrot, você deve calcular individualmente a associação de cada um dos 250.000 pixels em sua imagem. E para verificar se o valor associado a cada pixel permanece limitado, talvez seja necessário repetir o processo de multiplicação complexa 1.000 vezes ou mais. (As reiterações fornecem limites mais bem definidos para o conjunto; menos iterações produzem limites mais difusos.) Com até 250 milhões de passos de aritmética complexa necessários para produzir uma imagem de alta qualidade do conjunto de Mandelbrot, você pode entender por que o uso de trabalhadores é uma técnica valiosa. O exemplo 15-14 mostra o código de trabalho que usaremos. Este arquivo é relativamente compacto: é apenas o músculo computacional bruto para o programa maior. Duas coisas são dignas de nota sobre isso, no entanto:

- O trabalhador cria um objeto `ImageData` para representar a grade retangular de pixels para a qual está computando a associação do conjunto de Mandelbrot. Mas, em vez de armazenar valores reais de pixel no `ImageData`, ele usa uma matriz de tipo personalizado para tratar cada pixel como um inteiro de 32 bits. Ele armazena o número de iterações necessárias para cada pixel nesta matriz. Se a magnitude do número complexo calculado para cada pixel se tornar maior que quatro, então é matematicamente garantido que ele crescerá sem limites a partir de então, e dizemos que ele "escapou". Portanto, o valor que esse worker retorna para cada pixel é o número de iterações antes do valor escapado. Informamos ao trabalhador o número máximo de iterações que ele deve tentar para cada valor, e os pixels que atingem esse número máximo são considerados como

estar no conjunto. O worker transfere o ArrayBuffer associado ao ImageData de volta para o thread principal para que a memória associada a ele não precise ser copiada. Exemplo 15-14. Código de trabalho para regiões de computação do Mandelbrotset

Este é um trabalhador simples que recebe uma mensagem de seu thread-pai, // executa o cálculo descrito por essa mensagem e, em seguida, posta o // resultado desse cálculo de volta para o thread pai.

Primeiro, descompactamos a mensagem que received:// - tile é um objeto com propriedades de largura e altura. Ele especifica o

// - tamanho do retângulo de pixels para o qual seremos computação

Filiação ao conjunto de Mandelbrot.// - (x0, y0) é o ponto no plano complexo que corresponde ao

pixel superior esquerdo no bloco.// - perPixel é o tamanho do pixel nas dimensões real e imaginária.

// - maxIterations especifica o número máximo de iterações que vamos executar antes de decidir que um pixel está no set.

```
const {tile, x0, y0, perPixel, maxIterations} = message.data;
```

```
const {largura, altura} = bloco;
```

Em seguida, criamos um objeto ImageData para representar uma matriz retangular de pixels, obtenha seu ArrayBuffer interno e crie uma visualização de matriz digitada desse buffer para que possamos tratar cada pixel como um único inteiro em vez de quatro bytes individuais. Armazenaremos o número de iterações para cada pixel nesta matriz de iterações. (As iterações serão transformadas em

```
cores de pixel reais no thread pai.)
```

```
const imageData = new ImageData(largura, altura);
```

```
const iterations = new Uint32Array(imageData.data.buffer);
```

Agora começamos o cálculo. Existem três forloops aninhados aqui.

Os dois exteriores percorrem as linhas e colunas de pixels, e o interior

loop itera cada pixel para ver se ele "escapa" ou não. Os vários

As variáveis de loop são as following:// - linha e coluna são inteiros que representam a coordenada pixel.

- x e y representam o ponto complexo para cada pixel: $x + yi$.

- index é o índice na matriz de iterações para o pixel atual.

- n rastreia o número de iterações para cada pixel.// - max e min rastreiam o maior e o menor número de iterações

vimos até agora para qualquer pixel no índice

```
rectangle.let = 0, max = 0, min = maxIterations; for(let  
linha = 0, y = y0; linha < altura; linha++, y +=perPixel)  
{
```

```
    for(let coluna = 0, x = x0; largura do < da coluna; coluna++, x  
+= por pixel) {
```

Para cada pixel, começamos com o número complexo
 $c = x+yi$.

Em seguida, calculamos repetidamente o número complexo
 $z(n+1)$ com base em

este recursivo formula:// $z(0) = c/z(n+1) =$
 $z(n)^2 + c$ // Se $|z(n)|$ (a magnitude de $z(n)$) é >
2, então

o

pixel não faz parte do conjunto e paramos depois de n
Iterações.

seja n; O número de iterações para
longe

seja r = x, i = y; Comece com $z(0)$ definido como c
for(n = 0; n < maxIterations; n++) {
 seja rr = r*r, ii = i*i; Eleve as duas partes ao quadrado
de $z(n)$.

if (RR + II > 4) { Se $|z(n)|^2$ é > 4
então

quebrar; nós escapamos e
pode parar de iterar.

```

        }i = 2*r*i +
        y;                                Calcular imaginário
parte de z(n+1).
        r = rr - ii + x;      E a parte real de
z(n+1).
        }iterações[índice++] =
        n;                                Lembrar #
iterações para cada pixel.
        se (n > max) max = n;      Acompanhe o máximo
número que vimos.
        if (n < min) min = n;// E o mínimo como
poço.
    }
}

```

Quando o cálculo estiver concluído, envie os resultados de volta para o pai fio. O objeto imageData será copiado, mas o ArrayBuffer gigante ele contém será transferido para um bom aumento de desempenho.

```
postMessage({tile, imageData, min, max},
[imageData.data.buffer]);};
```

O aplicativo visualizador de conjunto de Mandelbrot que usa esse código de trabalho é mostrado no Exemplo 15-15. Agora que você quase chegou ao final deste capítulo, este longo exemplo é uma espécie de experiência fundamental que reúne vários recursos e APIs importantes do JavaScript do lado do cliente e do lado do cliente. O código é comentado minuciosamente e encorajo você a lê-lo com atenção.

Exemplo 15-15. Um aplicativo da web para exibir e explorar o Conjunto de Mandelbrot

```
/*
* Esta classe representa um sub-retângulo de uma tela ou
imagem. Usamos Tiles para
* dividir uma tela em regiões que podem ser
processadas independentemente pelos Workers.
*/
```



```

/*
 * Esta classe representa um pool de trabalhadores, todos
executando o mesmo código. O
 * O código de trabalho especificado deve responder a cada
mensagem recebida
 * realizar algum tipo de computação e, em seguida,
postar uma única mensagem com
 * o resultado desse cálculo.** Dado um WorkerPool e uma
mensagem que representa o trabalho a ser executado,
simplesmente

* chame addWork(), com a mensagem como um argumento. Se houver
um trabalhador
* objeto que está ocioso no momento, a mensagem será postada
nesse trabalhador
*imediatamente. Se não houver objetos de trabalho
ociosos, a mensagem será* enfileirada e será postada em
um trabalhador quando um estiver disponível.

** addWork() retorna uma promessa, que será resolvida com
a mensagem recebida

* do trabalho, ou rejeitará se o trabalhador lançar um
erro não tratado.
*/
class
WorkerPool {
    construtor(numWorkers, workerSource) {
        this.idleWorkers = [];      Trabalhadores que não são
atualmente trabalhando
        this.workQueue = [];       Trabalho não está atualmente
em processamento
        this.workerMap = new Map(); Mapear trabalhadores para resolver
e rejeitar funcs

        Crie o número especificado de trabalhadores, adicione a mensagem
e erro
        manipuladores e salvá-los no array
        idleWorkers.for(let i = 0; i < numWorkers; i++) {
            let trabalhador = new Worker(workerSource);
            worker.onmessage = mensagem => {
                this._workerDone(trabalhador, nulo,
                mensagem.data);}; worker.onerror = erro => {

                this._workerDone(trabalhador, erro,
                nulo);};
    }
}

```

```
this.idleWorkers[i] = trabalhador;}}
```

Esse método interno é chamado quando um trabalhador termina de trabalhar, enviando uma mensagem ou lançando um error._workerDone(worker, error, response) { Procure as funções resolve() e reject() para este trabalhador e, em seguida, remova a entrada do trabalhador do map.let [resolver, rejeitador] = this.workerMap.get(worker); this.workerMap.delete(trabalhador); Se não houver nenhum trabalho na fila, coloque esse trabalhador de volta na lista de trabalhadores ociosos. Caso contrário, pegue o trabalho de a fila e enviá-lo para este worker.if (this.workQueue.length === 0) { this.idleWorkers.push(trabalhador);} else { deixe [trabalho, resolvedor, rejeitador] = this.workQueue.shift(); this.workerMap.set(trabalhador, [resolvedor, rejeitador]); worker.postMessage(trabalho); Por fim, resolva ou rejeite a promessa associada com o trabalhador. erro === null ? resolver(resposta) : rejeitador(erro);}

Esse método adiciona trabalho ao pool de trabalho e retorna aPromise que será resolvido com a resposta de um trabalhador quando o trabalho for concluído. O trabalho é um valor a ser passado para um worker com postMessage(). Se houver um trabalhador ocioso, a mensagem de trabalho será enviada imediatamente. Caso contrário, será enfileirado até que um trabalhador esteja disponível.addWork(work) { return new Promise((resolve, reject) => {

```

        if (this.idleWorkers.length > 0) {
            let trabalhador = this.idleWorkers.pop();
            this.workerMap.set(trabalhador, [resolver,
                rejeitar]); worker.postMessage(trabalho);} else {
            this.workQueue.push([trabalho, resolve, rejeitar]);}}}
    }

    /*
    * Esta classe contém as informações de estado necessárias para
    renderizar um conjunto de Mandelbrot.
    * As propriedades cx e cy dão o ponto no plano complexo que é
    o
    * centro da imagem. A propriedade perPixel especifica o
    quanto o real e o
    * Partes imaginárias desse número complexo mudam para cada
    pixel da imagem.* A propriedade maxIterations especifica o
    quanto trabalhamos para calcular o conjunto.

    * Números maiores requerem mais computação, mas produzem
    imagens mais nítidas.
    * Observe que o tamanho da tela não faz parte do estado.
    Dado cx, cy e
    * perPixel, simplesmente renderizamos qualquer parte do
    Mandelbrotset que se encaixe
    * a tela em seu tamanho atual.** Objetos deste tipo são
    usados com history.pushState() e são usados para ler

    * o estado desejado de uma URL marcada ou
    compartilhada.*/
    class PageState {

        Este método de fábrica retorna um estado inicial para
        exibir todo o conjunto.
        estático initialState() {
            let s = new PageState(); s.cx =
            -0,5; s.cy = 0; s.perPixel =
            3/window.innerHeight;
            s.maxIterations = 500; retorno s;
    }

```

```
}
```

Esse método de fábrica obtém o estado de uma URL ou retorna null se

*um estado válido não pôde ser lido a partir do URL.static fromURL(url) {
 let s = new PageState(); let u = new URL(url);
 Inicializar o estado a partir do
parâmetros de pesquisa do url.
 s.cx = parseFloat(u.searchParams.get("cx")); s.cy =
 parseFloat(u.searchParams.get("cy")); s.perPixel =
 parseFloat(u.searchParams.get("pp")); s.maxIterations
 = parseInt(u.searchParams.get("it")); // Se obtivermos
 valores válidos, retornaremos o objeto PageState,
caso contrário, nulo.
 return (isNaN(s.cx) || isNaN(s.cy) || isNaN(s.perPixel)
 || isNaN(s.maxIterations))?
 nulo: s;
}*

Este método de instância codifica o estado atual na pesquisa.

*parâmetros do location.toURL() atual do
navegador {
let u = new URL(window.location);
uSearchParams.set("cx", this.cx);
uSearchParams.set("cy", this.cy);
uSearchParams.set("pp", this.perPixel);
uSearchParams.set("it", this.maxIterations);
retornar u.href;}}*

*Essas constantes controlam o paralelismo da computação de
Mandelbrotset. // Pode ser necessário ajustá-las para obter
o melhor desempenho em seu computador. const ROWS = 3, COLS
= 4, NUMWORKERS = navigator.hardwareConcurrency || 2;*

*Esta é a aula principal do nosso programa de conjunto Mandelbrot.
Simplesmente*

Chame a função construtora // com o <canvas> elemento a ser renderinto. O programa// pressupõe que este <canvas> elemento é estilizado de modo que seja sempre tão grande// quanto a janela do navegador.

```
class MandelbrotCanvas {  
  
    construtor (tela) {  
        Armazene a tela, obtenha seu objeto de contexto e inicializar um WorkerPool  
        this.canvas = tela; this.context =  
        canvas.getContext("2d"); this.workerPool =  
        new WorkerPool(NUMWORKERS,  
"mandelbrotWorker.js");  
  
        Defina algumas propriedades que usaremos  
        laterthis.tiles = null;// Sub-regiões da tela  
        telathis.pendingRender = null;// Não estamos atualmente Renderização  
        this.wantsRerender = false; Nenhuma renderização está atualmente solicitado  
        this.resizeTimer = null; Impede-nos de redimensionar com muita frequênciathis.colorTable = null; Para converter dados brutos aos valores de pixel.  
  
        Configure nossos manipuladores de eventos  
        this.canvas.addEventListener("pointerdown", e =>  
        this.handlePointer(e));  
        window.addEventListener("keydown", e =>  
        this.handleKey(e));  
        window.addEventListener("redimensionar", e =>  
        this.handleResize(e));  
        window.addEventListener("popstate", e =>  
        this.setState(e.state, false));  
  
        Inic平ize nosso estado a partir da URL ou comece com o estado inicial.  
        this.state =  
        PageState.fromURL(window.location) ||  
        PageState.initState();  
  
        Salve esse estado com o mecanismo de histórico.  
    }  
}
```

```
        history.replaceState(this.state, "",  
this.state.toURL());  
  
        Defina o tamanho da tela e obtenha uma matriz de blocos que  
cubra-o.  
        this.setSize();  
  
        E renderize o conjunto de Mandelbrot no  
canvas.this.render();}
```

Defina o tamanho da tela e inicialize uma matriz de Tileobjects. Este é chamado a partir do construtor e também por thehandleResize() quando a janela do navegador é redimensionada.

```
setSize() {  
    this.width = this.canvas.width = window.innerWidth;  
    this.height = this.canvas.height = window.innerHeight;  
    this.tiles = [... Tile.tiles(this.width, this.height,  
LINHAS, COLS)];  
}
```

Essa função faz uma alteração no PageState e, em seguida, renderiza novamente o Mandelbrot usando esse novo estado e também salva o novo estado com history.pushState(). Se o primeiro argumento for uma função, essa função será chamado com o objeto de estado como seu argumento e deve fazer mudanças no estado. Se o primeiro argumento for um objeto, então simplesmente Copie as propriedades desse objeto para o StateObject. Se o opcional O segundo argumento for falso, o novo estado não será salvo. (Nós faça isso ao chamar setState em resposta a um popstateevent.)

```
setState(f, save=true) {  
    Se o argumento for uma função, chame-o para atualizar o  
estado.  
    Caso contrário, copie suas propriedades para o diretório  
estado.  
    if (typeof f === "função") {
```

```
f(this.state);}  
else {  
    for(let propriedade em f) {  
        this.state[propriedade] =  
            f[propriedade];}}}
```

Em ambos os casos, comece a renderizar o novo estado o mais rápido possível. `this.render();`

Normalmente salvamos o novo estado. Exceto quando estamos chamado com

um segundo argumento de falso que fazemos quando obtemos um evento popstate.

```
if (salvar) {  
    history.pushState(this.state, "",  
this.state.toURL());  
}}
```

Este método desenha de forma assíncrona a parte do conjunto de Mandelbrot especificado pelo objeto PageState na tela. É chamado por

o construtor, por setState() quando o estado muda e pelo

Redimensionar manipulador de eventos quando o tamanho da tela for alterado.

```
render() {  
    Às vezes, o usuário pode usar o teclado ou o mouse para  
    renderizações de solicitação  
    mais rapidamente do que podemos realizá-los. Nós não queremos  
    para enviar todos os  
    as renderizações para o pool de trabalho. Em vez disso, se estivermos  
    renderização, vamos  
    apenas anote que uma nova renderização é necessária e  
    quando a corrente  
    render for concluído, renderizaremos o estado atual,  
    possivelmente pulando  
    vários estados intermediários.  
    if (this.pendingRender) {  
        Se já estamos  
        Renderização  
        this.wantsRerender = true;      anote para  
        renderizar novamente mais tarde
```

```
        retornar;  
qualquer coisa mais agora.  
}
```

e não faça

Obtenha nossas variáveis de estado e calcule o complexo número para o canto superior esquerdo da tela.let {cx, cy, perPixel, maxIterations} = this.state; let x0 = cx - perPixel * this.width/2; let y0 = cy - perPixel * this.height/2;

Para cada um de nossos blocos ROWS*COLS, chame addWork() com uma mensagem para o código em mandelbrotWorker.js. Colete o Promessa resultante objetos em um array.let promises = this.tiles.map(tile =>
this.workerPool.addWork({
 tile: tile,x0: x0 + tile.x *
 perPixel,y0: y0 + tile.y *
 perPixel,perPixel:
 perPixel,maxIterations:
 maxIterations});

Use Promise.all() para obter uma matriz de respostas de a matriz de Promessas. Cada resposta é o cálculo de um de nossos azulejos.

Lembre-se de mandelbrotWorker.js que cada resposta inclui o

Tile, um objeto ImageData que inclui contagens de iteração em vez de valores de pixel, e o mínimo e o máximo Iterações

para esse
tile.this.pendingRender =

Promise.all(promises).then(respostas => {

Primeiro, encontre as iterações máximas e mínimas gerais sobre todos os ladrilhos.

Precisamos desses números para que possamos atribuir cores a os pixels.

```
    let min = maxIterations, max = 0;
```

```
for(let r de respostas) {  
    if (r.min < min) min = r.min;if  
    (r.max > max) max = r.max;}
```

Agora precisamos de uma maneira de converter a iteração bruta contagens do trabalhadores em cores de pixel que serão exibidas na tela.

Sabemos que todos os pixels têm entre min e Máximo de iterações

Portanto, pré-calculamos as cores para cada iteração Conte e armazene na matriz colorTable.

Se ainda não alocamos uma tabela de cores ou se não é mais

*o tamanho certo, então aloque um novo.if
(!this.colorTable || this.colorTable.length !==
maxIterations+1){
 this.colorTable = novo
Uint32Array(maxIterations+1);
}*

Dado o máximo e o mínimo, calcule o apropriado valores no tabela de cores. Os pixels do conjunto serão coloridos totalmente opaco

preto. Os pixels fora do conjunto serão preto translúcido com maior

contagens de iteração resultando em maior opacidade.

Pixels com contagens mínimas de iteração serão transparentes e 0 branco

plano de fundo será exibido, resultando em um imagem em tons de cinza.

*if (min === max) { Se todos os pixels são os mesmos,
 if (min === maxIterations) { Em seguida, faça-os todo preto
 this.colorTable[min] = 0xFF000000;}
 else {// Ou todos transparente.
 this.colorTable[min] = 0;*

```
    }} else
    {
        No caso normal em que min e max são
diferente, use um
            escala logarítmica para atribuir cada
iteração conta um
                opacidade entre 0 e 255 e, em seguida, use o
Deslocamento para a esquerda
                    operador para transformar isso em um valor
                    de pixel.let maxlog = Math.log(1+max-min);
                    for(let i = min; i <= max; i++) {
                        this.colorTable[i] =
                            (Math.ceil(Math.log(1+i-min)/maxlog *
255) << 24);
                    }
    }
```

Agora traduza os números de iteração em cada resposta

```
ImageData para cores do colorTable.let
r de respostas) {
    let iterações = novo
Uint32Array(r.imageData.data.buffer);
    for(let i = 0; i < iterations.length; i++) {
        iterações[i] =
this.colorTable[iterações[i]];
    }
}
```

Por fim, renderize todos os objetos imageData em seu

blocos correspondentes da tela usando putImageData().

(Primeiro, porém, remova todas as transformações CSS na tela que pode

```
    foram definidas pelo manipulador de eventos
    pointerdown.)this.canvas.style.transform = "";
    for(let r de respostas) {
        this.context.putImageData(r.imageData,
r.tile.x, r.tile.y);
    }
}
```

```
.catch((razão) => {
    Se algo desse errado em qualquer uma de nossas promessas,
    nós vamos registrar
        um erro aqui. Isso não deveria acontecer, mas isso
    vai ajudar com
        depuração se isso acontecer.console.error("Promessa
        rejeitada em render():",
    razão);
}).finally(() => {
    Quando terminarmos de renderizar, limpe o
    sinalizadores pendingRender
    this.pendingRender = null;// E se as solicitações
    de renderização chegassem enquanto estávamos
    ocupado, renderize novamente agora.
    if (this.wantsRerender) {
        this.wantsRerender = false;
        this.render();}});}
```

Se o usuário redimensionar a janela, essa função será chamada repetidamente.

Redimensionar uma tela e rerenderizar o conjunto de Mandelbrot é caro
operação que não podemos fazer várias vezes por segundo,
então usamos um temporizador
para adiar o tratamento do redimensionamento até que 200ms
tenham decorrido desde o último

```
resize foi
recebido.handleResize(event) {
    Se já estivermos adiando um redimensionamento, limpe-
    o.if (this.resizeTimer)
        clearTimeout(this.resizeTimer);// E adie esse
        redimensionamento.this.resizeTimer = setTimeout(() => {
            this.resizeTimer = null; Observe que o redimensionamento tem
            foi manuseado
                this.setSize();           Redimensionar tela e
            Telhas
                this.render();          Renderizar novamente no novo
            tamanho
            }, 200);
```

```
}
```

Se o usuário pressionar uma tecla, esse manipulador de eventos serchamado.

Chamamos setState() em resposta a várias chaves e setState() renderiza o novo estado, atualiza o URL e salva o estado no histórico do navegador.

```
    handleKey(evento) {
        switch(event.key) {case "Escape":// Dige Escape
            para voltar ao
        estado inicial
            this.setState(PageState.initialState()); quebrar;
        case "+":// Dige + para aumentar o número de
```

Iterações

```
        this.setState(s => {
            s.maxIterations =
        Math.round(s.maxIterations*1.5);
        });
        quebrar;
        caso "-":           Tipo - para diminuir o número de

```

Iterações

```
        this.setState(s => {
            s.maxIterations =
        Math.round(s.maxIterations/1.5);
        if (s.maxIterations < 1) s.maxIterations = 1;});
        quebrar; case "o":// Dige o para diminuir o zoom
```

```
        this.setState(s => s.perPixel *= 2); quebrar;
        case "ArrowUp":// Seta para cima para rolar
            para cima
        this.setState(s => s.cy -=
            this.height/10 *
        s.perPixel);
        quebrar; caso
        "Seta para baixo": Seta para baixo para rolar para baixo
            this.setState(s => s.cy += this.height/10 *
        s.perPixel);
        quebrar; caso
        "SetaEsquerda":   Seta para a esquerda para rolar para a esquerda
            this.setState(s => s.cx -= this.width/10 *
```

```
s.perPixel);
    quebrar; case "ArrowRight": // Seta para a
    direita para rolar para a direita
        this.setState(s) => s.cx += this.width/10 *
s.perPixel);
    quebra;}}
```

Esse método é chamado quando obtemos um evento pointerdown na tela.

O evento pointerdown pode ser o início de um gesto de zoom (um clique e// toque) ou um gesto de movimento panorâmico (um arrastar). Este manipulador registragerencia para

os eventos pointermove e pointerup para responder ao resto do gesto. (Esses dois manipuladores extras são removidos quando o gesto

*termina com um ponteiro para cima.)handlePointer(evento) {
As coordenadas de pixel e a hora da inicial ponteiro para baixo.*

Como a tela é tão grande quanto a janela, esses coordenadas do evento

*também são coordenadas de tela. const x0 =
event.clientX, y0 = event.clientY, t0 =
Data.agora();*

*Este é o manipulador para mover events.const
pointerMoveHandler = event => {
Quanto nos movemos e quanto tempo*

Passado?

```
let dx=event.clientX-x0, dy=event.clientY-y0,
dt=Data.agora()-t0;
```

Se o ponteiro se moveu o suficiente ou o suficiente já passou por isso

este não é um clique regular, então use CSS para deslocar a tela.

(Vamos renderizá-lo novamente de verdade quando obtivermos o pointerup.)

```
if (dx > 10 || dy > 10 || dt > 500) {
    this.canvas.style.transform =
```

```
'translate(${dx}px, ${dy}px)';  
};
```

Este é o manipulador para eventos
pointerupconst pointerUpHandler = event => {
Quando o ponteiro sobe, o gesto termina,
então remova
os manipuladores move e up até o próximo
gesture.this.canvas.removeEventListener("pointermove",
pointerMoveHandler);
this.canvas.removeEventListener("pointerup",
pointerUpHandler);

Passado? *Quanto o ponteiro se moveu e quanto tempo*

dt=Data.agora()-t0;
Descompacte o objeto de estado em
Constantes.
const {cx, cy, perPixel} = this.state;

Se o ponteiro se moveu o suficiente ou se o suficiente
o tempo passou, então

Este foi um gesto panorâmico, e precisamos mudar
estado a ser alterado
o ponto central. Caso contrário, o usuário clicou ou
tocou em um
ponto e precisamos centralizar e ampliar isso
ponto.

if (dx > 10 || dy > 10 || dt > 500) {
O usuário fez uma panorâmica da imagem por (dx, dy)
Pixels.

Converta esses valores em deslocamentos no
plano complexo.

*this.setState({cx: cx - dx*perPixel, cy: cy -*
*dy*perPixel});*

} else {
O usuário clicou. Calcule quantos pixels
o centro se move.

let cdx = x0 - this.width/2;
let cdy = y0 - this.height/2;

Use CSS para aumentar o zoom de forma rápida e temporária

```
this.canvas.style.transform =
    'translate(${-cdx*2}px, ${-cdy*2}px)
scala(2));
```

Defina as coordenadas complexas do novo ponto central e amplie por um fator de 2.

```
s.cx += cdx * s.perPixel; s.cy +=
cdy * s.perPixel; s.perPixel /=
2;});}};
```

Quando o usuário inicia um gesto, registramos manipuladores pelo pointermove e pointerup que segue.

```
this.canvas.addEventListener("pointermove",
pointerMoveHandler);
this.canvas.addEventListener("pointerup",
pointerUpHandler);
}}
```

Por fim, veja como configuramos a tela. Observe que este arquivo JavaScript// é autossuficiente. O arquivo HTML só precisa incluir <script>este.let canvas = document.createElement("canvas"); // Crie um elemento canvasdocument.body.append(canvas); // Insira-o no bodydocument.body.style = "margin:0"; // Sem margem para <body>canvas.style.width = "100%"; // Torne o canvas tão largo quanto bodycanvas.style.height = "100%"; // e tão alto quanto o body.new MandelbrotCanvas(canvas); // E comece a renderizar nele!

15.15 Resumo e Sugestões para Leitura Adicional

Este longo capítulo abordou os fundamentos da programação JavaScript do lado do cliente:

- Como os scripts e módulos JavaScript são incluídos nas páginas da web e quando eles são executados. Modelo de programação assíncrono e orientado a eventos do JavaScript do lado do cliente. O Document Object Model (DOM) que permite que o código JavaScript inspecione e modifique o conteúdo HTML do documento em que está incorporado. Essa API DOM é o coração de toda a programação JavaScript do lado do cliente. Como o código
- JavaScript pode manipular os estilos CSS que são aplicados ao conteúdo do documento. Como o código JavaScript pode
- obter as coordenadas dos elementos do documento na janela do navegador e dentro do próprio documento. Como criar "Componentes Web" de interface do usuário reutilizáveis
- com JavaScript, HTML e CSS usando os elementos personalizados e as APIs Shadow DOM. Como exibir e gerar gráficos dinamicamente com SVG e o <canvas> elemento
- HTML. Como adicionar efeitos sonoros com script (gravados e sintetizados) às suas páginas da web. Como o JavaScript
- pode fazer o navegador carregar novas páginas, retroceder e avançar no histórico de navegação do usuário e até mesmo adicionar novas entradas ao histórico de navegação.

Como os programas JavaScript podem trocar dados com servidores web usando os protocolos HTTP e WebSocket. Como os programas JavaScript podem armazenar dados no navegador do usuário. Como os programas JavaScript podem usar threads de trabalho para obter uma forma segura de simultaneidade. Este foi o capítulo mais longo do livro, de longe. Mas não pode chegar perto de cobrir todas as APIs disponíveis para navegadores da web. A plataforma web está se espalhando e em constante evolução, e meu objetivo para este capítulo foi apresentar as APIs principais mais importantes. Com o conhecimento que você tem deste livro, você está bem equipado para aprender e usar novas APIs conforme necessário. Mas você não pode aprender sobre uma nova API se não souber que ela existe, portanto, as seções curtas a seguir terminam o capítulo com uma lista rápida de recursos da plataforma da Web que você pode querer investigar no futuro.

15.15.1 HTML e CSS

A web é construída sobre três tecnologias principais: HTML, CSS e JavaScript, e o conhecimento de JavaScript pode levá-lo apenas até certo ponto como desenvolvedor web, a menos que você também desenvolva sua experiência com HTML e CSS. É importante saber como usar JavaScript para manipular elementos HTML e estilos CSS, mas esse conhecimento é muito mais útil se você também souber quais elementos HTML e quais estilos CSS usar.

Portanto, antes de começar a explorar mais APIs JavaScript, encorajo você a investir algum tempo no domínio das outras ferramentas do kit de ferramentas de um desenvolvedor web. Os elementos HTML form e input, por exemplo, têm um comportamento sofisticado que é importante entender, e o flexbox

e os modos de layout de grade em CSS são incrivelmente poderosos.

Dois tópicos aos quais vale a pena prestar atenção especial nesta área são acessibilidade (incluindo atributos ARIA) e internacionalização (incluindo suporte para direções de escrita da direita para a esquerda).

15.15.2 Desempenho

Depois de escrever um aplicativo da web e lançá-lo para o mundo, a busca sem fim para torná-lo rápido começa. No entanto, é difícil otimizar coisas que você não pode medir, por isso vale a pena se familiarizar com as APIs de desempenho. A propriedade performance do objeto window é o principal ponto de entrada para essa API. Ele inclui uma fonte de tempo de alta resolução `performance.now()` e métodos `performance.mark()` e `performance.measure()` para marcar pontos críticos em seu código e medir o tempo decorrido entre eles. Chamar esses métodos cria objetos `PerformanceEntry` que você pode acessar com `performance.getEntries()`. Os navegadores adicionam seus próprios objetos `PerformanceEntry` sempre que o navegador carrega uma nova página ou busca um arquivo pela rede, e esses objetos `PerformanceEntry` criados automaticamente incluem detalhes granulares do desempenho de rede do aplicativo. A classe `relatedPerformanceObserver` permite que você especifique uma função a ser invocada quando novos objetos `PerformanceEntry` forem criados.

15.15.3 Segurança

Este capítulo introduziu a ideia geral de como se defender contra vulnerabilidades de segurança de script entre sites (XSS) em seus sites, mas

Não entramos em muitos detalhes. O tópico de segurança na web é importante, e você pode querer passar algum tempo aprendendo mais sobre ele. Além do XSS, vale a pena aprender sobre o cabeçalho HTTP Content-Security-Policy e entender como o CSP permite que você peça ao navegador da Web para restringir os recursos que ele concede ao código JavaScript. Compreender o CORS (Cross-Origin Resource Sharing) também é importante.

15.15.4 WebAssembly

WebAssembly (ou "wasm") é um bytecode de máquina virtual de baixo nível formato projetado para se integrar bem com interpretadores JavaScript em navegadores da web. Existem compiladores que permitem compilar programas C, C++ e Rust para o bytecode WebAssembly e executá-los nos navegadores da Web em velocidade próxima à nativa, sem quebrar a sandbox do navegador ou o modelo de segurança. O WebAssembly pode exportar funções que podem ser chamadas por programas JavaScript. Um caso de uso típico para WebAssembly seria compilar a biblioteca zlibcompression padrão da linguagem C para que o código JavaScript tenha acesso a algoritmos de compactação e descompactação de alta velocidade. Saiba mais em <https://webassembly.org>.

15.15.5 Mais recursos de documento e janela

Os objetos Window e Document têm vários recursos que não foram abordados neste capítulo:

- O objeto Window define os métodos alert(), confirm() e prompt() que exibem diálogos modais simples para o usuário. Esses métodos bloqueiam o thread principal. O

`confirm()` retorna de forma síncrona um valor booleano e `prompt()` retorna de forma síncrona uma string de entrada do usuário. Estes não são adequados para uso em produção, mas podem ser úteis para projetos e protótipos simples. As

- propriedades `navigator` e `screen` do objeto `Window` foram mencionadas de passagem no início deste capítulo, mas os objetos `Navigator` e `Screen` aos quais eles fazem referência têm alguns recursos que não foram descritos aqui e que podem ser úteis. O método `requestFullscreen()` de qualquer objeto `Element` solicita que esse elemento (um `<video>` elemento ou `<canvas>`, por exemplo) seja exibido no modo de tela cheia. O método `exitFullscreen()` do documento retorna ao modo de exibição normal. O método `requestAnimationFrame()` do objeto `Window` usa uma função como argumento e executará essa função quando o navegador estiver se preparando para renderizar o próximo quadro.
- Quando você está fazendo alterações visuais (especialmente as repetidas ou animadas), envolver seu código em uma chamada `requestAnimationFrame()` pode ajudar a garantir que as alterações sejam renderizadas sem problemas e de uma maneira otimizada pelo navegador. Se o usuário selecionar texto em seu documento, você poderá obter detalhes dessa seleção com o método `Window.getSelection()` e obter o texto selecionado com `getSelection().toString()`. Em alguns navegadores, `navigator.clipboard` é um objeto com uma API assíncrona para ler e definir o conteúdo da área de transferência do sistema para permitir interações de copiar e colar com aplicativos fora do navegador. Um recurso pouco conhecido dos navegadores da web é que o HTML
-

- Elementos com um atributo `contenteditable="true"` permitem que seu conteúdo seja editado. O método `document.execCommand()` permite recursos de edição de rich textediting para conteúdo editável. Um `MutationObserver` permite que o JavaScript monitore as alterações em ou abaixo de um elemento especificado no documento. Crie um `MutationObserver` com o construtor `MutationObserver()`, passando a função de retorno de chamada que deve ser chamada quando as alterações são feitas. Em seguida, chame o método `observe()` do `MutationObserver` para especificar quais partes de qual elemento devem ser monitoradas. Um `IntersectionObserver` permite que o JavaScript determine quais elementos do documento estão na tela e quais estão próximos de estar na tela. É particularmente útil para aplicativos que desejam carregar dinamicamente o conteúdo sob demanda à medida que o usuário rola.

15.15.6 Eventos

O grande número e diversidade de eventos suportados pela plataforma web podem ser assustadores. Este capítulo discutiu uma variedade de tipos de eventos, mas aqui estão mais alguns que podem ser úteis:

- Os navegadores disparam eventos "online" e "offline" no objeto `Window` quando o navegador ganha ou perde uma conexão com a Internet.
- Os navegadores disparam um evento "visibilitychange" no objeto `Document` quando um documento se torna visível ou invisível (geralmente porque um usuário trocou de guia). O JavaScript pode usar `document.visibilityState` para determinar se o documento está atualmente "visível" ou "oculto". Os navegadores suportam uma API complicada para suportar arrastar e soltar.

- UIs e para oferecer suporte à troca de dados com aplicativos fora do navegador. Essa API envolve vários eventos, incluindo "dragstart", "dragover", "dragend" e "drop". Esta API é difícil de usar corretamente, mas útil quando você precisa dela. É uma API importante saber se você deseja permitir que os usuários arraste arquivos de sua área de trabalho para seu aplicativo da web. A API de bloqueio de ponteiro permite que o JavaScript oculte o ponteiro do mouse e obtenha eventos brutos do mouse como quantidades de movimento relativas, em vez de posições absolutas na tela. Isso é normalmente útil para jogos. Chame requestPointerLock() no elemento para o qual você deseja que todos os eventos do mouse sejam direcionados. Depois de fazer isso, os eventos "mousemove" entregues a esse elemento terão as propriedades movementX e movementY. A API do Gamepad adiciona suporte para controladores de jogos. Use navigator.getGamepads() para obter objetos Gamepad conectados e ouça eventos "gamepadconnected" no objeto Window para ser notificado quando um novo controlador estiver conectado. O objeto Gamepad define uma API para consultá-lo estado atual dos botões no controlador.

15.15.7 Aplicativos Web progressivos e service workers

O termo Progressive Web Apps, ou PWAs, é uma palavra da moda que descreve aplicativos da Web criados usando algumas tecnologias-chave. A documentação cuidadosa dessas tecnologias-chave exigiria um livro próprio, e eu não as abordei neste capítulo, mas você deve estar ciente de todas essas APIs. Vale a pena notar que APIs modernas poderosas como essas são normalmente projetadas para funcionar apenas em conexões HTTPS seguras. Os sites que ainda estão usando URLs http:// não poderão tirar proveito destes:

- Um ServiceWorker é um tipo de thread de trabalho com a capacidade de interceptar, inspecionar e responder a solicitações de rede do aplicativo da web que ele "atende". Quando um aplicativo da Web registra um service worker, o código desse worker se torna persistente no armazenamento local do navegador e, quando o usuário visita o site associado novamente, o service worker é reativado. Os service workers podem armazenar em cache as respostas da rede (incluindo arquivos de código JavaScript), o que significa que os aplicativos da Web que usam service workers podem se instalar efetivamente no computador do usuário para inicialização rápida e uso offline. O Service Worker Cookbook [at`https://serviceworker.rs`](https://serviceworker.rs) é um recurso valioso para aprender sobre os service workers e suas tecnologias relacionadas. A API de cache foi projetada para uso por service workers (mas também está disponível para código JavaScript regular fora dos workers). Ele funciona com os objetos Request e Response definidos pela API `fetch()` e implementa um cache de pares Request/Response. A API de Cache permite que um service worker armazene em cache os scripts e outros ativos do aplicativo Web que ele atende e também pode ajudar a habilitar o uso offline do aplicativo Web (o que é particularmente importante para dispositivos móveis). Um manifesto da Web é um arquivo formatado em JSON que descreve um aplicativo da Web, incluindo um nome, uma URL e links para ícones em vários tamanhos. Se o seu aplicativo Web usa um service worker e inclui uma tag `<link rel="manifest">` que faz referência a um arquivo `.webmanifest`, os navegadores (especialmente navegadores em dispositivos móveis) podem oferecer a opção de adicionar um ícone para o aplicativo Web à sua área de trabalho ou tela inicial. A API de notificações permite que os aplicativos da web exibam notificações usando o sistema de notificação nativo do sistema operacional em dispositivos móveis e desktop. As notificações podem incluir uma imagem e texto, e seu código pode receber um evento se o usuário clicar no botão

- notificação. O uso dessa API é complicado pelo fato de que você deve primeiro solicitar a permissão do usuário para exibir notificações. A API Push permite que aplicativos Web que têm um serviceworker (e que têm a permissão do usuário) assinem notificações de um servidor e exibam essas notificações quando o aplicativo em si não estiver em execução. As notificações push são comuns em dispositivos móveis, e o PushAPI aproxima os aplicativos da web da paridade de recursos com aplicativos nativos no celular.

15.15.8 APIs de dispositivos móveis

Há várias APIs da Web que são úteis principalmente para aplicativos da Web em execução em dispositivos móveis. (Infelizmente, várias dessas APIs funcionam apenas em dispositivos Android e não em dispositivos iOS.)

- A API de localização geográfica permite que o JavaScript (com a permissão do usuário) determine a localização física do usuário. É bem suportado em computadores e dispositivos móveis, incluindo dispositivos iOS. Use `navigator.geolocation.getCurrentPosition()` para solicitar a posição atual do usuário e `navigator.geolocation.watchPosition()` para registrar um retorno de chamada a ser chamado quando a posição do usuário for alterada. O método `navigator.vibrate()` faz com que um dispositivo móvel (mas não iOS) vibre. Muitas vezes, isso só é permitido em resposta a um gesto do usuário, mas chamar esse método permitirá que seu aplicativo forneça comentários silenciosos de que um gesto foi reconhecido. A API `ScreenOrientation` permite que um aplicativo Web consulte
-

- a orientação atual da tela de um dispositivo móvel e também parabloquear-se na orientação paisagem ou retrato. Os eventos "devicemotion" e "deviceorientation" no objeto window, relatam dados do acelerômetro e do magnetômetro para o dispositivo, permitindo determinar como o dispositivo está acelerando e como o usuário o está orientando no espaço. (Esses eventos funcionam no iOS.) A API do sensor ainda não é amplamente suportada além do Chrome em dispositivos
- Android, mas permite o acesso JavaScript ao conjunto completo de sensores de dispositivos móveis, incluindo acelerômetro, giroscópio, magnetômetro e sensor de luz ambiente. Esses sensores permitem que o JavaScript determine para qual direção um usuário está voltado ou para detectar quando o usuário agita o telefone, por exemplo.

15.15.9 APIs binárias

Arrays tipados, ArrayBuffer e a classe DataView (todos abordados no §11.2) permitem que o JavaScript funcione com dados binários. Conforme descrito anteriormente neste capítulo, a API fetch() permite que os programas JavaScript carreguem dados binários pela rede. Outra fonte de dados binários são os arquivos do sistema de arquivos local do usuário. Por razões de segurança, o JavaScript não pode apenas ler arquivos locais. Mas se o usuário selecionar um arquivo para upload (usando um elemento de formulário <input type="file">) ou usar arrastar e soltar para soltar um arquivo em seu aplicativo da Web, o JavaScript poderá acessar esse arquivo como um objeto File.

File é uma subclasse de Blob e, como tal, é uma representação opaca de um pedaço de dados. Você pode usar uma classe FileReader para obter de forma assíncrona o conteúdo de um arquivo como um ArrayBuffer ou string. (Em alguns navegadores,

você pode ignorar o FileReader e, em vez disso, usar os métodos Promise-basedtext() e arrayBuffer() definidos pela classe Blob ou o método stream() para transmitir o acesso ao conteúdo do arquivo.)

Ao trabalhar com dados binários, especialmente streaming de dados binários, pode ser necessário decodificar bytes em texto ou codificar texto como bytes. As classes TheTextEncoder e TextDecoder ajudam nessa tarefa.

15.15.10 APIs de mídia

A função navigator.mediaDevices.getUserMedia() permite que o JavaScript solicite acesso ao microfone e/ou câmera de vídeo do usuário. Uma solicitação bem-sucedida resulta em um objeto MediaStream. Os fluxos de vídeo podem ser exibidos em uma <video> tag (definindo a propriedade thesrcObject como o fluxo). Os quadros estáticos do vídeo podem ser capturados em uma tela externa <canvas> com a função canvasdrawImage(), resultando em uma fotografia de resolução relativamente baixa. Os fluxos de áudio e vídeo retornados por getUserMedia() podem ser gravados e codificados em um Blob com um objeto MediaRecorder.

A API WebRTC mais complexa permite a transmissão e recepção de MediaStreams pela rede, permitindo videoconferência ponto a ponto, por exemplo.

15.15.11 Criptografia e APIs relacionadas

A propriedade crypto do objeto Window expõe o método agerandomValues() para números pseudoaleatórios criptograficamente seguros. Outros métodos de criptografia, descriptografia, chave

geração, assinaturas digitais e assim por diante estão disponíveis por meio de `crypto.subtle`. O nome dessa propriedade é um aviso para todos que usam esses métodos de que o uso adequado de algoritmos criptográficos é difícil e que você não deve usar esses métodos a menos que realmente saiba o que está fazendo. Além disso, os métodos `decrypto.subtle` estão disponíveis apenas para código JavaScript em execução dentro de documentos que foram carregados por uma conexão HTTPS segura.

A API de gerenciamento de credenciais e a API de autenticação da Web permitem que o JavaScript gere, armazene e recupere credenciais de chave pública (e outros tipos de) e permite a criação de contas e o login sem senhas. A API JavaScript consiste principalmente nas funções `navigator.credentials.create()` e `navigator.credentials.get()`, mas é necessária uma infraestrutura substancial no lado do servidor para que esses métodos funcionem. Essas APIs ainda não são universalmente suportadas, mas têm o potencial de revolucionar a maneira como fazemos login em sites.

A API de solicitação de pagamento adiciona suporte ao navegador para fazer pagamentos com cartão de crédito na web. Ele permite que os usuários armazenem seus detalhes de pagamento com segurança no navegador para que não precisem digitar o número do cartão de crédito toda vez que fizerem uma compra. Os aplicativos Web que desejam solicitar um pagamento criam um objeto `PaymentRequest` e chamam seu método `show()` para exibir a solicitação ao usuário.

1 As edições anteriores deste livro tinham uma extensa seção de referência cobrindo a biblioteca padrão JavaScript e as APIs da web. Foi removido na sétima edição porque o MDN tem tornado-o obsoleto: hoje, é mais rápido procurar algo no MDN do que folhear um livro, e meus ex-colegas do MDN fazem um trabalho melhor em manter sua documentação online atualizada do que este livro jamais poderia.

2 Algumas fontes, incluindo a especificação HTML, fazem uma distinção técnica entre manipuladores e ouvintes, com base na maneira como são registrados. Neste livro, nós trate os dois termos como sinônimos.

3 Se você usou a estrutura React para criar interfaces de usuário do lado do cliente, isso pode surpreendê-lo. O React faz uma série de pequenas alterações no modelo de evento do lado do cliente e uma delas é que, no React, os nomes das propriedades do manipulador de eventos são escritos em camelCase:onClick, onMouseOver e assim por diante. Ao trabalhar com a plataforma web nativamente, no entanto, as propriedades do manipulador de eventos são escritas inteiramente em minúsculas.

4 A especificação do elemento personalizado permite a subclasse de <button> e outras classes de elementos específicos, mas isso não é suportado no Safari e uma sintaxe diferente é necessária para usar um elemento personalizado que estenda qualquer coisa diferente de HTMLElement.

Capítulo 16. JavaScript do lado do servidor com nó

Node é JavaScript com ligações ao sistema operacional subjacente, tornando possível escrever programas JavaScript que leem e gravam arquivos, executam processos filhos e se comunicam pela rede. Isso torna o Node útil como:

Alternativa moderna para scripts de shell que não sofrem com a sintaxe arcana do bash e de outros shells Unix. Linguagem de programação de uso geral para executar programas confiáveis, não sujeita às restrições de segurança impostas por navegadores da web em código não confiável. Ambiente popular para escrever servidores web eficientes e altamente simultâneos. O recurso definidor do Node é sua simultaneidade baseada em eventos de thread único habilitada por uma API assíncrona por padrão. Se você programou em outras linguagens, mas não fez muita codificação JavaScript, ou se você é um programador JavaScript experiente do lado do cliente acostumado a escrever código para navegadores da web, usar o Node será um pouco de ajuste, assim como qualquer nova linguagem de programação ou ambiente. Este capítulo começa explicando o modelo de programação do Node, com ênfase na simultaneidade, na API do Node para trabalhar com dados de streaming e no tipo de buffer do Node para trabalhar com dados binários. Essas seções iniciais são seguidas por seções que destacam e demonstram

algumas das APIs de nó mais importantes, incluindo aquelas para trabalhar com arquivos, redes, processos e threads.

Um capítulo não é suficiente para documentar todas as APIs do Node, mas minha esperança é que este capítulo explique o suficiente dos fundamentos para torná-lo produtivo com o Node e confiante de que você pode dominar todas as novas APIs de que precisa.

INSTALANDO O NÓ

Node é um software de código aberto. Visite <https://nodejs.org> para baixar e instalar o Node para Windows e MacOS. No Linux, você pode instalar o Node com seu gerenciador de pacotes normal ou pode visitar <https://nodejs.org/en/download> para baixar os binários diretamente. Se você trabalha em software em contêineres, pode encontrar imagens oficiais do Node Docker em <https://hub.docker.com>.

Além do executável do Node, uma instalação do Node também inclui o npm, um gerenciador de pacotes que permite fácil acesso a um vasto ecossistema de ferramentas e bibliotecas JavaScript. Os exemplos neste capítulo usarão apenas os pacotes integrados do Node e não exigirão npm ou quaisquer bibliotecas externas.

Por fim, não negligencie a documentação oficial do Node, disponível em <https://nodejs.org/api> and <https://nodejs.org/docs/guides>. Descobri que está bem organizado e bem escrito.

16.1 Noções básicas de programação de nós

Começaremos este capítulo com uma rápida olhada em como os programas Node são estruturados e como eles interagem com o sistema operacional.

16.1.1 Saída do console

Se você está acostumado com a programação JavaScript para navegadores da web, uma das pequenas surpresas sobre o Node é que o `console.log()` não é apenas para depuração, mas é a maneira mais fácil do Node de exibir uma mensagem para o usuário ou, mais geralmente, enviar saída para o fluxo `stdout`. Aqui está o programa clássico "Hello World" no Node:

```
console.log("Olá, Mundo!");
```

Existem maneiras de nível inferior de escrever para stdout, mas nenhuma maneira mais sofisticada ou mais oficial do que simplesmente chamar console.log().

Em navegadores da web, console.log(), console.warn() e console.error() normalmente exibem pequenos ícones ao lado de sua saída no console do desenvolvedor para indicar a variedade da mensagem de log. O Node não faz isso, mas a saída exibida com console.error() é diferenciada da saída exibida com console.log() porque console.error() grava no fluxo stderr. Se você estiver usando o Node para escrever um programa projetado para ter stdout redirecionado para um arquivo ou pipe, você pode usar console.error() para exibir texto no console onde o usuário o verá, mesmo que o texto impresso com console.log() esteja oculto.

16.1.2 Argumentos de linha de comando e variáveis de ambiente

Se você já escreveu programas no estilo Unix projetados para serem invocados a partir de um terminal ou outra interface de linha de comando, você sabe que esses programas normalmente obtêm sua entrada principalmente de argumentos de linha de comando e, secundariamente, de variáveis de ambiente.

O Node segue essas convenções do Unix. Um programa Node pode ler seus argumentos de linha de comando a partir do array de strings process.argv. O primeiro elemento desse array é sempre o caminho para o executável Node. O segundo argumento é o caminho para o arquivo de JavaScript code que o Node está executando. Todos os elementos restantes nesta matriz são

os argumentos separados por espaço que você passou na linha de comando quando você invocou o Node.

Por exemplo, suponha que você salve este programa Node muito curto no fileargv.js:

```
console.log(process.argv);
```

Você pode então executar o programa e ver a saída como esta:

```
$ node --trace-uncatched argv.js --arg1 --arg2 nome do
arquivo[
  '/usr/local/bin/node', '/',
  'private/tmp(argv.js', '--
arg1', '--arg2', 'nome do
arquivo']
```

Há algumas coisas a serem observadas aqui:

- O primeiro e o segundo elementos de process.argv serão caminhos do sistema de arquivos totalmente qualificados para o executável do Node e o arquivo de JavaScript que está sendo executado, mesmo que você não os tenha digitado dessa maneira. Os argumentos de linha de comando destinados e interpretados pelo próprio executável do Node são consumidos pelo executável do Nó e não aparecem em process.argv. (O argumento de linha de comando --trace-uncatched não está realmente fazendo nada de útil no exemplo anterior; é apenas para demonstrar que ele não aparece na saída.) Quaisquer argumentos (como --arg1 e nome do arquivo) que aparecerem após o nome do arquivo JavaScript aparecerão em process.argv.

Os programas de nó também podem receber entrada de variáveis de ambiente no estilo Unix. O Node os disponibiliza por meio do `process.env`. Os nomes de propriedade desse objeto são nomes de variáveis de ambiente e os valores de propriedade (sempre cadeias de caracteres) são os valores dessas variáveis.

Aqui está uma lista parcial de variáveis de ambiente no meu sistema:

```
$ node -p -e
'process.env'{
  SHELL: '/bin/bash',USER: 'david',PATH:
  '/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin',PWD:
  '/tmp',LANG: 'en_US. UTF-8',HOME: '/Usuários/david',}
```

Você pode usar `node -h` ou `node --help` para descobrir o que o `-p` e `-e` argumentos de linha de comando sim. No entanto, como dica, observe que você pode reescrever a linha acima como `node --eval 'process.env' -i` imprimir.

16.1.3 Ciclo de Vida do Programa

O comando `node` espera que um argumento de linha de comando que especifica o arquivo de código JavaScript seja executado. Este arquivo inicial normalmente importa outros módulos de código JavaScript e também pode definir suas próprias classes e funções. Fundamentalmente, no entanto, o Node executa o código JavaScript no arquivo especificado de cima para baixo. Alguns programas do Node saem quando terminam de executar a última linha de código no arquivo. Muitas vezes, no entanto, um programa Node continuará em execução por muito tempo após a execução do arquivo inicial. Como discutiremos nas seções a seguir, o Node

Os programas geralmente são assíncronos e baseados em retornos de chamada e manipuladores de eventos. Os programas de nó não saem até que terminem de executar o arquivo inicial e até que todos os retornos de chamada tenham sido chamados e não haja mais eventos pendentes. Um programa de servidor baseado em Node que escuta as conexões de rede de entrada teoricamente será executado para sempre porque sempre estará esperando por mais eventos.

Um programa pode se forçar a sair chamando `process.exit()`. Os usuários geralmente podem encerrar um programa Node digitando Ctrl-C na janela do terminal onde o programa está sendo executado.

Um programa pode ignorar Ctrl-C registrando uma função de manipulador de sinal com `process.on("SIGINT", ()=>{})`.

Se o código em seu programa lançar uma exceção e nenhuma cláusula `catch` a capturar, o programa imprimirá um rastreamento de pilha e sairá. Devido à natureza assíncrona do Node, as exceções que ocorrem em retornos de chamada ou manipuladores de eventos devem ser tratadas localmente ou não devem ser tratadas, o que significa que lidar com exceções que ocorrem nas partes assíncronas do seu programa pode ser um problema difícil. Se você não quiser que essas exceções façam com que seu programa falhe completamente, registre uma função global `handler` que será invocada em vez de travar:

```
process.setUncaughtExceptionCaptureCallback(e => {
  console.error("Exceção não capturada:",
  e);});
```

Uma situação semelhante surge se uma promessa criada pelo seu programa for rejeitada e não houver invocação `.catch()` para lidar com ela. A partir do Node 13, este não é um erro fatal que faz com que seu programa seja encerrado, mas imprime uma mensagem de erro detalhada no console. Em alguma versão futura do

Espera-se que as rejeições de promessas não tratadas se tornem erros fatais. Se você não quiser rejeições não tratadas, imprimir mensagens de erro ou encerrar seu programa, registre uma função de manipulador global:

```
process.on("unhandledRejection", (reason, promise) => { //  
  reason é qualquer valor que teria sido passado para a  
  função a.catch()  
  promise é o objeto Promise que rejeitou});
```

16.1.4 Módulos de nó

O Capítulo 10 documentou os sistemas de módulos JavaScript, abrangendo os módulos Node e os módulos ES6. Como o Node foi criado antes que o JavaScript tivesse um sistema de módulos, o Node teve que criar o seu próprio. O sistema de módulos do Node usa a função require() para importar valores para um módulo e o objeto exports ou a propriedade module.exports para exportar valores de um módulo. Eles são uma parte fundamental do modelo de programação do Node e são abordados em detalhes no §10.2.

O Node 13 adiciona suporte para módulos ES6 padrão, bem como módulos baseados em requisitos (que o Node chama de "módulos CommonJS"). Os sistemas de dois módulos não são totalmente compatíveis, então isso é um pouco complicado. O Node precisa saber, antes de carregar um módulo, se esse módulo usará require() e module.exports ou se usará import e export. Quando o Node carrega um arquivo de código JavaScript como um módulo CommonJS, ele define automaticamente a função require() junto com as exportações de identificadores e o módulo, e não habilita as palavras-chave de importação e exportação. Por outro lado, quando o Node carrega um arquivo de código como um módulo ES6, ele deve habilitar as declarações de importação e exportação, e não deve definir

identificadores extras, como require, module e exports.

A maneira mais simples de dizer ao Node que tipo de módulo ele está carregando é codificar essas informações na extensão do arquivo. Se você salvar seu código JavaScript em um arquivo que termina com .mjs, o Node sempre o carregará como um módulo ES6, esperará que ele use import e export e não fornecerá uma função require(). E se você salvar seu código em um arquivo que termina com .cjs, o Node sempre o tratará como um módulo CommonJS, fornecerá uma função require() e lançará um SyntaxError se você usar declarações de importação ou exportação.

Para arquivos que não têm uma extensão .mjs ou .cjs explícita, o Node procura um arquivo chamado package.json no mesmo diretório que o arquivo e, em seguida, em cada um dos diretórios que o contêm. Depois que o arquivo package.json mais próximo for encontrado, o Node verificará se há uma propriedade de tipo de nível superior no objeto JSON. Se o valor da propriedade type for "module", o Node carregará o arquivo como um módulo ES6. Se o valor dessa propriedade for "commonjs", o Node carregará o arquivo como um módulo CommonJS. Observe que você não precisa ter um arquivo package.json para executar programas do Node: quando nenhum arquivo desse tipo é encontrado (ou quando o arquivo é encontrado, mas não tem uma propriedade type), o Node usa módulos CommonJS por padrão. This package.json truque só se torna necessário se você quiser usar ES6modules com o Node e não quiser usar a extensão de arquivo .mjs.

Como há uma enorme quantidade de código Node existente escrito usando o formato de módulo CommonJS, o Node permite que os módulos ES6 carreguem módulos CommonJS usando a palavra-chave import. O inverso não é true, no entanto: um módulo CommonJS não pode usar require() para carregar

um módulo ES6.

16.1.5 O Gerenciador de Pacotes de Nô

Quando você instala o Node, normalmente obtém um programa chamado npm também. Este é o Node Package Manager e ajuda você a baixar e gerenciar as bibliotecas das quais seu programa depende. O npm mantém o controle dessas dependências (bem como outras informações sobre seu programa) em um arquivo chamado package.json no diretório raiz do seu projeto. Este arquivo package.json criado pelo npm é onde você adicionaria "type": "module" se quisesse usar os módulos ES6 para o seu projeto.

Este capítulo não cobre o npm em nenhum detalhe (mas veja §17.4 para um pouco mais de profundidade). Estou mencionando isso aqui porque, a menos que você escreva programas que não usam bibliotecas externas, quase certamente estará usando npm ou uma ferramenta semelhante. Suponha, por exemplo, que você esteja desenvolvendo um servidor web e planeje usar o framework Express (<https://expressjs.com>) para simplificar a tarefa. Para começar, você pode criar um diretório para seu projeto e, nesse diretório, digite npminit. O npm solicitará o nome do seu projeto, número da versão, etc., e criará um arquivo package.json inicial com base em suas respostas.

Agora, para começar a usar o Express, digite npm install express. Isso diz ao npm para baixar a biblioteca Express junto com todas as suas dependências e instalar todos os pacotes em um node_modules/diretório local:

```
$ npm instalar express
```

```
npm criou um arquivo de bloqueio como package-lock.json. Você deve confirmar este arquivo.
npm WARN my-server@1.0.0 Sem descrição
npm WARN my-server@1.0.0 Nenhum campo de repositório.

+ express@4.17.1 adicionou 50 pacotes de 37
  colaboradores e auditou 126 pacotes em 3.058s
  encontrou 0 vulnerabilidades
```

Quando você instala um pacote com o npm, o npm registra essa dependência — que seu projeto depende do Express — no arquivo package.json. Com essa dependência registrada no package.json, você pode dar a outro programador uma cópia do seu código e do seu package.json, e eles podem simplesmente digitar npm install para baixar e instalar automaticamente todas as bibliotecas que seu programa precisa para ser executado.

16.2 O nó é assíncrono por padrão

JavaScript é uma linguagem de programação de uso geral, por isso é perfeitamente possível escrever programas com uso intensivo de CPU que multiplicam matrizes grandes ou realizam análises estatísticas complicadas. Mas o Node foi projetado e otimizado para programas - como servidores de rede - que são intensivos em E/S. E, em particular, o Node foi projetado para tornar possível implementar facilmente servidores altamente simultâneos que podem lidar com muitas solicitações ao mesmo tempo.

Ao contrário de muitas linguagens de programação, no entanto, o Node não alcança simultaneidade com threads. A programação multithread é notoriamente difícil de fazer corretamente e difícil de depurar. Além disso, os threads são uma abstração relativamente pesada e, se você quiser escrever um servidor, pode lidar com centenas de solicitações simultâneas, usando centenas de

Os threads podem exigir uma quantidade proibitiva de memória. Portanto, o Node adota o modelo de programação JavaScript de thread único que a web usa, e isso acaba sendo uma grande simplificação que torna a criação de servidores de rede uma habilidade rotineira e não misteriosa.

VERDADEIRO PARALELISMO COM NÓ

Os programas do Node podem executar vários processos do sistema operacional, e o Node 10 e posterior suportam [Workerobjects](#) (§16.11), que são um tipo de thread emprestado de navegadores da web. Se você usar vários processos ou criar um ou mais threads de trabalho e executar seu programa em um sistema com mais de uma CPU, seu programa não será mais um thread único e seu programa realmente executará vários fluxos de código em paralelo. Essas técnicas podem ser valiosas para operações com uso intensivo de CPU, mas não são comumente usadas para programas com uso intensivo de E/S, como servidores.

Vale a pena notar, no entanto, que os processos e Workers do Node evitam a complexidade típica da programação multithread porque a comunicação entre processos e inter-Worker é via passagem de mensagens e eles não podem compartilhar memória facilmente entre si.

O Node atinge altos níveis de simultaneidade, mantendo um modelo de programação de thread único, tornando sua API assíncrona e sem bloqueio por padrão. O Node leva sua abordagem sem bloqueio muito a sério e a um extremo que pode surpreendê-lo. Você provavelmente espera que as funções que leem e gravam na rede sejam assíncronas, mas o Node vai além e define funções assíncronas sem bloqueio para ler e gravar arquivos do sistema de arquivos local. Isso faz sentido, quando você pensa sobre isso: a API do Node foi projetada nos dias em que os discos rígidos giratórios ainda eram a norma e realmente havia milissegundos de bloqueio do "tempo de busca" enquanto esperava que o disco girasse antes que uma operação de arquivo pudesse começar. E em datacenters modernos, o sistema de arquivos "local" pode realmente estar em toda a rede em algum lugar com latências de rede em cima de drivelatencies. Mas mesmo que a leitura de um arquivo de forma assíncrona pareça normal para você, o Node vai ainda mais longe: as funções padrão para iniciar um

conexão de rede ou procurar um tempo de modificação de arquivo, por exemplo, também não são bloqueados.

Algumas funções na API do Node são síncronas, mas não bloqueiam: elas são executadas até a conclusão e retornam sem precisar bloquear. Mas a maioria das funções interessantes executa algum tipo de entrada ou saída, e essas são funções assíncronas para que possam evitar até mesmo a menor quantidade de bloqueio. O Node foi criado antes do JavaScript ter uma classe Promise, portanto, as APIs de nó assíncronas são baseadas em retorno de chamada. (Se você ainda não leu ou já esqueceu o Capítulo 13, este seria um bom momento para voltar a esse capítulo.) Geralmente, o último argumento que você passa para uma função Node assíncrona é um retorno de chamada. O Node usa retornos de chamada error-first, que normalmente são invocados com dois argumentos. O primeiro argumento para um retorno de chamada de erro primeiro é normalmente nulo no caso em que nenhum erro ocorreu, e o segundo argumento é qualquer dado ou resposta que tenha sido produzido pela função assíncrona original que você chamou. A razão para colocar o argumento de erro primeiro é tornar impossível para você omiti-lo, e você deve sempre verificar se há um valor não nulo neste argumento. Se for um objeto Error, ou mesmo um código de erro inteiro ou uma mensagem de erro de string, algo deu errado. Nesse caso, o segundo argumento para sua função de retorno de chamada provavelmente será nulo.

O código a seguir demonstra como usar a função `nonblockingreadFile()` para ler um arquivo de configuração, analisá-lo como JSON e, em seguida, passar o objeto de configuração analisado para outro retorno de chamada:

```
const fs = require("fs"); Requer o módulo do sistema de arquivos
Leia um arquivo de configuração, analise seu conteúdo como JSON e passe
```

o// valor resultante para o retorno de chamada. Se algo der errado,// imprima uma mensagem de erro em stderr e invoque o retorno de chamada com nullfunction

```
readConfigFile(path, callback) {

    fs.readFile(caminho, "utf8", (err, texto) => {
        se (err) {    Algo deu errado ao ler o
arquivo
            console.error(err);
            callback(null);
            retornar;}let dados =
nulo; tente {

            dados = JSON.parse(texto);} catch(e) {// Algo
deu errado ao analisar o
Conteúdo do arquivo
            console.error(e);}callback(d
ados);});}
```

O Node é anterior às promessas padronizadas, mas como é bastante consistente em relação aos retornos de chamada que priorizam o erro, é fácil criar variantes baseadas em promessas de suas APIs baseadas em retorno de chamada usando o wrapper `util.promisify()`. Veja como poderíamos reescrever a função `readConfigFile()` para retornar uma promessa:

```
const util = require("util"); const fs = require("fs");// Requer o módulo do sistema de arquivos
const pfs = {}// Variantes baseadas em promessas de algumas funções

readFile: util.promisify(fs.readFile);

function readConfigFile(caminho) {
    return pfs.readFile(caminho, "utf-8").then(texto => {
```

```
return JSON.parse(texto);}}
```

Também podemos simplificar a função baseada em promessa anterior usando `async` e `await` (novamente, se você ainda não leu o Capítulo 13, este seria um bom momento para fazê-lo):

```
função assíncrona readConfigFile(caminho) {  
let text = await pfs.readFile(caminho, "utf-8");  
return JSON.parse(texto);}
```

O wrapper `util.promisify()` pode produzir uma versão baseada em `Promise` de muitas funções do Node. No Nô 10 e posterior, o objeto `thefs.promises` tem várias funções predefinidas baseadas em promessas para trabalhar com o sistema de arquivos. Vamos discutir os mais adiante neste capítulo, mas observe que, no código anterior, poderíamos substituir `pfs.readFile()` por `fs.promises.readFile()`.

Dissemos que o modelo de programação do Node é assíncrono por padrão. Mas para conveniência do programador, o Node define bloqueios, variantes síncronas de muitas de suas funções, especialmente no módulo do sistema de arquivos. Essas funções normalmente têm nomes claramente rotulados como `withSync` no final.

Quando um servidor está a arrancar pela primeira vez e está a ler os seus ficheiros de configuração, ainda não está a lidar com pedidos de rede e é realmente possível pouca ou nenhuma simultaneidade. Portanto, nesta situação, não há realmente necessidade de evitar bloqueando, e podemos usar com segurança funções de bloqueio como `fs.readFileSync()`. Podemos descartar o assíncrono e aguardar de

este código e escrever uma versão puramente síncrona da função ourreadConfigFile(). Em vez de invocar um retorno de chamada ou retornar uma promessa, essa função simplesmente retorna o valor JSON analisado ou lança uma exceção:

```
const fs = require("fs"); function  
readConfigFileSync(caminho) {  
let texto = fs.readFileSync(caminho, "utf-  
8"); return JSON.parse(texto);}
```

Além de seus retornos de chamada de dois argumentos de erro primeiro, o Node também tem várias APIs que usam assincronia baseada em eventos, normalmente para lidar com dados de streaming. Abordaremos os eventos do Node com mais detalhes posteriormente.

Agora que discutimos a API agressivamente sem bloqueio do Node, vamos voltar ao tópico da simultaneidade. As funções de não bloqueio integradas do Node funcionam usando a versão do sistema operacional de retornos de chamada e manipuladores de eventos. Quando você chama uma dessas funções, o Node toma uma ação para iniciar a operação e, em seguida, registra algum tipo de manipulador de eventos com o sistema operacional para que ele seja notificado quando a operação for concluída. O retorno de chamada que você passou para a função Node é armazenado internamente para que o Node possa invocar seu retorno de chamada quando o sistema operacional enviar o evento apropriado para o Node.

Esse tipo de simultaneidade geralmente é chamado de simultaneidade baseada em eventos. Em sua essência, o Node tem um único thread que executa um "loop de eventos". Quando o programa aNode é iniciado, ele executa qualquer código que você disse para ele executar. Presumivelmente, esse código chama pelo menos uma função sem bloqueio, fazendo com que um retorno de chamada ou manipulador de eventos seja registrado no sistema operacional. (Se

não, então você escreveu um programa Node síncrono e o Node simplesmente sai quando chega ao fim.) Quando o Node chega ao final do seu programa, ele bloqueia até que um evento aconteça, momento em que o sistema operacional o inicia em execução novamente. O nó mapeia o evento do sistema operacional para o retorno de chamada JavaScript que você registrou e, em seguida, invoca essa função. Sua função de retorno de chamada pode invocar mais funções de nó sem bloqueio, fazendo com que os manipuladores de eventos moreOS sejam registrados. Depois que sua função de retorno de chamada é executada, o Node volta a dormir novamente e o ciclo se repete.

Para servidores Web e outros aplicativos com uso intensivo de E/S que passam a maior parte do tempo aguardando entrada e saída, esse estilo de simultaneidade baseada em eventos é eficiente e eficaz. Um servidor web pode lidar simultaneamente com solicitações de 50 clientes diferentes sem precisar de 50 threads diferentes, desde que use APIs sem bloqueio e haja algum tipo de mapeamento interno de soquetes de rede para funções JavaScript para invocar quando ocorrer atividade nesses soquetes.

16.3 Amortecedores

Um dos tipos de dados que você provavelmente usará com frequência no Node, especialmente ao ler dados de arquivos ou da rede, é a classe Buffer. Um Buffer é muito parecido com uma string, exceto que é uma sequência de bytes em vez de uma sequência de caracteres. O nó foi criado antes que o JavaScript principal suportasse matrizes tipadas (consulte §11.2) e havia no Uint8Array para representar uma matriz de bytes não assinados. Node definiu a classe Buffer para preencher essa necessidade. Agora que Uint8Array faz parte da linguagem JavaScript, a classe Buffer do Node é uma subclasse de Uint8Array.

O que distingue o Buffer de sua superclasse Uint8Array é que ele é

projeto para interoperar com strings JavaScript: os bytes em um buffer podem ser inicializados a partir de strings de caracteres ou convertidos em strings de caracteres. Uma codificação de caracteres mapeia cada caractere em algum conjunto de caracteres para um número inteiro. Dada uma string de texto e uma codificação de caracteres, podemos codificar os caracteres na string em uma sequência de bytes. E dada uma sequência (devidamente codificada) de bytes e uma codificação de caracteres, podemos decodificar esses bytes em uma sequência de caracteres. A classe Buffer do Node tem métodos que executam codificação e decodificação, e você pode reconhecer esses métodos porque eles esperam um argumento de codificação que especifica a codificação a ser usada.

As codificações no Node são especificadas por nome, como strings. As codificações com suporte são:

"utf8"

Este é o padrão quando nenhuma codificação é especificada e é a codificação Unicode que você provavelmente usará.

"UTF16LE"

Caracteres Unicode de dois bytes, com ordenação little-endian. Os pontos de código acima de \uffff são codificados como um par de sequências de dois bytes. A codificação "ucs2" é um alias.

"latin1"

A codificação ISO-8859-1 de um byte por caractere que define um conjunto de caracteres adequado para muitos idiomas da Europa Ocidental. Como há um mapeamento um-para-um entre bytes e caracteres latin-1, essa codificação também é conhecida como "binária".

"ASCII"

A codificação ASCII somente em inglês de 7 bits, um subconjunto estrito da codificação "utf8".

"Feitiço"

Essa codificação converte cada byte em um par de dígitos hexadecimais ASCII.

"base64"

Essa codificação converte cada sequência de três bytes em uma sequência de quatro caracteres ascii.

Aqui está um código de exemplo que demonstra como trabalhar com Buffers e como converter de e para strings:

```
let b = Buffer.from([0x41, 0x42, 0x43]);           <Buffer  
41 42  
43>b.toString()                                // =>  
"ABC"; padrão  
"utf8"b.toString("hex")                          // =>  
"414243"  
  
let computador = Buffer.from("IBM3111", "ascii"); Converter  
string para Bufferfor(let i = 0; i <               //  
computador.length; i++) {                         Usar  
Buffer como matriz de bytes  
    computador [i] --;                            Buffers  
são  
mutáveis}computador.toString()  
("ascii")                                         // =>  
"HAL2000"computador.subarray(0,3).map(x=>x+1  
).toString()                                     => "IBM"  
  
Crie novos buffers "vazios" com  
Buffer.alloc()let zeros = Buffer.alloc(1024);      // 1024  
zeroslet uns =  
Buffer.alloc(128, 1);                            128 unidades
```

```
let dead = Buffer.alloc(1024, "DEADBEEF", "hex");  
Padrão repetitivo de bytes
```

```
Os buffers têm métodos para ler e gravar valores  
multibytes// de e para um buffer em qualquer deslocamento  
especificado.  
dead.readUInt32BE(0)// =>  
0xDEADBEEF  
dead.readUInt32BE(1)// =>  
0xADBEEFDE  
dead.readBigUInt64BE(6)// =>  
0xBEEFDEADBEEFDEAD  
dead.readUInt32LE(1020)// =>  
0xEFBEADDE
```

Se você escrever um programa Node que realmente manipula dados binários, poderá usar a classe Buffer extensivamente. Por outro lado, se você estiver apenas trabalhando com texto lido ou gravado em um arquivo ou na rede, poderá encontrar apenas o Buffer como uma representação intermediária de seus dados. Várias APIs do Node podem receber entrada ou retornar saída como strings ou objetos Buffer. Normalmente, se você passar uma cadeia de caracteres ou esperar que uma cadeia de caracteres seja retornada de uma dessas APIs, precisará especificar o nome da codificação de texto que deseja usar. E se você fizer isso, talvez não precise usar um objeto Buffer.

16.4 Eventos e EventEmitter

Conforme descrito, todas as APIs do Node são assíncronas por padrão. Para muitos deles, essa assincronia assume a forma de retornos de chamada de erro de dois argumentos que são invocados quando a operação solicitada é concluída. Mas algumas das APIs mais complicadas são baseadas em eventos em vez disso. Normalmente, esse é o caso quando a API é projetada em torno de um objeto em vez de uma função, ou quando uma função de retorno de chamada precisa ser invocada várias vezes ou quando há vários tipos de funções de retorno de chamada que podem ser necessárias. Considere a rede. Server, para

Exemplo: Um objeto desse tipo é um soquete de servidor usado para aceitar conexões de entrada de clientes. Ele emite um evento de "escuta" quando começa a escutar conexões, um evento de "conexão" toda vez que um cliente se conecta e um evento de "fechamento" quando foi fechado e não está mais escutando.

No Node, os objetos que emitem eventos são instâncias de EventEmitter ou uma subclasse de EventEmitter:

```
const EventEmitter = require("eventos"); O nome do módulo  
não corresponde ao nome da classeconst net =  
require("net"); let server = nova rede. Server();// cria  
uma instância Serverobjectserverde EventEmitter// => true:  
Serversare EventEmitters
```

A principal característica do EventEmitters é que eles permitem que você registre funções do manipulador de eventos com o método on(). Os emissores podem emitir vários tipos de eventos, e os tipos de eventos são identificados pelo nome. Para registrar um manipulador de eventos, chame o método on(), passando o nome do tipo de evento e a função que deve ser invocada quando ocorrer um evento desse tipo. Os EventEmitters podem invocar funções de manipulador com qualquer número de argumentos, e você precisa ler a documentação de um tipo específico de evento de um EventEmitter específico para saber quais argumentos você deve esperar que sejam passados:

```
const net = require("net"); let  
server = nova rede. Servidor();          criar um servidor  
objectserver.on("conexão", soquete  
=> {                                A lista para  
Eventos de "conexão"  
Os eventos de "conexão" do servidor são passados um objeto de  
soquete// para o cliente que acabou de se conectar. Aqui enviamos  
alguns
```

```
dados
para o cliente e
disconnect.socket.end("Hello World",
"utf8");};
```

Se você preferir nomes de método mais explícitos para registrar eventlisteners, também poderá usar addListener(). E você pode remover um ouvinte de evento registrado anteriormente com off() ou removeListener(). Como um caso especial, você pode registrar um eventlistener que será removido automaticamente depois que ele for acionado pela primeira vez chamando once() em vez de on().

Quando um evento de um tipo específico ocorre para um objeto EventEmitter específico, o Node invoca todas as funções de manipulador que estão atualmente registradas nesse EventEmitter para eventos desse tipo. São invocados por ordem do primeiro para o último registrado. Se houver mais de uma função de manipulador, elas serão invocadas sequencialmente em um único thread: não há paralelismo no Node, lembre-se. E, mais importante, as funções de manipulação de eventos são invocadas de forma síncrona, notassíncrona. O que isso significa é que o método emit() não enfileira manipuladores de eventos para serem invocados posteriormente. emit()invoca todos os manipuladores registrados, um após o outro, e não retorna até que o último manipulador de eventos seja retornado.

O que isso significa, na verdade, é que quando uma das APIs de nó integradas emite um evento, essa API está basicamente bloqueando seus manipuladores de eventos. Se você escrever um manipulador de eventos que chama uma função de bloqueio como fs.readFileSync(), nenhum tratamento de eventos adicional acontecerá até que a leitura síncrona do arquivo seja concluída. Se o seu programa é um - como um servidor de rede - que precisa ser responsivo, então é importante que

Você mantém suas funções de manipulador de eventos sem bloqueio e rápidas. Se você precisar fazer muita computação quando um evento ocorre, geralmente é melhor usar o manipulador para agendar essa computação de forma assíncrona usando `setTimeout()` (consulte §11.10). O nó também define `setImmediate()`, que agenda uma função a ser invocada imediatamente após todos os retornos de chamada e eventos pendentes terem sido tratados.

A classe `EventEmitter` também define um método `emit()` que faz com que as funções do manipulador de eventos registradas sejam invocadas. Isso é útil se você estiver definindo sua própria API baseada em eventos, mas não é comumente usado quando você está apenas programando com APIs existentes. `emit()` deve ser invocado com o nome do tipo de evento como seu primeiro argumento. Quaisquer argumentos adicionais que são passados para `emit()` tornam-se argumentos para as funções do manipulador de eventos registradas. As funções do manipulador também são invocadas com o valor `this` definido como o próprio objeto `EventEmitter`, o que geralmente é conveniente. (Lembre-se, porém, de que as funções de seta sempre usam o valor `this` do contexto no qual são definidas e não podem ser invocadas com nenhum outro valor `this`. No entanto, as funções de seta geralmente são a maneira mais conveniente de escrever manipuladores de eventos.)

Qualquer valor retornado por uma função de manipulador de eventos é ignorado. Se uma função eventhandler lançar uma exceção, no entanto, ela se propagará a partir da chamada `emit()` e impedirá a execução de quaisquer funções de manipulador que foram registradas após aquela que lançou a exceção.

Lembre-se de que as APIs baseadas em retorno de chamada do Node usam retornos de chamada de erro primeiro, e é importante que você sempre verifique o primeiro argumento de retorno de chamada para ver se

Ocorreu um erro. Com APIs baseadas em eventos, o equivalente são eventos de "erro". Como as APIs baseadas em eventos são frequentemente usadas para rede e outras formas de E/S de streaming, elas são vulneráveis a erros assíncronos imprevisíveis, e a maioria dos EventEmitters define um evento de "erro" que eles emitem quando ocorre um erro. Sempre que você usar uma API baseada em eventos, crie o hábito de registrar um manipulador para eventos de "erro". Error" recebem tratamento especial pela classe EventEmitter. Ifemit() é chamado para emitir um evento de "erro" e, se não houver manipuladores registrados para esse tipo de evento, uma exceção será lançada. Como isso ocorre de forma assíncrona, não há como lidar com a exceção em um bloco catch, portanto, esse tipo de erro normalmente faz com que seu programa seja encerrado.

16.5 Transmissões

Ao implementar um algoritmo para processar dados, quase sempre é mais fácil ler todos os dados na memória, fazer o processamento e, em seguida, gravar os dados. Por exemplo, você pode escrever uma função Node para copiar um arquivo como este.

```
const fs = require("fs");

Uma função assíncrona, mas não transmitida (e,
portanto, ineficiente).function
copyFile(sourceFilename,
destinationFilename,callback) {
  fs.readFile(sourceFilename, (err, buffer) => {
    se (err) {
      callback(err);}
    else {
      fs.writeFile(nome_doarquivo_destino, buffer,
      retorno de chamada);
```

```
});}
```

Esta função copyFile() usa funções assíncronas e retornos de chamada, portanto, não bloqueia e é adequada para uso em programas simultâneos como servidores. Mas observe que ele deve alocar memória suficiente para manter todo o conteúdo do arquivo na memória de uma só vez. Isso pode ser bom em alguns casos de uso, mas começa a falhar se os arquivos a serem copiados forem muito grandes, ou se o seu programa for altamente simultâneo e houver muitos arquivos sendo copiados ao mesmo tempo. Outra falha da implementação thiscopyFile() é que ela não pode começar a escrever o novo arquivo até terminar de ler o arquivo antigo.

A solução para esses problemas é usar algoritmos de streaming em que os dados "fluem" para o seu programa, são processados e, em seguida, fluem para fora do seu programa. A ideia é que seu algoritmo processe os dados em pequenos pedaços e o conjunto de dados completo nunca seja mantido na memória de uma só vez. Quando as soluções de streaming são possíveis, elas são mais eficientes em termos de memória e também podem ser mais rápidas. As APIs de rede do Node são baseadas em fluxo e o módulo de sistema de arquivos do Node define APIs de streaming para leitura e gravação de arquivos, portanto, é provável que você use uma API de streaming em muitos dos programas do Node que você escreve. Veremos uma versão de streaming da função thecopyFile() em "Modo de fluxo".

O Node suporta quatro tipos básicos de fluxo:

Legível

Fluxos legíveis são fontes de dados. O fluxo retornado por

`fs.createReadStream()`, por exemplo, é um fluxo do qual o conteúdo de um arquivo especificado pode ser lido. `process.stdin` é outro fluxo legível que retorna dados da entrada padrão.

Gravável

Fluxos graváveis são coletores ou destinos para dados. O valor de retorno de `fs.createWriteStream()`, por exemplo, é um Writablestream: ele permite que os dados sejam gravados nele em partes e gera todos esses dados para um arquivo especificado.

Duplex

Os fluxos duplex combinam um fluxo legível e um fluxo gravável em um objeto. Os objetos Socket retornados por `net.connect()` e outras APIs de rede do Node, por exemplo, são fluxos Duplex. Se você gravar em um soquete, seus dados serão enviados pela rede para qualquer computador ao qual o soquete esteja conectado. E se você lerde um soquete, você acessa os dados gravados por esse outro computador.

Transformar

Os fluxos de transformação também são legíveis e graváveis, mas diferem dos fluxos duplex de uma maneira importante: os dados gravados em um fluxo de transformação tornam-se legíveis, geralmente em alguma forma transformada, do mesmo fluxo. A função `zlib.createGzip()`, por exemplo, retorna um fluxo Transform que compacta (com o algoritmo gzip) os dados gravados nele. De maneira semelhante, a função `crypto.createCipheriv()` retorna um Transformstream que criptografa ou descriptografa os dados gravados nele.

Por padrão, os fluxos lêem e gravam buffers. Se você chamar o método `theEncoding()` de um fluxo legível, ele retornará strings decodificadas para você em vez de objetos Buffer. E se você escrever um

string para um buffer gravável, ele será codificado automaticamente usando a codificação padrão do thebuffer ou qualquer codificação que você especificar. A API de fluxo do Node também suporta um "modo de objeto" em que os fluxos leem e gravam objetos mais complexos do que buffers e strings. Nenhuma das APIs do Node'score usa esse modo de objeto, mas você pode encontrá-lo em outras bibliotecas.

Os fluxos legíveis precisam ler seus dados de algum lugar, e os fluxos graváveis precisam gravar seus dados em algum lugar, então cada fluxo tem duas extremidades: uma entrada e uma saída ou uma fonte e um destino. O complicado sobre as APIs baseadas em fluxo é que as duas extremidades do fluxo quase sempre fluirão em velocidades diferentes. Talvez o código que lê de um fluxo queira ler e processar dados mais rapidamente do que os dados estão realmente sendo gravados no fluxo. Ou inverso: talvez os dados sejam gravados em um fluxo mais rapidamente do que podem ser lidos e retirados do fluxo na outra extremidade. As implementações de fluxo quase sempre incluem um buffer interno para armazenar dados que foram gravados, mas ainda não lidos. O buffer ajuda a garantir que haja dados disponíveis para leitura quando solicitados e que haja espaço para armazenar dados quando eles forem gravados. Mas nenhuma dessas coisas pode ser garantida, e é da natureza da programação baseada em fluxo que os leitores às vezes terão que esperar que os dados sejam gravados (porque o buffer de fluxo está vazio) e os gravadores às vezes terão que esperar que os dados sejam lidos (porque o buffer de fluxo está cheio).

Em ambientes de programação que usam simultaneidade baseada em thread, as APIs de fluxo normalmente têm chamadas de bloqueio: uma chamada para ler dados não retorna até que os dados cheguem ao fluxo e uma chamada para gravar blocos de dados até que haja espaço suficiente no buffer interno do fluxo para

acomodar os novos dados. Com um modelo de simultaneidade baseado em eventos, no entanto, o bloqueio de chamadas não faz sentido, e as APIs de fluxo do Node são baseadas em eventos e retornos de chamada. Ao contrário de outras APIs do Node, não há versões "Sync" dos métodos que serão descritos posteriormente neste capítulo.

A necessidade de coordenar a legibilidade do fluxo (buffer não vazio) e a capacidade de gravação (buffer não cheio) por meio de eventos torna as APIs de fluxo do Node um pouco complicadas. Isso é agravado pelo fato de que essas APIs evoluíram e mudaram ao longo dos anos: para fluxos legíveis, existem duas APIs completamente distintas que você pode usar. Apesar da complexidade, vale a pena entender e dominar as APIs de streaming do Node, pois elas permitem E/S de alto rendimento em seus programas.

As subseções a seguir demonstram como ler e gravar as classes de fluxo do Node.

16.5.1 Tubulações

Às vezes, você precisa ler dados de um fluxo simplesmente para se virar e gravar esses mesmos dados em outro fluxo. Imagine, por exemplo, que você está escrevendo um servidor HTTP simples que serve um diretório de arquivos estáticos. Nesse caso, você precisará ler dados de um fluxo de entrada de arquivo e gravá-los em um soquete de rede. Mas, em vez de escrever seu próprio código para lidar com a leitura e a gravação, você pode simplesmente conectar os dois soquetes como um "tubo" e deixar o Node lidar com as complexidades para você. Basta passar o fluxo gravável para o método pipe() do fluxo legível:

```
const fs = require("fs");
```

```
function pipeFileToSocket(nome do arquivo, soquete) {  
  fs.createReadStream(nome do  
  arquivo).pipe(soquete);}
```

A seguinte função de utilitário canaliza um fluxo para outro e invoca um retorno de chamada quando concluído ou quando ocorre um erro:

```
pipe de função (legível, gravável, retorno de chamada) {  
  Primeiro, configure a função de  
  tratamento de erro handleError(err) {  
    readable.close();  
    writable.close();  
    callback(err);}
```

Em seguida, defina o pipe e manipule o caso de terminação normal

```
  legível  
    .on("erro",  
      handleError).pipe(gravável).on("e  
    rro", handleError).on("terminar",  
    retorno de chamada);  
}
```

Os fluxos de transformação são particularmente úteis com pipes e createpipelines que envolvem mais de dois fluxos. Aqui está um exemplo de função que compacta um arquivo:

```
const fs = require("fs");  
const zlib = require("zlib");  
  
function gzip(nome do arquivo, retorno de chamada) {  
  Crie a fonte do stream:  
  let source =  
    fs.createReadStream(filename);  
  let destino =  
    fs.createWriteStream(nome do arquivo + ".gz");  
  let  
    gzipper = zlib.createGzip();
```

```

Configure o
pipelinesource.on("error"
, callback)
erro
    .pipe(gzipper).pipe(d
estination).on("erro"
, retorno de chamada)    Retorno de chamada de chamada na
erro                                leitura
    .on("finish", retorno de chamada); Retorno de chamada na
a escrita está
completa}
                                gravação

```

Usar o método pipe() para copiar dados de um fluxo legível para um fluxo gravável é fácil, mas, na prática, muitas vezes você precisa processar os dados de alguma forma à medida que eles fluem pelo seu programa. Uma maneira de fazer isso é implementar seu próprio fluxo de transformação para fazer esse processamento, e essa abordagem permite que você evite ler e gravar manualmente os fluxos. Aqui, por exemplo, está uma função que funciona como o utilitário Unixgrep: ele lê linhas de texto de um fluxo de entrada, mas escreve apenas linhas que correspondem a uma expressão regular especificada:

```

const fluxo = require("fluxo");

classe GrepStream estende o fluxo. Transformar {
    construtor (padrão) {
        super({decodeStrings: false}); // Não converter
        strings de volta para
        buffersthis.pattern = pattern;      O regular
        expressão que queremos combinar
        this.incompleteLine = "";      Qualquer resquício do
        Último pedaço de dados
    }
}

```

Esse método é invocado quando há uma string pronta para ser transformado. Ele deve passar dados transformados para o especificado função de retorno de chamada. Esperamos entrada de string, então isso

```
fluxo deve
estar conectado apenas a fluxos legíveis que tiveram
// setEncoding() chamado em them._transform(chunk,
encoding, callback) {
    if (tipo de pedaço !== "string") {
        callback(new Error("Esperava uma string, mas recebi um
tampão"));
        retornar;}Adicione o pedaço a qualquer linha
incompleta anteriormente

e quebrar
tudo em linhas de linha =
(this.incompleteLine +
chunk).split("\n");

O último elemento da matriz é o novo
linha incompleta
this.incompleteLine = linhas.pop();

Encontre todas as linhas
correspondentes saída = linhas Comece com
todas as linhas completas,
.filter(l => this.pattern.test(l)) // filtre-os
for
matches,.join("\n"); e junte-se
eles voltam.

Se algo corresponder, adicione um
newlineif final (saída) {
saída += "\n";}

Sempre chame o retorno de chamada, mesmo que não haja
saída
callback(null, saída);}

Isso é chamado logo antes do fluxo ser fechado.// É
nossa chance de escrever qualquer último
data._flush(callback) {
Se ainda tivermos uma linha incompleta, e ela
Corresponde
passá-lo para o callbackif
(this.pattern.test(this.incompleteLine)) {
```

```
callback(null, this.incompleteLine + "\n");}}
```

Agora podemos escrever um programa como 'grep' com este class.let pattern = new RegExp(process.argv[2]); Obtenha um RegExpfrom linha de comando.process.stdin// Comece comentrada padrão,

*.setEncoding("utf8") leia como
Cadeias de caracteres Unicode,
.pipe(novo GrepStream(padrão)) canalize-o para o nosso
GrepStream,
e canplícepróbetao.pádovab)para fora.*

*.on("erro", () => processo.exit()); Saia graciosamente
se stdout fechar.*

16.5.2 Iteração assíncrona

No Nô 12 e posterior, os fluxos legíveis são iteradores assíncronos, o que significa que, em uma função assíncrona, você pode usar o loop afor/await para ler blocos de string ou buffer de um fluxo usando um código estruturado como o código síncrono seria. (Consulte §13.4 para obter mais informações sobre iteradores assíncronos e loops for/await.)

Usar um iterador assíncrono é quase tão fácil quanto usar o método pipe() e provavelmente é mais fácil quando você precisa processar cada pedaço que você lê de alguma forma. Veja como poderíamos reescrever o programa grep na seção anterior usando uma função assíncrona e um loop for/await:

Leia linhas de texto do fluxo de origem e escreva todas as linhas// que correspondam ao padrão especificado para o destino

```
stream.async function grep(origem, destino,  
padrão, codificação="utf8") {  
  
  Configurar o fluxo de origem para ler strings,  
  notBuffers  
    source.setEncoding(codificação);  
  
  Defina um manipulador de erros no fluxo de destino em  
  casestandard  
  a saída fecha inesperadamente (ao canalizar a saída  
  para 'cabeça', por exemplo)  
    destination.on("erro", err => process.exit());
```

*É improvável que os pedaços que lemos terminem com uma nova
linha, então cada um provavelmente tem uma linha parcial no final. Acompanhe
isso aqui*

```
  let incompleteLine = "";
```

*Use um loop for/await para ler de forma assíncrona
chunksfrom o fluxo de entrada*

```
  for await (deixe o pedaço da fonte) {  
    Divilda o final do último pedaço mais este em  
    Linhas  
    let lines = (incompleteLine +  
      chunk).split("\n");// A última linha é  
      incompleteLine = lines.pop();// Agora  
      percorra as linhas e escreva as correspondências  
    para o destino  
    for(let linha de linhas) {  
      if (padrão.teste(linha)) {  
        destination.write(linha + "\n", codificação);}}}  
Por  
fim, verifique se há uma correspondência em qualquer  
text.if à direita (pattern.test(incompleteLine)) {
```

```
destination.write(incompleteLine + "\n", codificação);}}
```

*let padrão = new RegExp(process.argv[2]); Obtenha um RegExp
na linha de comando.*

```
grep(process.stdin, process.stdout, pattern) // Chama a
função async grep().
    .catch(err => {                                Manejar
exceções assíncronas.
    console.error(err);
    process.exit();});
```

16.5.3 Gravando em fluxos e manipulando contrapressão

A função `async grep()` no exemplo de código anterior demonstrou como usar um fluxo legível como um iterador assíncrono, mas também demonstrou que você pode gravar dados em um fluxo gravável simplesmente passando-o para o método `write()`. O método `write()` usa um buffer ou string como o primeiro argumento. (Os fluxos de objetos esperam outros tipos de objetos, mas estão além do escopo deste capítulo.) Se você passar um buffer, os bytes desse buffer serão gravados diretamente. Se você passar uma string, ela será codificada em um buffer de bytes antes de ser gravada. Os fluxos graváveis têm uma codificação padrão que é usada quando você passa uma string como o único argumento para escrever `()`. A `defaultencoding` normalmente é `"utf8"`, mas você pode defini-la explicitamente chamando `setDefaultEncoding()` no fluxo gravável. Alternativamente, quando você passa uma string como o primeiro argumento para `write()`, você pode passar um nome de codificação como o segundo argumento.

`write()` opcionalmente recebe uma função de retorno de chamada como seu terceiro argumento. Isso será invocado quando os dados tiverem sido realmente gravados e não estiverem mais no buffer interno do fluxo gravável. (Esse retorno de chamada também pode ser invocado se ocorrer um erro, mas isso não é garantido. Você deve registrar um manipulador de eventos de "erro" no fluxo gravável para detectar

erros.)

O método write() tem um valor de retorno muito importante. Quando você chama write() em um fluxo, ele sempre aceitará e armazenará em buffer o pedaço de dados que você passou. Em seguida, ele retornará true se o buffer interno ainda não estiver cheio. Ou, se o buffer agora estiver cheio ou cheio demais, ele retornará false. Esse valor de retorno é consultivo e você pode ignorá-lo — os fluxos graváveis aumentarão seu buffer interno o quanto for necessário se você continuar chamando write(). Mas lembre-se de que o motivo para usar uma API de streaming em primeiro lugar é evitar o custo de manter muitos dados na memória de uma só vez.

Um valor de retorno de false do método write() é uma forma de decontrapressão: uma mensagem do fluxo de que você gravou dados mais rapidamente do que pode ser tratado. A resposta adequada a esse tipo de contrapressão é parar de chamar write() até que o fluxo emita um evento de "drenagem", sinalizando que há mais uma vez espaço no buffer. Aqui, por exemplo, está uma função que grava em um fluxo e, em seguida, invoca um retorno de chamada quando não há problema em gravar mais dados no fluxo:

```
function write(stream, chunk, callback) {  
    Escreva o pedaço especificado no streamlet  
    especificado hasMoreRoom = stream.write(chunk);  
  
    Verifique o valor de retorno do método write(): if  
    (hasMoreRoom) { // Se ele retornou true, então  
        setImmediate(retorno de chamada); invocar retorno de chamada  
        Assincronamente.  
    } else { Se ele retornou  
        false, então  
            stream.once("drenar", retorno de chamada); invocar retorno de chamada em  
            evento de drenagem.  
    }  
}
```

```
}}
```

O fato de que às vezes não há problema em chamar write() várias vezes seguidas e às vezes você tem que esperar por um evento entre writestorna os algoritmos estranhos. Esta é uma das razões pelas quais usar o método pipe() é tão atraente: quando você usa pipe(), o Nodelida com a contrapressão para você automaticamente.

Se você estiver usando await e async em seu programa e estiver tratando fluxos legíveis como iteradores assíncronos, é simples implementar uma versão baseada em Promise da função do utilitário write() acima para lidar adequadamente com a contrapressão. Na função grep () assíncrona que acabamos de ver, não lidamos com a contrapressão. A função async copy() no exemplo a seguir demonstra como isso pode ser feito corretamente. Observe que essa função apenas copia partes de um fluxo de origem para um fluxo de destino e chamar copy(source,destination) é muito parecido com callingsource.pipe(destination):

Esta função grava o pedaço especificado no fluxo especificado e// retorna uma promessa que será cumprida quando estiver OK para escrever novamente.// Como retorna uma promessa, ela pode ser usada com await.function write(stream, chunk) {

Escreva o pedaço especificado no streamlet especificado hasMoreRoom = stream.write(chunk);

```
if (hasMoreRoom) {  
    Se buffer for  
    não cheio, retorno  
    return Promise.resolve(null);  
    um já  
    resolvido Objeto Promise  
} else {
```

```
        return new Promise(resolve => {      Caso contrário
    retornar uma promessa que
        stream.once("drenar", resolver); resolve no
    evento de drenagem.
});}}
```

*Copie os dados do fluxo de origem para o fluxo de destino//
respeitando a contrapressão do fluxo de destino.// Isso é
muito parecido com chamar source.pipe(destination).async
function copy(source, destination) {*

*Defina um manipulador de erros no fluxo de destino em
casestandard*

*a saída fecha inesperadamente (ao canalizar a saída
para 'cabeça', por exemplo)*

```
    destination.on("erro", err => process.exit());
```

*Use um loop for/await para ler de forma assíncrona
chunksfrom o fluxo de entrada*

```
    for await (deixe o pedaço da fonte) {
```

*Escreva o pedaço e espere até que haja mais espaço
no buffer.*

```
    await write(destino, pedaço);}}
```

*Copie a entrada padrão para a saída
padrãocopy(process.stdin, process.stdout);*

Antes de concluirmos esta discussão sobre gravação em fluxos, observe novamente que a falha em responder à contrapressão pode fazer com que seu programa use mais memória do que deveria quando o buffer interno de um fluxo gravável transborda e fica cada vez maior. Se você estiver escrevendo um servidor de rede, isso pode ser um problema de segurança explorável remotamente. Suponha que você escreva um servidor HTTP que entregue arquivos pela rede, mas não usou pipe() e não teve tempo para lidar com a contrapressão do método write(). Um invasor pode escrever um

Cliente HTTP que inicia solicitações de arquivos grandes (como imagens), mas nunca lê o corpo da solicitação. Como o cliente não está lendo os dados pela rede e o servidor não está respondendo à contrapressão, os buffers no servidor vão estourar. Com conexões simultâneas suficientes do invasor, isso pode se transformar em um ataque de negação de serviço que torna seu servidor lento ou até mesmo trava.

16.5.4 Lendo fluxos com eventos

Os fluxos legíveis do Node têm dois modos, cada um com sua própria API para leitura. Se você não puder usar pipes ou iteração assíncrona em seu programa, precisará escolher uma dessas duas APIs baseadas em eventos para lidar com fluxos. É importante que você use apenas um ou outro e não misture as duas APIs.

MODO FLUIDO No modo de fluxo, quando os dados legíveis chegam, eles são imediatamente emitidos na forma de um evento de "dados". Para ler de um fluxo neste modo, basta registrar um manipulador de eventos para eventos de "dados" e o fluxo enviará pedaços de dados (buffers ou strings) para você assim que estiverem disponíveis. Observe que não há necessidade de chamar o modo de entrada do método `read()`: você só precisa lidar com eventos "data". Observe que os fluxos recém-criados não começam no modo de fluxo. O registro de um manipulador de eventos "data" alterna um fluxo para o modo de fluxo. Convenientemente, isso significa que um fluxo não emite eventos de "dados" até que você registre o primeiro manipulador de eventos de "dados".

Se você estiver usando o modo de fluxo para ler dados de um fluxo legível, processá-los e gravá-los em um fluxo gravável, talvez seja necessário

lidar com a contrapressão do fluxo gravável. Se o método write() retornar false para indicar que o buffer de gravação está cheio, você poderá chamar pause() no fluxo legível para interromper temporariamente os eventos de dados. Então, quando você obtiver um evento "drain" do fluxo Writable, poderá chamar resume() no fluxo Readable para iniciar os eventos "data" fluindo novamente.

Um fluxo no modo de fluxo emite um evento "end" quando o final do fluxo é atingido. Esse evento indica que nenhum outro evento de "dados" será emitido. E, como acontece com todos os fluxos, um evento de "erro" é emitido se ocorrer um erro.

No início desta seção sobre streams, mostramos uma função nonstreamingcopyFile() e prometemos uma versão melhor por vir. O código a seguir mostra como implementar uma função copyFile() de streaming que usa a API de modo de fluxo e lida com a contrapressão. Isso teria sido mais fácil de implementar com uma chamada pipe(), mas serve aqui como uma demonstração útil dos vários manipuladores de eventos que são usados para coordenar o fluxo de dados de um fluxo para o outro.

```
const fs = require("fs");

Uma função de cópia de arquivo de streaming, usando o "modo de fluxo".// Copia o conteúdo do arquivo de origem nomeado para o arquivo nameddestination.// Em caso de sucesso, invoca o retorno de chamada com um argumento nulo. Onerror,// invoca o retorno de chamada com um objeto Error. function copyFile(sourceFilename, destinationFilename, callback) {

  let input = fs.createReadStream(nome_do_arquivo-fonte);
  let saída = fs.createWriteStream(nome_do_arquivo-destino);
```

```

        input.on("dados", (pedaço) => {           Quando chegarmos novo
dados
            let hasRoom = output.write(pedaço); Escreva-o no
fluxo de saída.
            if (!hasRoom) {                     Se a saída
stream é
fullinput.pause();                         em seguida, pause o
fluxo de entrada.
        });
        input.on("fim",
() => {
    o fim da entrada,                    Quando chegarmos
saída.end();                          Diga a saída
fluxo até o fim.
    });
    input.on("erro", err
=> {
    erro na entrada,                  Se obtivermos um
callback(err);
retorno de chamada com o erro
processo.exit();
    });
}

```

Quando a saída "dá�enastivêr mais cheia,

```

        input.resume();                      Dados de currículo
eventos na entrada
    });
    output.on("erro", err
=> {
    erro na saída,                      Se obtivermos um
callback(err);
retorno de chamada com o erro
processo.exit();
    });
    output.on("finish", ()              quando a saída é
=> {
    totalmente escrito                 chame o
callback(null);
retorno de chamada sem erro.
});}

```

Aqui está um utilitário de linha de comando simples para copiar filelet de = process.argv[2], para = process.argv[3];

```
console.log('Copiando arquivo ${de} para  
${para}...'); copyFile(de, para, err => {  
  se (err) {  
    console.error(err);}  
  else {  
    console.log("feito.");}});
```

MODO PAUSADO Outro modo para fluxos legíveis é o "modo pausado". Este é o tema em que os fluxos começam. Se você nunca registrar um manipulador de eventos "data" e nunca chamar o método pipe(), um fluxo legível permanecerá no modo pausado. No modo pausado, o fluxo não envia dados para você na forma de eventos de "dados". Em vez disso, você extrai dados do fluxo chamando explicitamente seu método read(). Essa não é uma chamada de bloqueio e, se não houver dados disponíveis para leitura no fluxo, ela retornará nula. Como não há uma API síncrona para aguardar dados, a modeAPI pausada também é baseada em eventos. Um fluxo legível no modo pausado emite eventos "legíveis" quando os dados ficam disponíveis para leitura no fluxo. Em resposta, seu código deve chamar o método read() para ler thatdata. Você deve fazer isso em um loop, chamando read() repetidamente até que ele retorne null. É necessário drenar completamente o buffer do fluxo para acionar um novo evento "legível" no futuro. Se você parar de chamar read() enquanto ainda houver dados legíveis, você não obterá outro evento "legível" e seu programa provavelmente travará.

Os fluxos no modo pausado emitem eventos de "fim" e "erro", assim como os fluxos do modo de fluxo. Se você estiver escrevendo um programa que lê dados de um fluxo legível e os grava em um fluxo gravável e, em seguida, pausado

pode não ser uma boa escolha. Para lidar adequadamente com a contrapressão, você só deseja ler quando o fluxo de entrada estiver legível e o fluxo de saída não tiver backup. No modo pausado, isso significa ler e gravar até que read() retorne null ou write() retorne false, e então começar a ler ou escrever novamente em um evento ordrain legível. Isso é deselegante e você pode achar que o modo de fluxo (ou tubos) é mais fácil neste caso.

O código a seguir demonstra como você pode calcular um hash SHA256 para o conteúdo de um arquivo especificado. Ele usa um modo inpaused de fluxo legível para ler o conteúdo de um arquivo em partes e, em seguida, passa cada chunk para o objeto que calcula o hash. (Observe que no Nô 12 e posterior, seria mais simples escrever essa função usando um for/awaitloop.)

```
const fs = require("fs"); const
cripto = require("cripto");

Calcule um hash sha256 do conteúdo do arquivo nomeado e
passe o hash // (como uma string) para a função de retorno
de chamada de erro primeiro.function sha256(nome do
arquivo, retorno de chamada) {

let input = fs.createReadStream(nome do arquivo); O fluxo
de dados.
    let hasher = crypto.createHash("sha256"); Durante
calculando o hash.

        input.on("legível", () => {                Quando há
            Dados prontos para leitura
                deixe pedaço; while(chunk =
                    input.read()) {                  Leia um pedaço e
            se não for nulo,
                hasher.update(pedaço);          Passe para o
            Hasher
```

```

        }
        e continue fazendo looping
até não ser legível
}); input.on("fim", ())
=> {
        No final do
riacho
        let hash = hasher.digest("hex"); calcule o
        hash, callback(null, hash); // e passe-o para
o retorno de chamada.
}); input.on("erro",
        retorno de chamada);
        Em caso de erro, chame
retorno de
chamada}

```

Aqui está um utilitário de linha de comando simples para calcular o hash de um arquivo SHA256(process.argv[2], (err, hash) => { // Passe o nome do arquivo da linha de comando.

```

        se (err) {
                Se obtivermos um
        erro
                console.error(err.toString()); imprima-o como um
        erro.
        } else {
                Caso contrário
                console.log (hash); Imprima o hash
        corda.
});}

```

16.6 Processo, CPU e sistema operacional Detalhes

O objeto `Process` global tem várias propriedades e funções úteis que geralmente se relacionam com o estado do processo Node em execução no momento. Consulte a documentação do Node para obter detalhes completos, mas aqui estão algumas propriedades e funções que você deve conhecer:

<code>process.argv</code>	<i>Uma matriz de linha de comando</i>
<code>process.uptime</code>	<i>A arquitetura da CPU: "x64", para</i>

<i>exemplo.proc</i>	
<i>ess.cwd()</i>	<i>Retorna o funcionamento atual</i>
<i>diretório.proc</i>	
<i>ess.chdir()</i>	<i>Define o funcionamento atual</i>
<i>diretório.process</i>	
<i>.cpuUsage()</i>	<i>Relata o uso da CPU.</i>
<i>process.env</i>	<i>Um objeto do ambiente</i>
<i>variáveis.proc</i>	
<i>ss.execPath</i>	<i>O caminho absoluto do sistema de arquivos para o nó</i>
<i>executable.process.exit()</i>	<i>Encerra o programa.</i>
<i>process.exitCode</i>	<i>Um código inteiro a ser relatado quando o programa sai.</i>
<i>process.getuid()</i>	<i>Retorne o ID do usuário Unix do current user.</i>
<i>process.hrtime().bigint()</i>	<i>// Retorna um timestamp de nanossegundo de "alta resolução".</i>
<i>process.kill()</i>	<i>// Envia um sinal para outro process.</i>
<i>process.memoryUsage()</i>	<i>// Retorna um objeto com memória usedetails.</i>
<i>process.nextTick()</i>	<i>// Como setImmediate(), invoca uma função soon.</i>
<i>process.pid</i>	<i>// O id do processo do currentprocess.</i>
<i>process.ppid</i>	<i>// O processo pai id.</i>
<i>process.platform</i>	<i>// O sistema operacional: "linux", "darwin" ou "win32", por example.</i>
<i>process.resourceUsage()</i>	<i>// Retorna um objeto com detalhes de uso de recursos.</i>
<i>process.setuid()</i>	<i>// Define o usuário atual, por id orname.</i>
<i>process.title</i>	<i>// O nome do processo que aparece em 'ps' listings.</i>
<i>process.umask()</i>	<i>// Define ou retorna as permissões padrão para novos arquivos.</i>
<i>process.uptime()</i>	<i>// Retorna o tempo de atividade do nó em segundos.</i>
<i>process.version</i>	<i>// String da versão do nó.</i>
<i>process.versions</i>	<i>// Strings de versão para as bibliotecasNó depende.</i>

O módulo "os" (que, ao contrário do processo, precisa ser explicitamente

loaded with require()) fornece acesso a detalhes de nível igualmente baixo sobre o computador e o sistema operacional em que o Node está sendo executado. Talvez você nunca precise usar nenhum desses recursos, mas vale a pena saber que o Node os disponibiliza:

```
const os = require("os"); os.arch()// Retorna a arquitetura da CPU. "x64" ou "arm", por exemplo.os.constants// Constantes úteis como  
os.constants.signals.SIGINT.os.cpus()// Dados sobre os núcleos da CPU do sistema, incluindo tempos de uso.os.endianess()// A endianidade nativa da CPU "BE" ou "LE".os.EOL// O terminador de linha nativa do S0: "\n"or "\r\n".os.freemem()// Retorna a quantidade de RAM livre inbytes.os.getPriority()// Retorna a prioridade de agendamento do S0 de um process.os.homedir()// Retorna o diretório inicial do usuário atual.os.hostname()// Retorna o nome do host do computador.os.loadavg()// Retorna as médias de carregamento de 1, 5 e 15 minutos.os.networkInterfaces()// Retorna detalhes sobre a rede disponível.  
connections.os.platform()// Retorna o sistema operacional: "linux", "darwin" ou "win32", por exemplo.os.release()// Retorna o número da versão do OS.os.setPriority()// Tenta definir a prioridade de agendamento para um process.os.tmpdir()// Retorna o padrão temporarydirectory.os.totalmem()// Retorna a quantidade total de RAM inbytes.os.type()// Retorna o sistema operacional: "Linux", "Darwin" ou "Windows_NT", por exemplo,
```

<code>os.uptime()</code>	<i>Retorna o tempo de atividade do sistema em segundos.</i>
<code>userInfo()</code>	<i>Retorna uid, nome de usuário, home e shell do usuário atual.</i>

16.7 Trabalhando com arquivos

O módulo "fs" do Node é uma API abrangente para trabalhar com arquivos e diretórios. É complementado pelo módulo "path", que define funções de utilidade para trabalhar com nomes de arquivos e diretórios. O módulo "fs" contém um punhado de funções de alto nível para facilmente ler, gravar e copiar arquivos. Mas a maioria das funções no módulo são ligações JavaScript de baixo nível para chamadas do sistema Unix (e seus equivalentes no Windows). Se você já trabalhou com chamadas de sistema de arquivos de baixo nível antes (em C ou outras linguagens), a API do Node será familiar para você. Caso contrário, você pode achar que partes da API "fs" são mais interessantes e não intuitivas. A função para excluir um arquivo, por exemplo, é chamada `unlink()`.

O módulo "fs" define uma API grande, principalmente porque geralmente existem várias variantes de cada operação fundamental. Conforme discutido no início do capítulo, a maioria das funções, como `fs.readFile()`, não são bloqueadoras, baseadas em retorno de chamada e assíncronas. Normalmente, porém, cada uma dessas funções tem uma variante de bloqueio síncrona, como `fs.readFileSync()`. No Nô 10 e posterior, muitas dessas funções também têm uma variante assíncrona baseada em promessa, como `fs.promises.readFile()`. A maioria das funções "fs" usa uma string como seu primeiro argumento, especificando o caminho (nome do arquivo mais nomes de diretório opcionais) para o arquivo que deve ser operado. Mas várias dessas funções também suportam uma variante que usa um "arquivo" inteiro

descriptor" como o primeiro argumento em vez de um caminho. Essas variantes têm nomes que começam com a letra "f". Por exemplo, `fs.truncate()` trunca um arquivo especificado por `path` e `fs.ftruncate()` trunca um arquivo especificado pelo descriptor de arquivo. Há um `Promise-based``fs.promises.truncate()` que espera um caminho e outra versão baseada em `Promise` que é implementada como um método de um objeto `FileHandle`. (A classe `FileHandle` é o equivalente a um descriptor de arquivo na API baseada em promessa.) Finalmente, há um punhado de funções no módulo "fs" que têm variantes cujos nomes são prefixados com a letra "l". Essas variantes "l" são como a função base, mas não seguem links simbólicos no sistema de arquivos e, em vez disso, operam diretamente nos próprios links simbólicos.

16.7.1 Caminhos, descritores de arquivo e identificadores de arquivo

Para usar o módulo "fs" para trabalhar com arquivos, primeiro você precisa ser capaz de nomear o arquivo com o qual deseja trabalhar. Os arquivos são mais frequentemente especificados por caminho, o que significa o nome do próprio arquivo, mais a hierarquia de diretórios nos quais o arquivo aparece. Se um caminho for absoluto, significa que os diretórios até a raiz do sistema de arquivos são especificados. Caso contrário, o caminho é relativo e só é significativo em relação a algum outro caminho, geralmente o diretório de trabalho atual. Trabalhar com caminhos pode ser um pouco complicado porque diferentes sistemas operacionais usam caracteres diferentes para separar nomes de diretórios, é fácil dobrar accidentalmente esses caracteres separadores ao concatenar caminhos e porque .. / Os segmentos de caminho do diretório pai precisam de tratamento especial. O módulo "path" do Node e alguns outros recursos importantes do Node:

Alguns caminhos importantes
`process.cwd()// Caminho absoluto do working directory.`
`__filename// Caminho absoluto do arquivo que contém o code.`
`__dirname// Caminho absoluto do diretório que contém __filename.`
`os.homedir()// O diretório pessoal do usuário.`

```
const caminho = require("caminho");  
  
Caminho:$é dependendo do seu sistema operacional
```

O módulo de caminho tem funções de análise
`simpleslet p = "src/pkg/test.js";// Um exemplo de caminhocaminho.basename(p)// => "test.js"`
`caminho.dirname(p)// => ".js"`
`caminho.basename(caminho.dirname(p))// => "src/pkg"`
`caminho.dirname(p)// => "src"`

```
normalize() limpa  
caminhos:path.normalize("a/b/c/.. /d/") => "a/b/d/": alças .. /  
segmentspath.normalize  
( "a/./b" ) => "a/b": tiras "./"  
segmentspath.normalize( "/a//b//") => "/a/b/": remove  
duplicar/
```

`join()` combina segmentos de caminho, adicionando separadores, thennormalizespath.join("src", "pkg", "t.js")// => "src/pkg/t.js"

`resolve()` pega um ou mais segmentos de caminho e retorna um caminho absoluto//. Ele começa com o último argumento e funciona de trás para frente, parando // quando ele construiu um caminho absoluto ou resolvendo
`againstprocess.cwd().path.resolve()// => process.cwd().path.resolve("t.js")// => path.join(process.cwd(), "t.js")path.resolve("/tmp", "t.js")// => "/tmp/t.js"`

```
path.resolve("/a", "/b", "t.js") // => "/b/t.js"
```

Observe que path.normalize() é simplesmente uma função de manipulação de string que não tem acesso ao sistema de arquivos real. As funções thefs.realpath() e fs.realpathSync() executam canonização com reconhecimento do sistema de arquivos: elas resolvem links simbólicos e interpretam nomes de caminhos relativos em relação ao diretório de trabalho atual.

Nos exemplos anteriores, assumimos que o código está sendo executado em um sistema operacional baseado em Unix e path.sep é "/". Se você quiser trabalhar com caminhos no estilo Unix mesmo quando estiver em um sistema Windows, use path.posixem vez de path. E, inversamente, se você quiser trabalhar com Windowspaths mesmo quando estiver em um sistema Unix, path.win32. path posix e path.win32 definem as mesmas propriedades e funções que o próprio path.

Algumas das funções "fs" que abordaremos nas próximas seções esperam um descritor de arquivo em vez de um nome de arquivo. Os descritores de arquivo são inteiros usados como referências no nível do sistema operacional para arquivos "abertos". Você obtém o descritor para um determinado nome chamando a função fs.open() (orfs.openSync()). Os processos só podem ter um número limitado de arquivos abertos ao mesmo tempo, por isso é importante que você chamefs.close() em seus descritores de arquivo quando terminar de usá-los. Você precisa abrir arquivos se quiser usar as funções lowest-levelfs.read() e fs.write() que permitem que você pule dentro de um arquivo, lendo e gravando partes dele em momentos diferentes. Existem outras funções no módulo "fs" que usam descritores de arquivo, mas todas elas têm versões baseadas em nomes, e só faz sentido usar as funções baseadas em descritores se você for abrir o arquivo

para ler ou escrever de qualquer maneira.

Finalmente, na API baseada em Promise definida por `fs.promises`, o equivalente a `fs.open()` é `fs.promises.open()`, que retorna uma Promise que resolve para um objeto `FileHandle`. Esse objeto `FileHandle` serve à mesma finalidade que um descritor de arquivo. Novamente, no entanto, a menos que você precise usar os métodos `read()` e `write()` de nível mais baixo de um `FileHandle`, não há realmente nenhuma razão para criar um. E se você criar um `FileHandle`, lembre-se de chamar seu método `close()` assim que terminar de usá-lo.

16.7.2 Lendo arquivos

O Node permite que você leia o conteúdo do arquivo de uma só vez, por meio de um fluxo ou com a API de baixo nível.

Se seus arquivos forem pequenos ou se o uso e o desempenho da memória não forem a prioridade mais alta, geralmente é mais fácil ler todo o conteúdo de um arquivo com uma única chamada. Você pode fazer isso de forma síncrona, com um retorno de chamada ou com uma promessa. Por padrão, você obterá os bytes do arquivo como `abuffer`, mas se especificar uma codificação, obterá uma cadeia de caracteres decodificada.

```
const fs = require("fs"); let buffer =
  fs.readFileSync("test.data"); //  

Síncrono, retorna buffer
let text =
  fs.readFileSync("data.csv", "utf8"); Síncrono,
retorna cadeia de caracteres

Leia os bytes do arquivo
asynchronously
fs.readFile("test.data", (err, buffer) => {
  se (err) {
```

```
Lidar com o erro aqui} else
{
Os bytes do arquivo estão no buffer}});
```

```
readfs.promises assíncronos
baseados em promessas
    .readFile("data.csv",
    "utf8").then(processFileText
    ).catch(handleReadError);
```

```
Ou use a API Promise com await dentro de uma função assíncrona função
assíncrona processText(filename, encoding="utf8") {
let text = await fs.promises.readFile(nome do
arquivo,codificação);
// ... processe o texto aqui ...}
```

Se você for capaz de processar o conteúdo de um arquivo sequencialmente e não precisar ter todo o conteúdo do arquivo na memória ao mesmo tempo, a leitura de um arquivo por meio de um fluxo pode ser a abordagem mais eficiente. Cobrimos os fluxos extensivamente: aqui está como você pode usar a stream e o método pipe() para gravar o conteúdo de um arquivo na saída padrão:

```
function printFile(nome do arquivo, codificação="utf8") {
fs.createReadStream(nome do
arquivo,codificação).pipe(proces
so.stdout);}
```

Finalmente, se você precisar de controle de baixo nível sobre exatamente quais bytes você lê de um arquivo e quando os lê, você pode abrir um arquivo para obter um descriptor de arquivo e usar fs.read(), fs.readSync(), or fs.promises.read() para ler um número especificado de bytes de um arquivo

local de origem especificado do arquivo em um buffer especificado na posição de destino especificada:

```
const fs = require("fs");

Lendo uma parte específica de um arquivo de
dadosfs.open("data", (err, fd) => {
  se (err) {
    Relatar erro de alguma
    formareturnar;}tente {

Ler bytes de 20 a 420 em um recém-alocado
buffer.
  fs.read(fd, Buffer.alloc(400), 0, 400, 20, (err, n,
b) => {
    err é o erro, se houver.// n é o número de
    bytes realmente lidos// b é o buffer em que
    os bytes foram lidos
  em.
});}fina
lmente {
  Use uma cláusula finally para que sempre
  fs.close(fd); Feche o descritor de arquivo aberto
});
```

A API read() baseada em retorno de chamada é difícil de usar se você precisar ler mais de um bloco de dados de um arquivo. Se você puder usar o API síncrona (ou a API baseada em promessa com await), torna-se fácil ler vários pedaços de um arquivo:

```
const fs = require("fs");

function readData(nome do arquivo) {
  let fd = fs.openSync(nome do
  arquivo); tente {
    Leia o cabeçalho do arquivo
```

```
let cabeçalho = Buffer.alloc(12); Um bufferfs.readSync  
de 12 bytes(fd, cabeçalho, 0, 12, 0);  
  
Verifique o número mágico do arquivolet  
magic = header.readInt32LE(0); if  
(mágica !== 0xDADAFEE) {  
throw new Error("O arquivo é do tipo  
errado");}  
  
Agora obtenha o deslocamento e o comprimento dos dados do  
cabeçalho  
let deslocamento = header.readInt32LE(4);  
let comprimento = header.readInt32LE(8);  
  
E leia esses bytes dos dados do filelet =  
Buffer.alloc(length); fs.readSync(fd, dados,  
0, comprimento, deslocamento); dados de  
retorno;} finalmente {  
  
Sempre feche o arquivo, mesmo que uma exceção seja  
jogado acima  
fs.closeSync(fd);}
```

16.7.3 Gravando arquivos

Escrever arquivos no Node é muito parecido com lê-los, com alguns detalhes extras que você precisa saber. Um desses detalhes é que a maneira como você cria um novo arquivo é simplesmente escrevendo em um nome de arquivo que ainda não existe.

Assim como na leitura, existem três maneiras básicas de gravar arquivos no Node. Se você tiver todo o conteúdo do arquivo em uma string ou buffer, poderá gravar tudo em uma chamada com `fs.writeFile()` (baseado em retorno de chamada), `fs.writeFileSync()` (síncrono) ou

`fs.promises.writeFile()` (baseado em promessa):

```
fs.writeFileSync(caminho.resolve(__dirname, "settings.json"),
    JSON.stringify(configurações));
```

Se os dados que você está gravando no arquivo forem uma string e você quiser usar uma codificação diferente de "utf8", passe a codificação como um terceiro argumento opcional.

As funções relacionadas `fs.appendFile()`, `fs.appendFileSync()` e `fs.promises.appendFile()` são semelhantes, mas quando o arquivo especificado já existe, elas anexam seus dados ao final em vez de substituir o conteúdo do arquivo existente.

Se os dados que você deseja gravar em um arquivo não estiverem todos em uma única parte, ou se não estiverem todos na memória ao mesmo tempo, usar um fluxo gravável é uma boa abordagem, supondo que você planeje gravar os dados do início ao fim sem pular no arquivo:

```
const fs = require("fs");
let saída =
fs.createWriteStream("numbers.txt");
for(let i = 0; i < 100; i++) {
    output.write(`${i}\n`);
}
saída.end();
```

Finalmente, se você deseja gravar dados em um arquivo em vários blocos e deseja controlar a posição exata dentro do arquivo em que cada bloco é gravado, você pode abrir o arquivo com `fs.open()`, `fs.openSync()` ou `fs.promises.open()` e, em seguida, usar o descritor de arquivo resultante com o `fs.write()` ou

`fs.writeFileSync()`. Essas funções vêm em diferentes formas para cadeias de caracteres e buffers. A variante de cadeia de caracteres usa um descritor de arquivo, uma cadeia de caracteres e a posição do arquivo na qual gravar essa cadeia de caracteres (com encoding como um quarto argumento opcional). A variante de buffer usa um descritor de arquivo, um buffer, um deslocamento e um comprimento que especificam um pedaço de dados dentro do buffer e uma posição de arquivo na qual gravar os bytes desse bloco. E se você tiver um array de objetos Buffer que deseja escrever, poderá fazer isso com um único `fs.writev()` ou `fs.writevSync()`. Existem funções de baixo nível semelhantes para escrever buffers e strings usando `fs.promises.open()` e o objeto `FileHandle` que ele produz.

CADEIAS DE CARACTERES DO MODO DE ARQUIVO

Vimos os métodos `fs.open()` e `fs.openSync()` antes ao usar a API de baixo nível para ler arquivos. Nesse caso de uso, foi suficiente apenas passar o nome do arquivo para a função open. No entanto, quando você deseja gravar um arquivo, também deve especificar um segundo argumento de cadeia de caracteres que especifica como você pretende usar o descritor de arquivo. Algumas das cadeias de caracteres de sinalizador disponíveis são as seguintes:

"`w`"

Abra o arquivo para gravação

"`w+`"

Aberto para escrita e leitura

"`wX`"

Aberto para criar um novo arquivo; falhará se o arquivo nomeado já existir

"`wX+`"

Aberto para criação, e também permitir a leitura; falhará se o arquivo nomeado já existir

"`uma`"

Abra o arquivo para anexar; o conteúdo existente não será substituído

"A+"

Aberto para anexação, mas também permite leitura

Se você não passar uma dessas cadeias de caracteres de sinalizador para fs.open() ou fs.openSync(), elas usarão o sinalizador default "r", tornando o descritor de arquivo somente leitura. Observe que também pode ser útil passar esses sinalizadores para outros métodos de gravação de arquivos:

*Escreva em um arquivo em uma chamada, mas anexe a qualquer coisa que já esteja lá.// Isso funciona como
fs.appendFileSync()fs.writeFileSync("messages.log", "hello", { flag:
"a" });*

*Abra um fluxo de gravação, mas gere um erro se o arquivo já existir.// Não queremos sobrescrever algo acidentalmente!// Observe que a opção acima é "flag" e é "flags"
herefs.createWriteStream("messages.log", { flags: "wx" });*

Você pode cortar o final de um arquivo com fs.truncate(), fs.truncateSync() ou fs.promises.truncate(). Essas funções usam um caminho como primeiro argumento e um comprimento como segundo e modificam o arquivo para que ele tenha o comprimento especificado. Se você omitir o comprimento, zero será usado e o arquivo ficará vazio. Apesar do nome dessas funções, elas também podem ser usadas para estender um arquivo: se você especificar um comprimento maior que o tamanho atual do arquivo, o arquivo será estendido com zero bytes para o novo tamanho. Se você já abriu o arquivo que deseja modificar, pode usar ftruncate() ou ftruncateSync() com o descritor de arquivo ou FileHandle.

As várias funções de gravação de arquivos descritas aqui retornam ou invocam seu retorno de chamada ou resolvem sua promessa quando os dados foram "gravados" no sentido de que o Node os entregou ao sistema operacional. Mas isso não significa necessariamente que os dados já foram gravados em armazenamento persistente: pelo menos alguns de seus dados ainda podem estar armazenados em buffer em algum lugar do sistema operacional ou em um driver de dispositivo esperando para ser

gravado em disco. Se você chamar `fs.writeFileSync()` para gravar de forma síncrona alguns dados em um arquivo e se houver uma queda de energia imediatamente após o retorno da função, você ainda poderá perder dados. Se você quiser forçar seu `dataout` para o disco para ter certeza de que ele foi salvo com segurança, use `fs.fsync()` ou `fs.fsyncSync()`. Essas funções funcionam apenas com descritores de arquivo: não há versão baseada em caminho.

16.7.4 Operações de arquivo

A discussão anterior das classes de fluxo do Node incluiu dois exemplos de funções `copyFile()`. Estes não são utilitários práticos que você realmente usaria porque o módulo "fs" define seu próprio método `fs.copyFile()` (e também `fs.writeFileSync()` e `fs.promises.copyFile()`, é claro).

Essas funções usam o nome do arquivo original e o nome da cópia como seus dois primeiros argumentos. Eles podem ser especificados como strings ou como objetos URL ou Buffer. Um terceiro argumento opcional é um inteiro cujos bits especificam sinalizadores que controlam os detalhes da operação de cópia. E para o `fs.copyFile()` baseado em retorno de chamada, o argumento final é uma função de retorno de chamada que será chamada sem argumentos quando a cópia estiver concluída ou que será chamada com um argumento de erro se algo falhar. Veja a seguir alguns exemplos:

Arquivo síncrono básico

```
copy.fs.writeFileSync("ch15.txt", "ch15.bak");
```

O argumento `COPYFILE_EXCL` copia apenas se o novo arquivo ainda não existir//. Isso impede que as cópias sobrecrevam os arquivos existentes.

```
fs.copyFile("ch15.txt", "ch16.txt", fs.constants.COPYFILE_EXCL, err => {
```

Esse retorno de chamada será chamado quando terminar. Em caso de erro, errnão será nulo.});

Este código demonstra a versão baseada em Promise da função copyFile.// Dois sinalizadores são combinados com o operador OR bit a bit |. As bandeiras significam que os arquivos existentes não serão substituídos e que, se o sistema de arquivos suportar//, a cópia será um clone copy-on-write do arquivo original, o que significa// que nenhum espaço de armazenamento adicional será necessário até que o original// ou a cópia seja modificada. fs.promises.copyFile("Dados importantes",

```
'Dados importantes ${new
Data().toISOString()}"'
    fs.constants.COPYFILE_EXCL |
fs.constants.COPYFILE_FICLONE)
    .then(() => {
        console.log("Backup
concluído");});. catch(err => {
    console.error("Falha no backup",
err);});
```

A função fs.rename() (junto com as variantes síncronas usuais baseadas em andPromise) move e/ou renomeia um arquivo. Chame-o com o caminho atual para o arquivo e o novo caminho desejado para o arquivo. Não há argumento de sinalização, mas a versão baseada em retorno de chamada recebe um retorno de chamada como o terceiro argumento:

```
fs.renameSync("ch15.bak", "backups/ch15.bak");
```

Observe que não há nenhum sinalizador para impedir que a renomeação substitua um arquivo existente. Lembre-se também de que os arquivos só podem ser renomeados dentro de um sistema de arquivos.

As funções `fs.link()` e `fs.symlink()` e suas variantes têm as mesmas assinaturas que `fs.rename()` e se comportam como `fs.copyFile()`, exceto que criam links físicos e simbólicos, respectivamente, em vez de criar uma cópia.

Finalmente, `fs.unlink()`, `fs.unlinkSync()` e `fs.promises.unlink()` são as funções do Node para excluir um arquivo. (A nomenclatura não intuitiva é herdada do Unix, onde excluir um arquivo é basicamente o oposto de criar um link físico para ele.) Chame esta função com a cadeia de caracteres, buffer ou caminho de URL para o arquivo a ser excluído e passe um retorno de chamada se você estiver usando a versão baseada em retorno de chamada:

```
fs.unlinkSync("backups/ch15.bak");
```

16.7.5 Metadados de arquivo

As funções `fs.stat()`, `fs.statSync()` e `fs.promises.stat()` permitem que você obtenha metadados para um arquivo ou diretório especificado. Por exemplo:

```
const fs = require("fs"); let stats =  
fs.statSync("livro/ch15.md"); stats.isFile()// => true: este  
é um filestats.isDirectory()// => false: não é um  
directorystats.size// tamanho do arquivo em  
bytesstats.atime// tempo de acesso: Data em que foi escrito  
pela última vezstats.mtime// hora de modificação: Data em  
que foi escrito pela última vezstats.uid// o ID do usuário  
do ownerstats.gid do arquivo// o id do grupo do  
ownerstats.mode.toString(8) do arquivo // as permissões do  
arquivo, como uma octalstring
```

O objeto Stats retornado contém outras propriedades e métodos mais obscuros, mas esse código demonstra aqueles que você provavelmente usará.

`fs.lstat()` e suas variantes funcionam exatamente como `fs.stat()`, exceto quese o arquivo especificado for um link simbólico, o Node retornará metadados para o próprio link em vez de seguir o link.

Se você abriu um arquivo para produzir um descritor de arquivo ou um objeto `FileHandle`, então você pode usar `fs.fstat()` ou suas variantes para obter metadatainformation para o arquivo aberto sem ter que especificar o nome do arquivo novamente.

Além de consultar metadados com `fs.stat()` e todas as suas variantes, também existem funções para alterar metadados.

`fs.chmod()`, `fs.lchmod()` e `fs.fchmod()` (junto com versões síncronas e baseadas em promessas) definem o "modo" ou as permissões de um arquivo ou diretório. Os valores de modo são inteiros em que cada bit tem um significado específico e são mais fáceis de pensar em octalnotação. Por exemplo, para tornar um arquivo somente leitura para seu proprietário e inacessível para todos os outros, use `0o400`:

```
fs.chmodSync("ch15.md", 0o400); Não o exclua  
acidentalmente!
```

`fs.chown()`, `fs.lchown()` e `fs.fchown()` (junto com versões síncronas e baseadas em promessas) definem o proprietário e o grupo (asIDs) para um arquivo ou diretório. (Isso é importante porque interage com as permissões de arquivo definidas por `fs.chmod()`.)

Finalmente, você pode definir o tempo de acesso e o tempo de modificação de um arquivo ou diretório com `fs.utimes()` e `fs.futimes()` e suas variantes.

16.7.6 Trabalhando com diretórios

Para criar um novo diretório no Node, use `fs.mkdir()`, `fs.mkdirSync()` ou `fs.promises.mkdir()`. O primeiro argumento é o caminho do diretório a ser criado. O segundo argumento opcional pode ser um número inteiro que especifica o modo (bits de permissões) para o novo diretório. Ou você pode passar um objeto com modo opcional e propriedades recursivas. Se recursivo for verdadeiro, então esta função criará todos os diretórios no caminho que ainda não existem:

```
Certifique-se de que dist/ e dist/lib/  
existam.fs.mkdirSync("dist/lib", { recursive: true });
```

`fs.mkdtemp()` e suas variantes pegam um prefixo de caminho que você fornece, anexam alguns caracteres aleatórios a ele (isso é importante para a segurança), criam um diretório com esse nome e retornam (ou passam para um retorno de chamada) o caminho do diretório para você.

Para excluir um diretório, use `fs.rmdir()` ou uma de suas variantes. Observe que os diretórios devem estar vazios antes de serem excluídos:

```
Crie um diretório temporário aleatório e obtenha seu  
caminho, então// exclua-o quando terminarmos tempDirPath;  
tente {
```

```
tempDirPath =  
fs.mkdtempSync(path.join(os.tmpdir(), "d"));
```

```
Faça algo com o diretório aqui} finalmente
{
Exclua o diretório temporário quando terminarmos com
itfs.rmdirSync(tempDirPath);}
```

O módulo "fs" fornece duas APIs distintas para listar o conteúdo de um diretório. Primeiro, fs.readdir(), fs.readdirSync(), efs.promises.readdir() lêem todo o diretório de uma só vez e fornecem uma matriz de strings ou uma matriz de objetos Diren que especificam os nomes e tipos (arquivo ou diretório) de cada item. Os nomes de arquivo retornados por essas funções são apenas o nome local do arquivo, não o caminho inteiro. Aqui estão alguns exemplos:

```
let tempFiles = fs.readdirSync("/tmp"); Retorna uma matriz
de cordas

Use a API baseada em Promise para obter um array Diren e, em
seguida, imprima os caminhos dos
subdiretórios
fs.promises.readdir("/tmp", {withFileTypes: true})
  .then(entradas => {
    entradas.filtro(entrada => entrada.éDiretório())
      .map(entry => entry.name).forEach(nome =>
        console.log(path.join("/tmp/",
        nome)));
  }).catch(console.erro
  r);
```

Se você prevê a necessidade de listar diretórios que podem ter milhares de entradas, você pode preferir a abordagem de streaming offs opendir() e suas variantes. Essas funções retornam um objeto Dir que representa o diretório especificado. Você pode usar os métodos read() ou readSync() do objeto Dir para ler um Diren por vez. Se você passar uma função de retorno de chamada para read(), ela chamará o retorno de chamada.

E se você omitir o argumento de retorno de chamada, ele retornará uma promessa. Quando não houver mais entradas de diretório, você obterá null em vez de um objeto Direntr.

A maneira mais fácil de usar objetos Dir é como iteradores assíncronos com loop afor/await. Aqui, por exemplo, está uma função que usa a API de streaming para listar entradas de diretório, chama stat() em cada entrada e imprime nomes e tamanhos de arquivos e diretórios:

```
const fs = require("fs"); const
caminho = require("caminho");

função assíncrona listDirectory(dirpath) {
  let dir = await fs.promises.opendir(dirpath);
  for await (deixe a entrada de dir) {
    deixe o nome = entry.name;
    if (entry.isDirectory()) {
      nome += "/"; Adicionar uma barra à direita para
Subdiretórios
      }let stats = await
      fs.promises.stat(path.join(dirpath,
      nome));
    let tamanho = stats.size;
    console.log(String(size).padStart(10), nome);}}
```

16.8 Clientes e servidores HTTP

Os módulos "http", "https" e "http2" do Node são implementações completas, mas de nível relativamente baixo, dos protocolos HTTP. Eles definem APIs abrangentes para implementar clientes e servidores HTTP. Como as APIs são de nível relativamente baixo, não há espaço para

este capítulo para cobrir todos os recursos. Mas os exemplos a seguir demonstram como escrever clientes e servidores básicos.

A maneira mais simples de fazer uma solicitação HTTP GET básica é com `http.get()` ou `https.get()`. O primeiro argumento para essas funções é a URL a ser buscada. (Se for um URL `http://`, você deve usar o módulo "http" e, se for um URL `https://`, você deve usar o módulo "https".) O segundo argumento é um retorno de chamada que será invocado com um objeto `IncomingMessage` quando a resposta do servidor começar a chegar. Quando o retorno de chamada é chamado, o status HTTP e os cabeçalhos estão disponíveis, mas o corpo pode ainda não estar pronto. O objeto `IncomingMessage` é um fluxo legível e você pode usar as técnicas demonstradas anteriormente neste capítulo para ler o corpo da resposta dele.

A função `getJSON()` no final do §13.2.6 usou a função `http.get()` como parte de uma demonstração do construtor `Promise()`. Agora que você conhece os fluxos do Node e o modelo de programação do Node de forma mais geral, vale a pena revisitar esse exemplo para ver como `http.get()` é usado.

`http.get()` e `https.get()` são variantes ligeiramente simplificadas das funções mais gerais `http.request()` e `https.request()`. A função `postJSON()` a seguir demonstra como usar `https.request()` para fazer uma solicitação HTTPS POST que inclui um corpo de solicitação JSON. Como a função `getJSON()` do Capítulo 13, ela espera uma resposta JSON e retorna uma Promise `thatfulfilleds` para a versão analisada dessa resposta:

```

const https = require("https");

/*
 * Converta o objeto body em uma string JSON e, em seguida,
 * HTTPS POSTit no
 * ponto de extremidade da API especificado no host
 * especificado. Quando a resposta chega,
 * analise o corpo da resposta como JSON e resolva o
 * returnedPromise com
 * esse valor analisado.*/
function postJSON(host,
endpoint, body, port, username,password) {

    Retorne um objeto Promise imediatamente e, em seguida, chame
o resolveor reject
quando a solicitação HTTPS for bem-sucedida ou
falhar.return new Promise((resolve, reject) => {
    Converta o objeto body em uma stringlet
    bodyText = JSON.stringify(body);

    Configure o requestlet HTTPS
    requestOptions = {
        método: "POST",           Ou "GET", "PUT",
        "DELETE", etc.
        anfitrião: anfitrião, O host ao qual se conectar
        path: endpoint// A URL pathheaders: {}//
        cabeçalhos HTTP para o

    pedir
        "Tipo de conteúdo": "application/json", "Content-
        Length": Buffer.byteLength(bodyText)}};

    if (porta) {                               Se uma porta for
especificado
        requestOptions.port = porta; usá-lo para o
pedir.
        } // Se as credenciais forem especificadas, adicione
        uma Autorização
cabeçalho.
        if (nome de usuário e senha) {
            requestOptions.auth = '${nome de
            usuário}:${senha}';}
}

```

```
Agora crie a solicitação com base na configuração
objeto
    let pedido = https.request(requestOptions);

Escreva o corpo da solicitação POST e termine o
pedir.
    request.write(bodyText)
    ; request.end();

Erros de falha na solicitação (como nenhuma rede
conexão)
    request.on("erro", e => rejeitar(e));

Manipule a resposta quando ela começar a
chegar.request.on("response", response => {
    if (response.statusCode !== 200) {
        reject(new Error('Status HTTP
${response.statusCode}'));
    }
Não nos importamos com o corpo de resposta em
neste caso, mas
não queremos que fique por aqui em um
buffer em algum lugar, então
colocamos o fluxo no modo de fluxo
sem registro
um manipulador de "dados" para que o corpo seja
Descartado.
    resposta.resume();
    retornar;}

Queremos texto, não bytes. Estamos assumindo que o
o texto será
Formatado em JSON, mas não está se preocupando em verificar
o
Cabeçalho do tipo de
conteúdo.response.setEncoding("utf8");

O Node não tem um analisador JSON de streaming, portanto,
lemos o
corpo inteiro da resposta em uma string.let
corpo = ""; response.on("dados", chunk => { body
+= chunk; });


```

E agora lidar com a resposta quando estiver completar.

```
        response.on("end", () => {           Quando o
resposta é feita,
            tente {
analisá-lo como JSON
                resolve(JSON.parse(corpo)); e
Resolva o resultado.
            } catch(e) {           Ou, se
qualquer coisa dá errado,
                rejeitar(e);       rejeitar
com o erro
});});});});}
```

Além de fazer solicitações HTTP e HTTPS, os módulos "http" e "https" também permitem que você escreva servidores que respondam a essas solicitações. A abordagem básica é a seguinte:

Crie um novo objeto Server. Chame seu método listen() para começar a escutar solicitações em uma porta especificada. Registre um manipulador de eventos para eventos de "solicitação", use thathandler para ler a solicitação do cliente (particularmente a propriedade request.url) e escreva sua resposta. O código a seguir cria um servidor HTTP simples que atende a arquivos estáticos do sistema de arquivos local e também implementa um endpoint de depuração que responde à solicitação de um cliente ecoando essa solicitação.

Este é um servidor HTTP estático simples que serve arquivos de um diretório especificado//. Ele também implementa um /test/mirror especial

```
endpoint que// ecoa a solicitação recebida, o que pode ser útil ao depurar clientes.const http = require("http");// Use "https" se você tiver acertificateconst url = require("url");// Para analisar URLsconst path = require("path");// Para manipular caminhos do sistema de arquivosconst fs = require("fs");// Para ler arquivos
```

*Serve arquivos do diretório raiz especificado por meio de um servidor HTTP que// escuta na porta especificada.*function serve(rootDirectory, port) {

```
    let server = novo http. Servidor(); Criar um novo HTTP servidor
    server.listen(porta); Ouça no porta especificada
    console.log("Escuta na porta", porta);
```

*Quando as solicitações chegarem, trate-as com esta função.*server.on("request", (request, response) => {
 *Obtenha a parte do caminho da URL da solicitação, ignorando// quaisquer parâmetros de consulta anexados a ela.*let endpoint = url.parse(request.url).pathname;

Se a solicitação for para "/test/mirror", envie de volta o pedido verbatim. Útil quando você precisa ver a solicitação cabeçalhos e corpo.
if (endpoint === "/test/mirror") {
 *Defina a resposta header*response.setHeader("Content-Type", "text/plain;
charset=UTF-8");

Especifique o status da resposta
code`response.writeHead(200); // 200 OK`

Comece o corpo da resposta com
request`response.write('${request.method} ${request.url}
HTTP/${{
 request.httpVersion}\r\n');`

```
Saída da solicitação
headerslet headers =
request.rawHeaders; for(let i = 0; i <
headers.length; i += 2) {
    response.write('${cabeçalhos[i]}:
${cabeçalhos[i+1]}\r\n');
}
```

*Finalize os cabeçalhos com um
line*

*Agora precisamos copiar qualquer corpo de solicitação para o
corpo da resposta*

*Como ambos são fluxos, podemos usar um
piperequest.pipe(response);} Caso contrário, sirva um
arquivo do diretório local.else {*

*Mapeie o ponto de extremidade para um arquivo no local
sistema de arquivos*

*let nome do arquivo = endpoint.substring(1); tira
entrelinha/*

*Não permita ".. /" no caminho porque
Seja uma segurança
para servir qualquer coisa fora da raiz
diretório.*

*filename = filename.replace(/\.\.\.\//g, ""); // Agora
converte de relativo para absoluto
filename = path.resolve(rootDirectory, filename);*

*Agora adivinhe o tipo de conteúdo do arquivo de tipo com
na
prorrogação*

*deixe tipo; switch(caminho.extname(nome do
arquivo)) {
maiúsculas e minúsculas ".html":maiúsculas e minúsculas
".htm": tipo = "texto/html"; quebrar; maiúsculas e
minúsculas ".js":type = "texto/javascript"; quebrar;
caso ".css": tipo = "texto/css"; quebrar; caso ".png":
tipo = "imagem/png"; quebrar; caso ".txt": tipo =
"texto/simples"; quebrar; padrão:tipo =
"aplicativo/fluxo de octeto";
quebrar;
}*

```

        let stream = fs.createReadStream(nome do
            arquivo); stream.once("readable", () => {// Se
            o fluxo se tornar legível, defina
            o
            Content-Type e um status 200 OK.
            Em seguida, canalize o
            fluxo do leitor de arquivos para a resposta. O
            tubulação vai
            chamar automaticamente response.end() quando o
            o fluxo termina.
            response.setHeader("Tipo-Conteúdo", tipo);
            resposta.writeHead(200);
            stream.pipe(resposta);});

            stream.on("erro", (err) => {
            Em vez disso, se recebermos um erro ao tentar abrir
            o fluxo
            então o arquivo provavelmente não existe ou
            não é legível.
            Enviar uma resposta de texto sem formatação 404 Not
            com o
            Found
            error message.response.setHeader("Tipo
            de conteúdo",
            "texto/plain; charset=UTF-8");
            resposta.writeHead(404);
            resposta.end(err.message);});});});

```

*Quando formos invocados a partir da linha de comando, chame
o serve() function serve(process.argv[2] || "/tmp",
parseInt(process.argv[3]) || 8000);*

Os módulos integrados do Node são tudo o que você precisa para escrever servidores HTTP e HTTPS simples. Observe, no entanto, que os servidores de produção normalmente não são construídos diretamente sobre esses módulos. Em vez disso, a maioria dos servidores não triviais é implementada usando bibliotecas externas, como o Express

framework — que fornecem "middleware" e outros utilitários de nível superior que os desenvolvedores da Web de back-end esperam.

16.9 Servidores e clientes de rede não HTTP

Servidores e clientes da Web tornaram-se tão onipresentes que é fácil esquecer que é possível escrever clientes e servidores que não usam HTTP. Embora o Node tenha a reputação de ser um bom ambiente para escrever servidores web, o Node também tem suporte total para escrever outros tipos de servidores e clientes de rede.

Se você se sente confortável trabalhando com fluxos, a rede é relativamente simples, porque os soquetes de rede são simplesmente um tipo de fluxo Duplex. O módulo "net" define as classes Server e Socket. Para criar um servidor, chame `net.createServer()` e, em seguida, chame o método `listen()` do objeto resultante para informar ao servidor em qual porta escutar as conexões. O objeto Server gerará eventos de "conexão" quando um cliente se conectar nessa porta e o valor passado para o `eventlistener` será um objeto Socket. O objeto Socket é um fluxo Duplex e você pode usá-lo para ler dados do cliente e gravar dados no cliente. Chame `end()` no soquete para desconectar.

Escrever um cliente é ainda mais fácil: passe um número de porta e um nome de host para `net.createConnection()` para criar um soquete para se comunicar com qualquer servidor que esteja sendo executado nesse host e escutando nessa porta. Em seguida, use esse soquete para ler e gravar dados de e para o servidor.

O código a seguir demonstra como escrever um servidor com a "rede"

módulo. Quando o cliente se conecta, o servidor conta uma piada toc-toc:

Um servidor TCP que fornece brincadeiras interativas na porta 6789.// (Por que seis tem medo de sete? Porque sete comeram nove!)const net = require("net"); const readline = require("readline");

Crie um objeto Server e comece a ouvir connectionslet server = net.createServer(); server.listen(6789, () => console.log("Entregando risadas na porta 6789"));

Quando um cliente se conectar, diga a ele uma piada toc-toc.server.on("connection", socket => { tellJoke(soquete) .then(() => socket.end())// Quando a piada terminar, feche o soquete.. catch((err) => { console.error(err); // Registre todos os erros que acontecer socket.end(); // mas ainda assim fechar o soquete!});});

Estas são todas as piadas que conhecemos.const piadas = { "Boo": "Não chore... é só uma piada!", "Alface": "Deixe-nos entrar! Está congelando aqui!", "Uma velhinha": "Uau, eu não sabia que você podia!"};

Execute interativamente uma piada toc-toc sobre este soquete, sem bloquear.async function tellJoke(socket) {

*Escolha uma das piadas em randomlet randomElement = a => a[Math.floor(Math.random() *a.length)];*

```
let quem = randomElement(Object.keys(piadas));
let punchline = piadas [quem];
```

Use o módulo readline para ler a entrada do usuário uma linha por vez.

```
let leitorLine= readline.createInterface({  
    entrada: soquete,  
    saída: soquete,  
    prompt: ">> "});
```

Uma função de utilitário para gerar uma linha de texto para o cliente

```
e, em seguida, (por padrão) exibir um  
prompt.function output(text, prompt=true) {  
    socket.write('${text}\r\n'); if  
(prompt) lineReader.prompt();}
```

As piadas toc-toc têm uma estrutura de chamada e resposta.// Esperamos uma entrada diferente do usuário em diferentes estágios e tomar medidas diferentes quando obtivermos essa entrada em diferentes estágios.

```
deixe estágio = 0;
```

Comece a piada toc-toc da maneira tradicional.output("Toc toc!");

Agora leia as falas de forma assíncrona do cliente até que a piada seja feita.

```
for await (let inputLine de lineReader) {  
    if (estágio === 0) {  
        if (inputLine.toLowerCase() === "quem está aí?") {  
            Se o usuário der a resposta correta em  
estágio 0  
                em seguida, conte a primeira parte da piada e  
Vá para o estágio 1.  
                saída (quem);  
                estágio = 1;}  
            else{  
                Caso contrário, ensine o usuário a fazer knock-  
bater piadas.  
                output('Por favor, digite "Quem está  
aí?".');
```

```
    } else if (estágio === 1) {
        if (inputLine.toLowerCase() ===
            '${who.toLowerCase()} quem?') {
                Se a resposta do usuário estiver correta no estágio
1, então
                entregar a piada e retornar desde o
A piada acabou.
                saída('${punchline}', falso);
            retornar;} else {

Faça o usuário jogar junto.output('Por favor, digite
"${who} who?". `);}}}}
```

Servidores simples baseados em texto como esse normalmente não precisam de um customclient. Se o utilitário nc ("netcat") estiver instalado em seu sistema, você poderá usá-lo para se comunicar com este servidor da seguinte maneira:

```
$ nc localhost 6789Toc toc!>> Quem
está aí? Uma velhinha>> Uma
velhinha quem? Uau, eu não sabia
que você poderia yodel!
```

Por outro lado, escrever um cliente personalizado para o servidor de piadas é fácil emNó. Nós apenas nos conectamos ao servidor, então canalizamos a saída do servidor para stdout e canalizamos stdin para a entrada do servidor:

```
Conecte-se à porta joke (6789) no servidor nomeado na linha
de comandolet socket =
require("net").createConnection(6789,process.argv[2]);
socket.pipe(process.stdout);// Dados do pipe do soquete para
stdoutprocess.stdin.pipe(socket);// Dados do pipe de
```

```
stdin para o socketsocket.on("close", () =>
process.exit()); Saia quando o soquete fechar.
```

Além de suportar servidores baseados em TCP, o módulo "net" do Node também suporta comunicação entre processos em "soquetes de domínio Unix" que são identificados por um caminho do sistema de arquivos em vez de um número de porta. Não vamos cobrir esse tipo de soquete neste capítulo, mas a documentação do Nó tem detalhes. Outros recursos do Node que não temos espaço para cobrir aqui incluem o módulo "dgram" para clientes e servidores baseados em UDP e o módulo "tls" que está para "net" como "https" está para "http". O tls. Servidor e tls. As classes TLSSocket permitem a criação de servidores TCP (como o servidor de piada toc-toc) que usam conexões criptografadas por SSL como os servidores HTTPS.

16.10 Trabalhando com processos filhos

Além de escrever servidores altamente simultâneos, o Node também funciona bem para escrever scripts que executam outros programas. No Node, o módulo "child_process" define várias funções para executar outros programas como processos filhos. Esta seção demonstra algumas dessas funções, começando com a mais simples e passando para a mais complicada.

16.10.1 execSync() e execFileSync()

A maneira mais fácil de executar outro programa é withchild_process.execSync(). Esta função usa o comando para executar como seu primeiro argumento. Ele cria um processo filho, executa um shell nesse processo e usa o shell para executar o comando que você passou. Então ele

até que o comando (e o shell) saiam. Se o comando for encerrado com um erro, execSync() lançará uma exceção. Caso contrário, execSync() retorna qualquer saída que o comando grava em itsstdout stream. Por padrão, esse valor retornado é um buffer, mas você pode especificar uma codificação em um segundo argumento opcional para obter uma cadeia de caracteres. Se o comando gravar qualquer saída em stderr, essa saída será passada para o fluxo stderr do processo pai.

Assim, por exemplo, se você estiver escrevendo um script e o desempenho não for uma preocupação, você pode usar child_process.execSync() para listar um diretório com um comando de shell Unix familiar em vez de usar a função thefs.readdirSync():

```
const child_process = require("child_process"); let
listagem = child_process.execSync("ls -l web/*.html",
{encoding: "utf8"});
```

O fato de execSync() invocar um shell Unix completo significa que a string que você passa para ele pode incluir vários comandos separados por ponto e vírgula e pode aproveitar os recursos do shell, como wildcards de nome de arquivo, pipes e redirecionamento de saída. Isso também significa que você deve ter cuidado para nunca passar um comando para execSync() se qualquer parte desse comando for entrada do usuário ou vier de uma fonte não confiável semelhante. A sintaxe complexa dos comandos shell pode ser facilmente subvertida para permitir que um invasor execute código arbitrário.

Se você não precisar dos recursos de um shell, poderá evitar a sobrecarga de iniciar um shell usando child_process.execFileSync(). Essa função executa um programa diretamente, sem invocar um shell. Mas como nenhum shell está envolvido, ele não pode analisar uma linha de comando e você

deve passar o executável como o primeiro argumento e uma matriz de argumentos de linha de comando como o segundo argumento:

```
let listagem = child_process.execFileSync("ls", ["-l", "web/"],  
                                         {codificação: "utf8"});
```

OPÇÕES DE PROCESSO FILHO

`execSync()` e muitas das outras funções `child_process` têm um segundo ou terceiro argumento opcional que especifica detalhes adicionais sobre como o processo filho deve ser executado. A encoding propriedade desse objeto foi usada anteriormente para especificar que gostaríamos que a saída do comando fosse entregue como uma string em vez de como um buffer. Outras propriedades importantes que você pode especificar incluem o seguinte (observe que nem todas as opções estão disponíveis para todas as funções de processo filho):

- `cwd` especifica o diretório de trabalho para o processo filho. Se você omitir isso, o `childprocess` herdará o valor de `process.cwd()`.
- `env` especifica as variáveis de ambiente às quais o processo filho terá acesso. Por padrão, os processos filhos simplesmente herdam `process.env`, mas você pode especificar um objeto diferente se desejar.
- `Input` especifica uma cadeia de caracteres ou buffer de dados de entrada que deve ser usado como a entrada padrão para o processo filho. Essa opção só está disponível para as funções síncronas que não retornam um objeto `ChildProcess`.
- `maxBuffer` especifica o número máximo de bytes de saída que serão coletados pelas funções `exec`. (Não se aplica a `spawn()` e `fork()`, que usam fluxos.) Se um processo filho produzir mais saída do que isso, ele será eliminado e sairá com um erro.
- `shell` especifica o caminho para um shell executável ou `true`. Para funções de processo filho que normalmente executam um comando shell, essa opção permite especificar qual shell usar. Para funções que normalmente não usam um shell, esta opção permite especificar que um shell deve ser usado (definindo a propriedade como `true`) ou especificar exatamente qual shell usar.
- O tempo limite especifica o número máximo de milissegundos que o processo filho deve ter permissão para ser executado. Se não tiver saído antes de decorrido esse tempo, ele será eliminado e sairá com um erro. (Esta opção se aplica às funções `exec`, mas não a `spawn()` ou `fork()`.)
- `uid` especifica o ID do usuário (um número) sob o qual o programa deve ser executado. Se o processo pai estiver em execução em uma conta privilegiada, ele poderá usar essa opção para executar o filho com privilégios reduzidos.

16.10.2 `exec()` e `execFile()`

As funções `execSync()` e `execFileSync()` são, como suas

Os nomes indicam, síncronos: eles bloqueiam e não retornam até que o processo filho seja encerrado. Usar essas funções é muito parecido com digitar comandos Unix em uma janela de terminal: eles permitem que você execute uma sequência de comandos um de cada vez. Mas se você estiver escrevendo um programa que precisa realizar várias tarefas, e essas tarefas não dependem umas das outras de forma alguma, talvez você queira paralelizá-las e executar vários comandos ao mesmo tempo. Você pode fazer isso com as funções assíncronas `child_process.exec()` and `child_process.execFile()`.

`exec()` e `execFile()` são como suas variantes síncronas, exceto que eles retornam imediatamente com um objeto `ChildProcess` que representa o processo filho em execução e recebem um retorno de chamada de erro primeiro como seu argumento final. O retorno de chamada é invocado quando o processo filho é encerrado e, na verdade, é chamado com três argumentos. O primeiro é o erro, se houver; ele será nulo se o processo for encerrado normalmente. O segundo argumento é a saída coletada que foi enviada para o fluxo de saída padrão do filho. E o terceiro argumento é qualquer saída que foi enviada para o fluxo de erro padrão da criança.

O objeto `ChildProcess` retornado por `exec()` e `execFile()` permite que você encerre o processo filho e grave dados nele (que ele pode ler de sua entrada padrão). Abordaremos `ChildProcess` em mais detalhes quando discutirmos a função `child_process.spawn()`.

Se você planeja executar vários processos filhos ao mesmo tempo, pode ser mais fácil usar a versão "promissificada" de `exec()` que

retorna um objeto Promise que, se o processo filho for encerrado sem erro, será resolvido para um objeto com as propriedades stdout e stderr. Aqui, por exemplo, está uma função que recebe uma matriz de comandos shell como sua entrada e retorna uma Promise que resolve o resultado de todos esses comandos:

```
const child_process = require("child_process");
const util = require("util"); const execP =
util.promisify(child_process.exec);

function parallelExec(comandos) {
  Use a matriz de comandos para criar uma matriz
  dePromessas
  let promises = commands.map(command => execP(command,
  {encoding: "utf8"}));
  Devolva uma promessa que será cumprida para uma matriz do
  cumprimento
  valores de cada uma das promessas individuais. (Em vez de
  retornar objetos
  Com as propriedades stdout e stderr, apenas retornamos o
  valor stdout.)
  return Promise.all(promises)
    .then(saídas => saídas.map(saída => saída.stdout));
}

module.exports = parallelExec;
```

16.10.3 spawn()

As várias funções exec descritas até agora — síncronas e assíncronas — são projetadas para serem usadas com processos filhos que são executados rapidamente e não produzem muita saída. Mesmo o asynchronousexec() e o execFile() não são streaming: eles retornam a saída do processo em um único lote, somente após o encerramento do processo.

A função child_process.spawn() permite que você transmita

acesso à saída do processo filho, enquanto o processo ainda está em execução. Ele também permite que você grave dados no processo filho (que verá esses dados como entrada em seu fluxo de entrada padrão): isso significa que é possível interagir dinamicamente com um processo filho, enviando-o com base na saída que ele gera.

spawn() não usa um shell por padrão, então você deve invocá-lo como execFile() com o executável a ser executado e uma matriz separada de argumentos de linha de comando para passar para ele. spawn() retorna um objeto ChildProcess como execFile(), mas não recebe um argumento de retorno. Em vez de usar uma função de retorno de chamada, você escuta eventos no objeto ChildProcess e em seus fluxos.

O objeto ChildProcess retornado por spawn() é um emissor de eventos. Você pode ouvir o evento "exit" para ser notificado quando o processo filho for encerrado. Um objeto ChildProcess também tem três propriedades de fluxo. stdio e stderr são fluxos legíveis: quando o processo filho grava em itsstdio e seus fluxos stderr, essa saída se torna legível por meio dos fluxos ChildProcess. Observe a inversão dos nomes aqui. No processo filho, "stdio" é um fluxo de saída gravável, mas no processo pai, a propriedade stdio de um objeto ChildProcess é um fluxo de entrada legível.

Da mesma forma, a propriedade stdin do objeto ChildProcess é umFluxo gravável: qualquer coisa que você gravar nesse fluxo fica disponível para o processo filho em sua entrada padrão.

O objeto ChildProcess também define uma propriedade pid que especifica o

ID do processo do filho. E define um método kill() que você pode usar para encerrar um processo filho.

16.10.4 fork()

child_process.fork() é uma função especializada para executar um módulo de código JavaScript em um processo Node filho. fork() espera os mesmos argumentos que spawn(), mas o primeiro argumento deve especificar o caminho para um arquivo de código JavaScript em vez de um arquivo binário executável.

Um processo filho criado com fork() pode se comunicar com o processo pai por meio de seus fluxos de entrada e saída padrão, conforme descrito na seção anterior para spawn(). Mas, além disso, fork() permite outro canal de comunicação muito mais fácil entre os processos pai e filho.

Ao criar um processo filho com fork(), você pode usar o método `process.send()` do objeto ChildProcess retornado para enviar uma cópia de um objeto para o processo filho. E você pode ouvir o evento "message" no ChildProcess para receber mensagens do filho. O código em execução no processo filho pode usar `process.send()` para enviar uma mensagem para o pai e pode ouvir eventos de "mensagem" no processo para receber mensagens do pai.

Aqui, por exemplo, está um código que usa fork() para criar um childprocess, envia uma mensagem a esse filho e aguarda uma resposta:

```
const child_process = require("child_process");
```

Inicie um novo processo de nó executando o código em child.js em

```
nosso diretório filho =
child_process.fork('${__dirname}/child.js');

Envie uma mensagem para
child.send({x: 4, y: 3});

Imprima a resposta da criança quando ela
chegar.child.on("message", message => {
    console.log(mensagem.hipotenusa); Isso deve imprimir "5"//
    Como enviamos apenas uma mensagem, esperamos apenas uma
    resposta.
    Depois de recebê-lo, chamamos disconnect() para encerrar a
    conexão
entre pais e filhos. Isso permite que ambos os processos
saiam de forma limpa.
    criança.desconectar();
});
```

E aqui está o código que é executado no processo filho:

```
Aguarde as mensagens do nosso
process.on("message", message => {
    Quando recebermos um, faça um cálculo e envie o
    resultado
    de volta para o parent.process.send({hypotenuse:
        Math.hypot(message.x,message.y)});});
```

Iniciar processos filhos é uma operação cara, e o processo filho teria que estar fazendo ordens de magnitude mais computação antes que fizesse sentido usar fork() e interprocesscommunication dessa maneira. Se você estiver escrevendo um programa que precisa ser muito responsivo a eventos recebidos e também precisa executar cálculos demorados, considere usar um processo filho separado para executar os cálculos para que eles não bloqueiem o loop de eventos e reduzam a capacidade de resposta do processo pai.

(Embora um thread — consulte §16.11 — possa ser uma opção melhor do que um processo filho neste cenário.)

O primeiro argumento a ser enviado () será serializado com JSON.stringify() e desserializado no processo filho com JSON.parse(), portanto, você deve incluir apenas valores compatíveis com o formato JSON. send() tem um segundo argumento especial, no entanto, que permite transferir objetos Socket e Server (do módulo "net") para um processo filho. Os servidores de rede tendem a ser vinculados a E/S em vez de computação, mas se você escreveu um servidor que precisa fazer mais computação do que uma única CPU pode suportar, e se você estiver executando esse servidor em uma máquina com várias CPUs, então você pode usar fork() para criar vários processos filhos para lidar com solicitações. No processo pai, você pode ouvir eventos de "conexão" em seu objeto Server, obter o objeto Socket desse evento de "conexão" e enviá-lo — usando o segundo argumento especial — para um dos processos filhos a serem manipulados. (Observe que esta é uma solução improvável para um cenário incomum. Em vez de escrever um servidor que bifurca processos filhos, provavelmente é mais simples manter seu servidor de thread único e implantar várias instâncias dele em produção para lidar com a carga.)

16.11 Threads de trabalho

Conforme explicado no início deste capítulo, o modelo de simultaneidade do Node é de thread único e baseado em eventos. Mas na versão 10 e posterior, o Node permite uma verdadeira programação multithread, com uma API que espelha de perto a API Web Workers definida por navegadores da web (§15.13). A programação multithread tem uma reputação bem merecida

por ser difícil. Isso se deve quase inteiramente à necessidade desincronizar cuidadosamente o acesso por threads à memória compartilhada.

Mas Threads JavaScript (tanto no Node quanto nos navegadores) não compartilham memória por padrão, então os perigos e dificuldades de usar threads não se aplicam a esses "trabalhadores" em JavaScript.

Em vez de usar memória compartilhada, os threads de trabalho do JavaScript se comunicam por passagem de mensagens. O thread principal pode enviar uma mensagem para um thread de trabalho chamando o método `postMessage()` do objeto `Worker` que representa esse thread. O thread de trabalho pode receber mensagens de seu pai escutando eventos de "mensagem". E os trabalhadores podem enviar mensagens para o thread principal com sua própria versão de `postMessage()`, que o pai pode receber com seu próprio manipulador de eventos "message". O código de exemplo deixará claro como isso funciona.

Há três razões pelas quais você pode querer usar threads de trabalho no aplicativo aNode:

- Se o seu aplicativo realmente precisa fazer mais computação do que um núcleo de CPU pode lidar, os threads permitem que você distribua o trabalho entre os vários núcleos, que se tornaram comuns nos computadores hoje. Se você estiver fazendo computação científica, aprendizado de máquina ou processamento gráfico no Node, talvez queira usar threads simplesmente para jogar mais poder de computação no seu problema. Mesmo que seu aplicativo não esteja usando todo o poder de oneCPU, você ainda pode querer usar threads para manter a capacidade de resposta do thread principal. Considere um servidor que lida com solicitações grandes, mas relativamente pouco frequentes. Suponha que

recebe apenas uma solicitação por segundo, mas precisa gastar cerca de meio segundo de computação (bloqueando a CPU) para processar cada solicitação. Em média, ficará ocioso 50% do tempo. Mas quando duas solicitações chegam com alguns milissegundos de diferença uma da outra, o servidor nem mesmo poderá iniciar uma resposta à segunda solicitação até que o cálculo da primeira resposta seja concluído. Em vez disso, se o servidor usar um thread de trabalho para executar a computação, o servidor poderá iniciar a resposta a ambas as solicitações imediatamente e fornecer uma experiência melhor para os clientes do servidor. Supondo que o servidor tenha mais de um núcleo de CPU, ele também pode calcular o corpo de ambas as respostas em paralelo, mas mesmo que haja apenas um único núcleo, o uso de workers ainda melhora a capacidade de resposta. Em geral, os workers nos permitem transformar operações síncronas de bloqueio em operações assíncronas sem bloqueio. Se você estiver escrevendo um programa que depende de código herdado que é inevitavelmente síncrono, poderá usar workers para evitar o bloqueio quando precisar chamar esse código legado. Os threads de trabalho não são tão pesados quanto os processos filhos, mas não são leves. Geralmente não faz sentido criar um trabalhador, a menos que você tenha um trabalho significativo para fazer. E, de modo geral, se o seu programa não estiver vinculado à CPU e não estiver tendo problemas de capacidade de resposta, você provavelmente não precisará de workerthreads.

16.11.1 Criando trabalhadores e passando mensagens

O módulo Node que define os trabalhadores é conhecido como "worker_threads". Nesta seção, vamos nos referir a ele com os threads identificadores:

```
const threads = require("worker_threads");
```

Este módulo define uma classe Worker para representar um thread de trabalho e você pode criar um novo thread com os threads. Worker(). O código a seguir demonstra o uso desse construtor para criar um trabalho e mostra como passar mensagens do thread principal para o trabalhador e do trabalhador para o thread principal. Ele também demonstra um truque que permite colocar o código do thread principal e o código do thread de trabalho no mesmo arquivo.

```
const threads = require("worker_threads");
```

O módulo `worker_threads` exporta a propriedade booleana `isMainThread`.// Essa propriedade é true quando o Node está executando o thread principal e é// false quando o Node está executando um trabalhador. Podemos usar este fato para implementar // os threads principal e de trabalho no mesmo arquivo.`if (threads.isMainThread) {`

Se estivermos executando no thread principal, tudo o que fazemos é exportar uma função. Em vez de executar um computacionalmente intensivo tarefa no thread principal, essa função passa a tarefa para um trabalhador e retorna uma promessa que será resolvida quando o trabalhador terminar.

```
module.exports = função reticulateSplines(splines) {
    return new Promise((resolve,reject) => {
        Crie um worker que carregue e execute esse mesmo arquivo de código.
        Observe o uso do __filename especial variável.
        let reticulator = novos threads.Trabalhador(__filename);

        Passe uma cópia da matriz splines para o worker.reticulator.postMessage(splines);

        E então resolver ou rejeitar a promessa quando
        obter uma mensagem ou erro do
        worker.reticulator.on("message", resolve);
```

```

reticulator.on("erro", rejeitar);});}
else {
}

Se chegarmos aqui, significa que estamos no trabalhador,
então registramos um
manipulador para obter mensagens do thread principal.
Este trabalhador é projetado
para receber apenas uma única mensagem, então registramos
o manipulador de eventos
com once() em vez de on(). Isso permite que o trabalhador
saia naturalmente
    quando seu trabalho estiver
        concluído.threads.parentPort.once("message", splines => {
            Quando obtemos as splines do thread pai,
laço
            através deles e reticular todos
                eles.for(let spline of splines) {
                    Por exemplo, suponha que spline
objetos geralmente
                    tem um método reticulate() que faz muitos
computação.
                    spline.reticulate ? spline.reticulate() :
spline.reticulated = true;
                }
}

        Quando todas as splines foram (finalmente!)
reticulado
        passe uma cópia de volta para o thread
principal.threads.parentPort.postMessage(splines);
});}

```

O primeiro argumento para o construtor Worker() é o caminho para um arquivo de código JavaScript que deve ser executado no thread. No código anterior, usamos o identificador de __filename predefinido para criar um worker que carrega e executa o mesmo arquivo que o thread principal. Em geral, porém, você estará passando um caminho de arquivo. Observe que, se você especificar um caminho relativo, ele será relativo a process.cwd(), não relativo ao caminho em execução no momento

módulo. Se você quiser um caminho relativo ao módulo atual, use algo como
path.resolve(__dirname,'workers/reticulator.js').

O construtor Worker() também pode aceitar um objeto como seu segundo argumento, e as propriedades desse objeto fornecem configuração opcional para o trabalhador. Abordaremos várias dessas opções mais tarde, mas, por enquanto, observe que, se você passar {eval: true} como o segundo argumento, o primeiro argumento para Worker() será interpretado como uma string de código JavaScript a ser avaliada em vez de um nome de arquivo:

```
novos tópicos. Trabalhador('
const threads = require("worker_threads");
threads.parentPort.postMessage(threads.isMainThread);',
{eval: true}).on("message", console.log); // Isso
imprimirá "false"
```

O Node faz uma cópia do objeto passado para postMessage() em vez de compartilhá-lo diretamente com o thread de trabalho. Isso impede que o thread de trabalho e o thread principal compartilhem memória. Você pode esperar que essa cópia seja feita com JSON.stringify() e JSON.parse() (§11.6). Mas, na verdade, o Node empresta uma técnica mais robusta conhecida como algoritmo de clone estruturado dos navegadores da web.

O algoritmo de clone estruturado permite a serialização da maioria dos tipos JavaScript, incluindo objetos Map, Set, Date e RegExp e matrizes tipadas, mas não pode, em geral, copiar tipos definidos pelo ambiente de host do Node, como soquetes e fluxos. Observe, no entanto, que Buffer objects são parcialmente suportados: se você passar um Buffer para

`postMessage()` ele será recebido como um `Uint8Array` e pode ser convertido novamente em um `Buffer` com `Buffer.from()`. Leia mais sobre o algoritmo de clone estruturado em "O Algoritmo de Clone Estruturado".

16.11.2 O ambiente de execução do trabalhador

Na maioria das vezes, o código JavaScript em um thread de trabalho do Node é executado exatamente como seria no thread principal do Node. Existem algumas diferenças das quais você deve estar ciente, e algumas dessas diferenças envolvem propriedades do segundo argumento opcional para o construtor `Worker()`:

- Como vimos, `threads.isMainThread` é verdadeiro no thread principal, mas é sempre falso em qualquer thread de trabalho.
- Em um thread de trabalho, você pode usar `threads.parentPort.postMessage()` para enviar uma mensagem para o thread pai e `threads.parentPort.on` para registrar manipuladores de eventos para mensagens do thread pai. No thread principal, `threads.parentPort` é sempre nulo. Em um thread de trabalho, `threads.workerData` é definido como uma cópia da propriedade `workerData` do segundo argumento para o construtor `Worker()`. No thread principal, essa propriedade é sempre nula. Você pode usar essa propriedade `workerData` para passar uma mensagem inicial para o worker que estará disponível assim que ele for iniciado, para que o worker não precise esperar por um evento de "mensagem" antes de começar a trabalhar. Por padrão, `process.env` em um thread de trabalho é uma cópia de `process.env` no thread pai. Mas o thread pai pode

- especifique um conjunto personalizado de variáveis de ambiente definindo a propriedade env do segundo argumento para o construtor Worker(). Como um caso especial (e potencialmente perigoso), o thread pai pode definir a propriedade env como threads. SHARE_ENV, o que fará com que os dois threads compartilhem um único conjunto de variáveis de ambiente para que uma alteração em um thread seja visível no outro. Por padrão, o fluxo process.stdin em um worker nunca tem nenhum dado legível nele. Você pode alterar esse padrão ignorando stdin: true no segundo argumento para o construtor theWorker(). Se você fizer isso, a stdinpropriedade do objeto Worker será um fluxo gravável. Todos os dados que o pai grava em worker.stdin tornam-se legíveis em process.stdin no worker. Por padrão, process.stdout e process.stderrstreams no worker são simplesmente canalizados para os correspondingstreams no thread pai. Isso significa, por exemplo, que thatconsole.log() e console.error() produzem saída exatamente da mesma maneira em um thread de trabalho e no thread principal. Você pode substituir esse padrão passando stdout:true ou stderr:true no segundo argumento para o construtor Worker(). Se você fizer isso, qualquer saída que theworker gravar nesses fluxos se tornará legível pelo parentthread no worker.stdout e worker.stderrthreads. (Há uma inversão potencialmente confusa de streamdirections aqui, e vimos a mesma coisa com childprocesses anteriormente no capítulo: os fluxos de saída de um workerthread são fluxos de entrada para o thread pai e o inputstream de um worker é um fluxo de saída para o pai.) Se um thread de trabalho chamar process.exit(), somente o thread será encerrado, não o processo inteiro.

- Os threads de trabalho não têm permissão para alterar o estado compartilhado do processo do qual fazem parte. Funções como `process.chdir()` e `process.setuid()` lançarão exceções quando invocadas de um trabalhador. Os sinais do sistema operacional (como SIGINT e SIGTERM) são entregues apenas ao thread principal; eles não podem ser recebidos ou manipulados em threads de trabalho.

16.11.3 Canais de comunicação e portas de mensagem

Quando um novo thread de trabalho é criado, um canal de comunicação é criado junto com ele que permite que as mensagens sejam passadas entre o worker e o thread pai. Como vimos, o `workerthread` usa `threads.parentPort` para enviar e receber mensagens de e para o thread pai, e o thread pai usa o objeto `Worker` para enviar e receber mensagens de e para o thread de trabalho.

A API de thread de trabalho também permite a criação de canais de comunicação personalizados usando a API `MessageChannel` definida por navegadores da Web e abordada em §15.13.5. Se você leu essa seção, muito do que se segue soará familiar para você.

Suponha que um trabalhador precise lidar com dois tipos diferentes de mensagens enviadas por dois módulos diferentes no thread principal. Esses dois módulos diferentes podem compartilhar o canal padrão e enviar mensagens com `worker.postMessage()`, mas seria mais limpo se cada módulo tivesse seu próprio canal privado para enviar mensagens ao worker. Ou considere o caso em que o fio principal cria dois trabalhadores independentes. Um canal de comunicação personalizado pode permitir que os dois trabalhadores

para se comunicarem diretamente uns com os outros, em vez de ter que enviar todas as suas mensagens por meio dos pais.

Crie um novo canal de mensagens com o construtor `MessageChannel()`. Um objeto `MessageChannel` tem duas propriedades, `namedport1` e `port2`. Essas propriedades referem-se a um par de objetos `MessagePort`. Chamar `postMessage()` em uma das portas fará com que um evento "message" seja gerado na outra com um clone estruturado do objeto `Message`:

```
const threads = require("worker_threads");
let canal =
novos threads. MessageChannel();
channel.port2.on("message", console.log); // Registra
todas as mensagens que
recebemoschannel.port1.postMessage("hello"); // Fará com
que "hello" seja impresso
```

Você também pode chamar `close()` em qualquer porta para interromper a conexão entre as duas portas e sinalizar que nenhuma mensagem será trocada. Quando `close()` é chamado em qualquer uma das portas, um evento "close" é entregue a ambas as portas.

Observe que o exemplo de código acima cria um par de objetos `MessagePort` e, em seguida, usa esses objetos para transmitir uma mensagem dentro do thread principal. Para usar canais de comunicação personalizados com os trabalhadores, devemos transferir uma das duas portas do thread em que é criado para o thread em que será usado. A próxima seção explica como fazer isso.

16.11.4 Transferindo MessagePorts e TypedArrays

A função postMessage() usa o algoritmo de clone estruturado e, como observamos, ela não pode copiar objetos como SSockets e Streams. It pode lidar com objetos MessagePort, mas apenas como um caso especial usando uma técnica especial. O método postMessage() (de um objeto Worker, de threads.parentPort ou de qualquer objeto MessagePort) recebe um segundo argumento opcional. Esse argumento (chamado transferList) é uma matriz de objetos que devem ser transferidos entre threads em vez de serem copiados.

Um objeto MessagePort não pode ser copiado pelo algoritmo de clone estruturado, mas pode ser transferido. Se o primeiro argumento topostMessage() incluiu um ou mais MessagePorts (aninhados arbitrariamente profundamente dentro do objeto Message), esses objetos MessagePort também devem aparecer como membros da matriz passada como o segundo argumento. Isso informa ao Node que ele não precisa fazer uma cópia do MessagePort e, em vez disso, pode apenas fornecer o objeto existente para o outro thread. A principal coisa a entender, no entanto, sobre a transferência de valores entre threads é que, uma vez que um valor é transferido, ele não pode mais ser usado no thread que chamou postMessage().

Veja como você pode criar um novo MessageChannel e transferir um de seus MessagePorts para um trabalhador:

```
Crie um canal de comunicação personalizadoconst  
threads = require("worker_threads"); let canal  
= novos threads. MessageChannel();
```

Use o canal padrão do trabalhador para transferir uma extremidade do canal novo// para o trabalhador. Suponha que, quando o worker recebe esta mensagem, ele imediatamente começa a escutar mensagens em

```
o novo channel.worker.postMessage({ comando:  
"changeChannel", data:channel.port1 },  
  
[ canal.port1 ]);
```

*Agora envie uma mensagem para o trabalhador usando nossa
extremidade docanal personalizado.port2.postMessage("Você
pode me ouvir agora?");*

*E ouça as respostas do trabalhador como
wellchannel.port2.on("message", handleMessagesFromWorker);*

Os objetos MessagePort não são os únicos que podem ser transferidos. Se você chamar postMessage() com um array tipado como a mensagem (ou com uma mensagem que contém um ou mais arrays tipados aninhados arbitrariamente dentro da mensagem), esse array tipado (ou esses arrays tipados) será simplesmente copiado pelo algoritmo de clone estruturado. Mas as matrizes digitadas podem ser grandes; Por exemplo, se você estiver usando um thread de trabalho para fazer processamento de imagens em milhões de pixels. Portanto, para maior eficiência, postMessage() também nos dá a opção de transferir matrizes digitadas em vez de copiá-las. (Os threads compartilham memória por padrão. Threads de trabalho em JavaScript geralmente evitam memória compartilhada, mas quando permitimos esse tipo de transferência controlada, isso pode ser feito com muita eficiência.) O que torna isso seguro é que, quando uma matriz tipada é transferida para outro thread, ela se torna inutilizável no thread que a transferiu. No cenário de processamento de imagem, o thread principal pode transferir os pixels de uma imagem para o thread de trabalho e, em seguida, o thread de trabalho pode transferir os pixels processados de volta para o thread principal quando isso for concluído. A memória não precisaria ser copiada, mas nunca seria acessível por dois threads ao mesmo tempo.

Para transferir uma matriz tipada em vez de copiá-la, inclua o ArrayBuffer

que apóia a matriz no segundo argumento para postMessage():

```
let pixels = new Uint32Array(1024*1024); 4 megabytes de  
memória
```

Suponha que lemos alguns dados nessa matriz digitada e, em seguida, transferimos os pixels para um trabalhador sem copiar. Observe que não colocamos o próprio array// na lista de transferências, mas o objeto Buffer do arrayem vez disso.worker.postMessage(pixels, [pixels.buffer]);

Assim como acontece com os MessagePorts transferidos, uma matriz tipada transferida torna-se inutilizável depois de transferida. Nenhuma exceção será lançada se você tentar usar um MessagePort ou uma matriz tipada que tenha sido transferida; esses objetos simplesmente param de fazer qualquer coisa quando você interage com eles.

16.11.5 Compartilhando matrizes tipadas entre threads

Além de transferir matrizes tipadas entre threads, é realmente possível compartilhar uma matriz tipada entre threads. Basta criar um SharedArrayBuffer do tamanho desejado e usar esse buffer para criar uma matriz tipada. Quando uma matriz tipada que é apoiada por aSharedArrayBuffer é passada via postMessage(), a memória subjacente será compartilhada entre os threads. Você não deve incluir o buffer compartilhado no segundo argumento para postMessage() neste caso.

Você realmente não deve fazer isso, no entanto, porque o JavaScript nunca foi projetado com a segurança de thread em mente e a programação multithread é muito difícil de acertar. (E é por isso que SharedArrayBuffer não foi abordado no §11.2: é um recurso de nicho difícil de acertar.) Mesmo

O operador Simple ++ não é thread-safe porque precisa ler um valor, incrementá-lo e gravá-lo de volta. Se dois threads estiverem incrementando um valor ao mesmo tempo, ele geralmente será incrementado apenas uma vez, como demonstra o código a seguir:

```
const threads = require("worker_threads");

if (threads.isMainThread) {
    No thread principal, criamos uma matriz tipada
    compartilhada com
    um elemento. Ambos os tópicos serão capazes de ler e
    escrever
    sharedArray[0] ao mesmo tempo. let sharedBuffer
    = new SharedArrayBuffer(4); let sharedArray =
    new Int32Array(sharedBuffer);

    Agora crie um thread de trabalho, passando o array
    compartilhado para ele com
    como seu valor inicial workerData para que não tenhamos
    que nos preocupar com
    Enviando e recebendo um messagelet worker = new threads.
    Trabalhador(__filename, { workerData:sharedArray });

    Aguarde até que o trabalhador comece a ser
    executado e, em seguida, incremente o
    inteiro compartilhado 10 milhões de
    vezes. worker.on("online", () => {
        for(let i = 0; i < 10_000_000; i++) sharedArray[0]++;
        Assim que terminarmos nossos incrementos, começamos
        Ouvindo por
        eventos de mensagem para que saibamos quando o
        trabalhador terminou. worker.on("message", () => {
            Embora o inteiro compartilhado tenha sido
            Incrementado
            20 milhões de vezes, seu valor geralmente será
            muito menos.
            No meu computador, o valor final é normalmente
            menos de 12 milhões.
            console.log(sharedArray[0]);});
}
```

```
});}
else {
  No thread de trabalho, obtemos a matriz compartilhada
  fromworkerData
  e, em seguida, incremente-o 10 milhões de vezes.let
  sharedArray = threads.workerData; for(let i = 0; i <
  10_000_000; i++) sharedArray[0]++; // Quando terminarmos de
  incrementar, deixe o thread principal
  knowthreads.parentPort.postMessage("done");}
```

Um cenário em que pode ser razoável usar a SharedArrayBuffer é quando os dois threads operam em seções totalmente separadas da memória compartilhada. Você pode impor isso criando matrizes de dois tipos que servem como exibições de regiões não sobrepostas do buffer compartilhado e, em seguida, fazer com que seus dois threads usem essas duas matrizes de tipos separados. Uma classificação de mesclagem paralela pode ser feita assim: um thread classifica a metade inferior de uma matriz e o outro thread classifica a metade superior, por exemplo. Ou alguns tipos de algoritmos de processamento de imagem também são adequados para essa abordagem: vários threads trabalhando em regiões disjuntas da imagem.

Se você realmente precisar permitir que vários threads acessem a mesma região de uma matriz compartilhada, poderá dar um passo em direção à segurança do thread com as funções definidas pelo objeto Atomics. Atomics foi adicionado ao JavaScript quando SharedArrayBuffer foi para definir operações atômicas nos elementos de uma matriz compartilhada. Por exemplo, a função Atomics.add() lê o elemento especificado de uma matriz compartilhada, adiciona um valor especificado a ele e grava a soma de volta na matriz. Ele faz isso atomicamente como se fosse uma única operação e garante que nenhum outro thread possa ler ou gravar o valor enquanto a operação estiver ocorrendo. Atomics.add() nos permite reescrever o código de incremento paralelo que

Acabei de olhar e obter o resultado correto de 20 milhões de incrementos de um elemento de matriz compartilhado:

```
const threads = require("worker_threads");

if (threads.isMainThread) {
let sharedBuffer = new SharedArrayBuffer(4); let sharedArray
= new Int32Array(sharedBuffer); let worker = novos threads.
Trabalhador(__filename, { workerData:sharedArray });

    worker.on("online", () => {
        for(let i = 0; i < 10_000_000; i++) {
            Atomics.add(arrayCompartilhado, 0, 1); Threadsafe
incremento atômico
        }

        worker.on("mensagem", (mensagem) => {
            Quando ambos os threads estiverem concluídos, use um
funcção threadsafe
para ler a matriz compartilhada e confirmar se ela
tem o
valor esperado de
20.000.000.console.log(Atomics.load(sharedArray,
0));});});});} else {

    let sharedArray = threads.workerData;
    for(let i = 0; i < 10_000_000; i++) {
        Atomics.add(arrayCompartilhado, 0, 1); Threadsafe
incremento atômico
    }threads.parentPort.postMessage("concluído
");}
```

Esta nova versão do código imprime corretamente o número 20.000.000. Mas é cerca de nove vezes mais lento do que o código incorreto que substitui. Seria muito mais simples e muito mais rápido fazer apenas 20 milhões

incrementos em um thread. Observe também que as operações atômicas podem ser capazes de garantir a segurança do thread para algoritmos de processamento de imagem para os quais cada elemento de matriz é um valor totalmente independente de todos os outros valores. Mas na maioria dos programas do mundo real, vários elementos de matriz geralmente estão relacionados uns aos outros e algum tipo de sincronização de thread de nível superior é necessária. A função de baixo nível `Atomics.wait()` e `Atomics.notify()` pode ajudar com isso, mas uma discussão sobre seu uso está fora do escopo deste livro.

16.12 Resumo

Embora o JavaScript tenha sido criado para ser executado em navegadores da web, o Node transformou o JavaScript em uma linguagem de programação de uso geral. É particularmente popular para implementar servidores web, mas suas ligações profundas ao sistema operacional significam que também é uma boa alternativa para scripts de shell.

Os tópicos mais importantes abordados neste longo capítulo incluem:

- APIs assíncronas por padrão do Node e seu estilo de simultaneidade baseado em eventos e thread único. Tipos de dados, buffers e fluxos fundamentais do Node. Módulos "fs" e "path" do Node para trabalhar com o sistema de arquivos. Módulos "http" e "https" do Node para escrever clientes e servidores HTTP. Módulo "net" do Node para escrever clientes e servidores não-HTTP.
-

- Módulo "child_process" do Node para criar e se comunicar com processos filhos. Módulo
 - "worker_threads" do Node para programação multithread verdadeira usando passagem de mensagens em vez de memória compartilhada.
-

1 O Node define uma função `fs.copyFile()` que você realmente usaria na prática.
2 Geralmente, é mais limpo e simples definir o código de trabalho em um arquivo separado. Mas esse truque de ter dois tópicos executando seções diferentes do mesmo arquivo me surpreendeu quando eu encontrei-o para a chamada de sistema `fork()` do Unix. E acho que vale a pena demonstrar essa técnica simplesmente por sua estranha elegância.

Capítulo 17. Ferramentas e extensões JavaScript

Parabéns por chegar ao capítulo final deste livro. Se você leu tudo o que vem antes, agora você tem um detalhadocompreensão da linguagem JavaScript e sabe como usá-la emNode e em navegadores da web. Este capítulo é uma espécie de graduaçãoapresente: apresenta um punhado de ferramentas de programação importantes quemuitos programadores JavaScript acham úteis e também descreve duas extensões amplamente usadas para a linguagem JavaScript principal. Quer você opte ou não por usar essas ferramentas e extensões para seus próprios projetos, é quase certo que as verá usadas em outros projetos, por isso é importantesaber pelo menos quais são.

As ferramentas e extensões de linguagem abordadas neste capítulo são:

- ESLint para encontrar possíveis bugs e problemas de estilo em seu código. Mais bonito para formatar seu código
- JavaScript de maneira padronizada. Jest como uma solução completa para escrever testes de unidade JavaScript.npm para
- gerenciar e instalar as bibliotecas de software das quais seu programa depende. Ferramentas de agrupamento de código, como webpack, Rollup e Parcel, que convertem seus módulos de código JavaScript em um único pacote para uso na web.
-

Babel para traduzir código JavaScript que usa novos recursos de linguagem (ou que usa extensões de linguagem) em código JavaScript que pode ser executado em navegadores da web atuais. A extensão de linguagem JSX (usada pela estrutura React) que permite descrever interfaces de usuário usando JavaScript expressões que se parecem com marcação HTML. A extensão de linguagem Flow (ou a extensão TypeScript semelhante) que permite anotar seu código JavaScript com tipos e verificar seu código quanto à segurança de tipo. Este capítulo não documenta essas ferramentas e extensões de forma abrangente. O objetivo é simplesmente explicá-los com profundidade suficiente para que você possa entender por que eles são úteis e quando você pode querer usá-los. Tudo o que foi abordado neste capítulo é amplamente utilizado no mundo da programação JavaScript e, se você decidir adotar uma ferramenta ou extensão, encontrará muita documentação e tutoriais online.

17.1 Linting com ESLint

Em programação, o termo lint refere-se ao código que, embora tecnicamente correto, é feio, ou um possível bug, ou abaixo do ideal de alguma forma. Alinter é uma ferramenta para detectar lint em seu código, e linting é o processo de executar um linter em seu código (e depois corrigir seu código para remover o lint para que o linter não reclame mais).

O linter mais comumente usado para JavaScript hoje é o [ESLint](#). Se você executá-lo e dedicar um tempo para realmente corrigir os problemas apontados, isso tornará seu código mais limpo e menos propenso a ter bugs. Considere o seguinte código:

```
var x = 'não utilizado';

função de exportação fatorial(x) {
    if (x == 1) {
        retornar
    } else {
        return x * fatorial(x-1)}}


```

Se você executar ESLint neste código, poderá obter uma saída como esta:

```
$ código eslint/ch17/linty.js

código/ch17/linty.js
  1:1  erro      Var inesperado, use let ou const em vez disso
sem var
  1:5  erro      'x' recebe um valor, mas nunca é usado
sem haste não utilizada
  1:9  aviso     As cadeias de caracteres devem usar aspas duplas
aspas
  4:11 erro     Esperava '===' e, em vez disso, viu '=='
eqeqeq
  5:1  erro      Recuo esperado de 8 espaços, mas encontrado 6
indentar
  7:28 erro     Ponto e vírgula ausente
semi

* 6 problemas (5 erros, 1 aviso)
3 erros e 1 aviso potencialmente corrigíveis com a opção '--
fix'.
```

Linters podem parecer minuciosos às vezes. Realmente importa se usamos aspas duplas ou aspas simples para nossas strings? Por outro lado, obter o recuo correto é importante para a legibilidade, e usar === e let em vez de == e var protege você de bugs sutis. E variáveis não utilizadas são peso morto em seu código - não há razão para mantê-las por perto.

O ESLint define muitas regras de linting e possui um ecossistema de plug-ins que adicionam muito mais. Mas o ESLint é totalmente configurável e você pode definir um arquivo de configuração que ajusta o ESLint para impor exatamente as regras desejadas e apenas essas regras.

17.2 Formatação JavaScript com Prettier

Uma das razões pelas quais alguns projetos usam linters é impor um estilo de codificação consistente para que, quando uma equipe de programadores estiver trabalhando em uma base de código compartilhada, eles usem convenções de código compatíveis. Isso inclui regras de recuo de código, mas também pode incluir coisas como que tipo de aspas são preferidas e se deve haver um espaço entre a palavra-chave `for` e o parêntese aberto que a segue.

Uma alternativa moderna para impor regras de formatação de código por meio de um linter é adotar uma ferramenta como o Prettier para analisar e reformatar automaticamente todo o seu código.

Suponha que você tenha escrito a seguinte função, que funciona, mas é formatada de forma não convencional:

```
função fatorial(x){  
    if(x==1){retornar 1}  
    else{return x*fatorial(x-1)}  
}
```

Executar o Prettier neste código corrige o recuo, adiciona ponto e vírgula ausente, adiciona espaços ao redor de operadores binários e insere quebras de linha

depois de { e antes }, resultando em um código de aparência muito mais convencional:

```
$ mais bonito fatorial.jsfunction
fatorial(x) {
  if (x === 1) {
    retornar
    1;} else {
  return x * fatorial(x - 1);}}
```

Se você invocar o Prettier com a opção --write, ele simplesmente reformatar o arquivo especificado no local, em vez de imprimir uma versão reformatada. Se você usar o git para gerenciar seu código-fonte, poderá invocar Prettier com a opção --write em um gancho de confirmação para que o código seja formatado automaticamente antes de ser verificado.

O Prettier é particularmente poderoso se você configurar seu editor de código para executá-lo automaticamente toda vez que salvar um arquivo. Acho libertador escrever código desleixado e vê-lo corrigido automaticamente para mim.

Prettier é configurável, mas tem apenas algumas opções. Você pode selecionar o comprimento máximo da linha, a quantidade de recuo, se o ponto e vírgula deve ser usado, se as strings devem ser aspas simples ou duplas e algumas outras coisas. Em geral, as opções padrão do Mais bonito são bastante razoáveis. A ideia é que você apenas adote o Prettier para o seu projeto e nunca mais precise pensar na formatação do código.

Pessoalmente, gosto muito de usar o Prettier em projetos JavaScript. Eu não o usei para o código deste livro, no entanto, porque em grande parte do meu código eu confio na formatação manual cuidadosa para alinhar meus comentários verticalmente, e

Prettier bagunça tudo.

17.3 Teste de unidade com Jest

Escrever testes é uma parte importante de qualquer projeto de programação não trivial. Linguagens dinâmicas como JavaScript suportam estruturas de teste que reduzem drasticamente o esforço necessário para escrever testes e quase tornam a escrita de testes divertida! Existem muitas ferramentas e bibliotecas de teste para JavaScript, e muitas são escritas de forma modular para que seja possível escolher uma biblioteca como seu executor de teste, outra biblioteca para asserções e uma terceira para simulação. Nesta seção, no entanto, descreveremos Jest, que é um framework popular que inclui tudo o que você precisa em um único pacote.

Suponha que você tenha escrito a seguinte função:

```
constgetJSON = require("./getJSON.js");

/**
 * getTemperature() pega o nome de uma cidade como sua
 * entrada e retorna
 * uma promessa que ressolverá a temperatura atual daquela
 * cidade,
 * em graus Fahrenheit. Ele se baseia em um serviço da Web
 * (falso) que retorna
 * temperaturas mundiais em graus Celsius.*/
module.exports
= async function getTemperature(city) { // Obtenha a
temperatura em Celsius do serviço web

    let c = await getJSON(
        'https://globaltemps.example.com/api/city/${city.toLowerCase()}'
    ); // Converta para Fahrenheit e retorne esse
    valor.return (c * 5 / 9) + 32; // TODO: verifique
    novamente isso
```

```
fórmul  
a};
```

Um bom conjunto de testes para essa função pode verificar se getTemperature() está buscando a URL correta e se está convertendo as escalas de temperatura corretamente. Podemos fazer isso com um teste baseado em Jest como o seguinte. Este código define uma implementação simulada de getJSON() para que o teste não faça uma solicitação de rede. E como getTemperature() é uma função assíncrona, os testes também são assíncronos - pode ser complicado testar funções assíncronas, mas Jest torna isso relativamente fácil:

Importe a função que vamos testar

```
const getTemperature =  
require("./getTemperature.js");
```

E simule o módulo getJSON() que

```
getTemperature() depende em jest.mock("./getJSON");  
const getJSON = require("./getJSON.js");
```

Diga à função getJSON() simulada para retornar uma Promise já resolvida// com o valor de cumprimento 0.getJSON.mockResolvedValue(0);

Nosso conjunto de testes para getTemperature()

```
describe("getTemperature()", () => {  
  Este é o primeiro teste. Estamos garantindo que  
  as chamadas thatgetTemperature()  
  getJSON() com a URL que expecttest("Invoca a  
  API correta", async() => {  
    let expectedURL =  
    "https://globaltemps.example.com/api/city/vancouver";  
    let t = await(getTemperature("Vancouver")); // Zombarias  
    de brincadeira lembram como foram chamadas, e nós  
    pode verificar isso.  
    expect(getJSON).toHaveBeenCalledWith(expectedURL);});  
});
```

```

Este segundo teste verifica se
getTemperature() converte
    Celsius para Fahrenheit corretamente test("Converte
    C em F corretamente", async () => {
       getJSON.mockResolvedValue(0);                                Se
getJSON retorna 0C
        expect(await getTemperature("x")).toBe(32);  Nós
espere 32F

        100C deve ser convertido em
        212F getJSON.mockResolvedValue(100);                  Se
getJSON retorna 100C
        expect(await getTemperature("x")).toBe(212);  Nós
espere 212F
});});;

```

Com o teste escrito, podemos usar o comando jest para executá-lo e descobrimos que um de nossos testes falha:

```

$ é getTemperature
FAILch17/getTemperature.test.js
  etTemperature()
    ✓ Invoca a API correta (4ms)×
      Converte C em F corretamente (3ms)

● getTemperature() > Converte C em F corretamente
  expect(received).toBe(expected) // Object.is igualdade

    Esperado: 212Recebido:
    87.55555555555556

    29 |       100C deve converter para 212F
    30 |       getJSON.mockResolvedValue(100); Se
getJSON retorna 100C
      > 31 |       esperar(aguardar
getTemperature("x")).toBe(212); Espere 212F
      |^32 |}); 33 | }); 34 |

```

em Objeto. <anonymous>

```
(Capítulo 17/getTemperature.test.js:31:43)
```

```
Conjuntos de testes: 1 com falha, 1
totalTests:1 com falha, 1 aprovado, 2
totalSnapshots:0 totalTime:1.403sExecutou
todos os conjuntos de testes correspondentes a
/getTemperature/i.
```

Nossa implementação `getTemperature()` está usando a fórmula errada para converter C em F. Ele multiplica por 5 e divide por 9 em vez de multiplicar por 9 e dividir por 5. Se corrigirmos o código e executarmos Jest again, podemos ver os testes passarem. E, como bônus, se adicionarmos o argumento `--coverage` quando invocarmos jest, ele calculará e exibirá a cobertura de código para nossos testes:

```
$ jest --coverage getTemperature
PASSch17/getTemperature.test.js
  getTemperature()
    ✓ Invoca a API correta (3ms)
    Converte C em F corretamente (1ms)

-----|-----|-----|-----|-----|-----|
-----|Arquivo| % Stmt| % Ramificação| % Funcs| %|
Linhas|Linha Descoberta #s|
```

Todos os arquivos	71,43	100	33.33	83.33
getJSON.js	33.33	100	0	50
2 getTemperature.js	100 100		100 100	

```
|-----|-----|-----|-----|-----|
-----|Conjuntos de testes: 1
aprovado, 1 totalTestes: 2 aprovados, 2
totalSnapshots: 0 totalTime: 1.508s Executou
todos os conjuntos de testes correspondentes a
/getTemperature/i.
```

A execução de nosso teste nos deu 100% de cobertura de código para o módulo que estávamos testando, que é exatamente o que queríamos. Ele nos deu apenas uma cobertura parcial de `getJSON()`, mas zombamos desse módulo e não estávamos tentando testá-lo, então isso é esperado.

17.4 Gerenciamento de pacotes com npm

No desenvolvimento de software moderno, é comum que qualquer programa não trivial que você escreva dependa de bibliotecas de software de terceiros. Se você estiver escrevendo um servidor web no Node, por exemplo, você pode estar usando a estrutura Express. E se você estiver criando uma interface de usuário para ser exibida em um navegador da web, poderá usar uma estrutura de front-end como React ou LitElement ou Angular. Um gerenciador de pacotes facilita a localização e instalação de pacotes de terceiros como esses. Tão importante quanto, um gerenciador de pacotes rastreia de quais pacotes seu código depende e salva essas informações em um arquivo para que, quando outra pessoa quiser experimentar seu programa, ela possa baixar seu código e sua lista de dependências e, em seguida, usar seu próprio gerenciador de pacotes para instalar todos os pacotes de terceiros que seu código precisa.

npm é o gerenciador de pacotes que vem junto com o Node e foi introduzido em §16.1.5. No entanto, é tão útil para a programação JavaScript do lado do cliente quanto para a programação do lado do servidor com o Node.

Se você estiver experimentando o projeto JavaScript de outra pessoa, uma das primeiras coisas que você fará depois de baixar o código é `git clone` o projeto e executar `npm install`. Isso lê as dependências listadas em `package.json` e baixa os pacotes de terceiros que o projeto precisa e os salva em um diretório `node_modules/`.

Você também pode digitar `npm install <package-name>` para instalar um pacote específico no diretório `node_modules/` do seu projeto:

```
$ npm install express
```

Além de instalar o pacote nomeado, o npm também faz um registro da dependência no arquivo `package.json` do projeto. Gravar dependências dessa maneira é o que permite que outras pessoas instalem essas dependências simplesmente digitando `npm install`.

O outro tipo de dependência é de ferramentas de desenvolvedor que são necessárias para desenvolvedores que desejam trabalhar em seu projeto, mas não são realmente necessárias para executar o código. Se um projeto usa o Prettier, por exemplo, para garantir que todo o seu código seja formatado de forma consistente, o Prettier é uma "dependência de desenvolvimento" e você pode instalar e gravar uma delas com `--save-dev`:

```
$ npm install --save-dev mais bonito
```

Às vezes, você pode querer instalar ferramentas de desenvolvedor globalmente para que elas sejam acessíveis em qualquer lugar, mesmo para código que não faz parte de um projeto formal com um arquivo `package.json` e um diretório `node_modules/`. Para isso, você pode usar a opção `-g` (para global):

```
$ npm install -g eslint jest/usr/local/bin/eslint
-
>/usr/local/lib/node_modules/eslint/bin/eslint.js/
usr/local/bin/jest -
>/usr/local/lib/node_modules/jest/bin/jest.js+
jest@24.9.0+ eslint@6.7.2 adicionou 653 pacotes de
414 colaboradores em 25.596s
```

```
$ qual eslint
```

```
/usr/local/bin/eslint$  
que  
jest/usr/local/bin/jest
```

Além do comando "install", o npm suporta os comandos "uninstall" e "update", que fazem o que seus nomes dizem. O npm também tem um comando "audit" interessante que você pode usar para encontrar e corrigir vulnerabilidades de segurança em suas dependências:

```
$ npm auditoria --correção  
==== Relatório de segurança de auditoria NPM ====  
encontrou 0 vulnerabilidades  
em 876354 pacotes digitalizados
```

Quando você instala uma ferramenta como o ESLint localmente para um projeto, o script eslint termina em ./node_modules/.bin/eslint, o que torna o comando difícil de executar. Felizmente, o npm vem com um comando conhecido como "npx", que você pode usar para executar ferramentas instaladas localmente com comandos como npx eslint ou npx jest. (E se você usar o npx para invocar uma ferramenta que ainda não foi instalada, ele a instalará para você.)

A empresa por trás do npm também mantém o repositório <https://npmjs.com/package>, que contém centenas de milhares de pacotes de código aberto. Mas você não precisa usar o gerenciador de pacotes npm para acessar este repositório de pacotes. As alternativas incluem fios e pnpm.

17.5 Pacote de código

Se você estiver escrevendo um grande programa JavaScript para ser executado em navegadores da web, provavelmente desejará usar uma ferramenta de empacotamento de código, especialmente se você

Use bibliotecas externas que são entregues como módulos. Os desenvolvedores da Web usam os módulos ES6 (§10.3) há anos, desde muito antes de as palavras-chave de importação e exportação serem suportadas na web. Para fazer isso, os programadores usam uma ferramenta de empacotamento de código que começa no ponto de entrada principal (ou pontos de entrada) do programa e segue a árvore de diretivas de importação para encontrar todos os módulos dos quais o programa depende. Em seguida, ele combina todos esses arquivos de módulo individuais em um único pacote de código JavaScript e reescreve as diretivas de importação e exportação para fazer o código funcionar neste novo formato. O resultado é um único arquivo de código que pode ser carregado em um navegador da Web que não suporta módulos.

Os módulos ES6 são quase universalmente suportados por navegadores da web hoje, mas os desenvolvedores da web ainda tendem a usar empacotadores de código, pelo menos ao liberar código de produção. Os desenvolvedores acham que a experiência do usuário é melhor quando um único pacote de código de tamanho médio é carregado quando um usuário visita um site pela primeira vez do que quando muitos módulos pequenos são carregados.

NOTA

O desempenho da Web é um tópico notoriamente complicado e há muitas variáveis a serem consideradas, incluindo melhorias contínuas por fornecedores de navegadores, portanto, a única maneira de ter certeza da maneira mais rápida de carregar seu código é testando minuciosamente e medindo com cuidado. Lembre-se de que há uma variável que está completamente sob seu controle: o tamanho do código. Menos código JavaScript sempre será carregado e executado mais rápido do que mais código JavaScript!

Existem várias boas ferramentas de empacotamento JavaScript disponíveis. Os empacotadores comumente usados incluem [webpack](#), [Rollup](#) e [Parcel](#). As características básicas dos empacotadores são mais ou menos as mesmas e são

diferenciados com base em quão configuráveis eles são ou quão fáceis de usar. O Webpack existe há muito tempo, possui um grande ecossistema de plug-ins, é altamente configurável e pode suportar bibliotecas não modulares mais antigas. Mas também pode ser complexo e difícil de configurar. No outro extremo do espectro está o Parcel, que pretende ser uma alternativa de configuração zero que simplesmente faz a coisa certa.

Além de realizar o empacotamento básico, as ferramentas de empacotamento também podem fornecer alguns recursos adicionais:

- Alguns programas têm mais de um ponto de entrada. Um aplicativo da web com várias páginas, por exemplo, pode ser escrito com um ponto de entrada diferente para cada página. Os empacotadores geralmente permitem que você crie um pacote por ponto de entrada ou crie um único pacote que suporte vários pontos de entrada. Os programas podem
- usar import() em sua forma funcional (§10.3.6) em vez de sua forma estática para carregar dinamicamente os módulos quando eles são realmente necessários, em vez de carregá-los estaticamente no momento da inicialização do programa. Fazer isso geralmente é uma boa maneira de melhorar o tempo de inicialização do seu programa. As ferramentas de empacotamento que suportam import() podem ser capazes de produzir vários pacotes de saída: um para carregar no momento da inicialização e um ou mais que são carregados dinamicamente quando necessário. Isso pode funcionar bem se houver apenas algumas chamadas para import() em seu programa e elas carregarem módulos com conjuntos de dependências relativamente disjuntos. Se os módulos carregados dinamicamente compartilham dependências, torna-se complicado descobrir quantos pacotes produzir, e é provável que você tenha que configurar manualmente seu empacotador para resolver isso. Os empacotadores geralmente podem gerar um arquivo de mapa de origem que define um mapeamento entre as linhas de código no pacote e o

linhas correspondentes nos arquivos de origem originais. Isso permite que as ferramentas de desenvolvedor do navegador exibam automaticamente erros de JavaScript em seus locais originais separados.

- Às vezes, quando você importa um módulo para o seu programa, você usa apenas alguns de seus recursos. Uma boa ferramenta de empacotamento pode analisar o código para determinar quais partes não são utilizadas e podem ser omitidas dos pacotes. Esse recurso atende pelo nome caprichoso de "sacudir árvores". Os empacotadores normalmente têm uma arquitetura baseada em plug-ins e suportam plug-ins que permitem importar e agrupar "módulos" que não são realmente arquivos de código JavaScript. Suponha que seu programa inclua uma grande estrutura de dados compatível com JSON. Os empacotadores de código podem ser configurados para permitir que você move essa estrutura de dados para um arquivo JSON separado e, em seguida, importe-o para o seu programa com uma declaração como `import widgets from "./big-widget-list.json"`. Da mesma forma, os desenvolvedores da Web que incorporam CSS em seus programas JavaScript podem usar plug-ins de empacotamento que lhes permitem importar arquivos CSS com uma diretiva de importação. Observe, no entanto, que se você importar qualquer coisa que não seja um arquivo JavaScript, estará usando uma extensão JavaScript não padrão e tornando seu código dependente da ferramenta de empacotamento. Em uma linguagem como
- JavaScript que não requer compilação, executar uma ferramenta de empacotamento parece uma etapa de compilação, e é frustrante ter que executar um empacotador após cada edição de código antes de executar o código em seu navegador. Os empacotadores normalmente suportam observadores do sistema de arquivos que detectam qualquer arquivo em um diretório de projeto e regeneram automaticamente os pacotes necessários. Com esse recurso em vigor normalmente você pode salvar seu código e recarregar imediatamente a janela do navegador da Web para experimentá-lo. Alguns empacotadores também suportam um modo de "substituição de módulo quente"

Para desenvolvedores em que cada vez que um pacote é regenerado, ele é carregado automaticamente no navegador. Quando isso funciona, é uma experiência mágica para os desenvolvedores, mas existem alguns truques acontecendo sob o capô para fazê-lo funcionar, e não é adequado para todos os projetos.

17.6 Transpilação com Babel

Babel é uma ferramenta que compila JavaScript escrito usando recursos de linguagem moderna em JavaScript que não usa esses recursos de linguagem moderna. Por compilar JavaScript para JavaScript, o Babel às vezes é chamado de "transpilador". O Babel foi criado para que os desenvolvedores da Web pudessem usar os novos recursos de linguagem do ES6 e posteriores, enquanto ainda visavam navegadores da Web que suportavam apenas o ES5.

Recursos de linguagem, como o operador de exponenciação `**` e as funções de seta, podem ser transformados com relativa facilidade em expressões `Math.pow()` e `function`. Outros recursos de linguagem, como a palavra-chave `class`, exigem transformações muito mais complexas e, em geral, a saída de código do Babel não deve ser legível por humanos. Como as ferramentas do bundler, no entanto, o Babel pode produzir mapas de origem que mapeiam locais de código transformados de volta aos seus locais de origem originais, e isso ajuda drasticamente ao trabalhar com código transformado.

Os fornecedores de navegadores estão fazendo um trabalho melhor em acompanhar a evolução da linguagem JavaScript, e há muito menos necessidade hoje em dia de compilar funções de seta e declarações de classe. O Babel ainda pode ajudar quando você deseja usar os recursos mais recentes, como separadores de sublinhado em literais numéricos.

Como a maioria das outras ferramentas descritas neste capítulo, você pode instalar o Babel com npm e executá-lo com npx. O Babel lê um arquivo .babelrcconfiguration que informa como você gostaria que seu código JavaScript fosse transformado. Babel define "predefinições" que você pode escolher dependendo de quais extensões de idioma você deseja usar e quanto agressivamente você deseja transformar os recursos de linguagem padrão. Uma das predefinições interessantes do Babel é para compactação de código por minificação (remoção de comentários e espaços em branco, renomeação de variáveis e assim por diante).

Se você usa o Babel e uma ferramenta de agrupamento de código, pode configurar o empacotador de código para executar automaticamente o Babel em seus arquivos JavaScript à medida que ele cria o pacote para você. Nesse caso, essa pode ser uma opção conveniente porque simplifica o processo de produção de código executável. O Webpack, por exemplo, suporta um módulo "babel-loader" que você pode instalar e configurar para executar o Babel em cada módulo JavaScript à medida que ele é empacotado.

Embora haja menos necessidade de transformar a linguagem JavaScript principal hoje, o Babel ainda é comumente usado para dar suporte a extensões não padrão para a linguagem, e descreveremos duas dessas extensões de linguagem nas seções a seguir.

17.7 JSX: Expressões de marcação em JavaScript

JSX é uma extensão do JavaScript principal que usa sintaxe no estilo HTML para definir uma árvore de elementos. O JSX está mais intimamente associado ao Reactframework para interfaces de usuário na web. No React, as árvores de elementos definidos com JSX são finalmente renderizados em um navegador da web como HTML. Mesmo que você não tenha planos de usar o React por conta própria, é

popularidade significa que é provável que você veja código que usa JSX. Esta seção explica o que você precisa saber para entendê-lo. (Esta seção é sobre a extensão da linguagem JSX, não sobre o React, e explica apenas o suficiente do React para fornecer contexto para a sintaxe JSX.)

Você pode pensar em um elemento JSX como um novo tipo de sintaxe de expressão JavaScript. Os literais de string JavaScript são delimitados com aspas e os literais de expressão regular são delimitados com barras. Da mesma forma, os literais de expressão JSX são delimitados com colchetes angulares. Aqui está um muito simples:

```
deixe linha = <hr/>;
```

Se você usa JSX, precisará usar o Babel (ou uma ferramenta semelhante) para compilar expressões JSX em JavaScript regular. A transformação é simples o suficiente para que alguns desenvolvedores optem por usar o React sem usar o JSX. O Babel transforma a expressão JSX nesta instrução de atribuição em uma chamada de função simples:

```
let linha = React.createElement("hr", null);
```

A sintaxe JSX é semelhante ao HTML e, como os elementos HTML, os elementos React podem ter atributos como estes:

```
let image = ;
```

Quando um elemento tem um ou mais atributos, eles se tornam propriedades de um objeto passado como o segundo argumento para createElement():

```
let imagem = React.createElement("img", {  
    fnt: "logo.png",
```

```
        alt: "O logotipo  
JSX",hidden: true});
```

Como os elementos HTML, os elementos JSX podem ter strings e outros elementos como filhos. Assim como os operadores aritméticos do JavaScript podem ser usados para escrever expressões aritméticas de complexidade arbitrária, os elementos JSX também podem ser aninhados arbitrariamente profundamente para criar árvores de elementos:

```
barra lateral fácil =  
  <div className="sidebar">  
    <h1>Título</h1><hr/><p>Este é o  
    conteúdo da barra lateral</p></div>;
```

As expressões regulares de chamada de função JavaScript também podem ser aninhadas arbitrariamente profundamente, e essas expressões JSX aninhadas se traduzem em um conjunto de chamadas createElement() aninhadas. Quando um elemento JSX tem filhos, esses filhos (que normalmente são strings e outros elementos JSX) são passados como o terceiro e os argumentos subsequentes:

```
let barra lateral = React.createElement(  
  "div", { className: "sidebar"}, Este chamado externo  
cria um <div>  
    React.createElement("h1", nulo, Este é o primeiro  
filho do <div>  
                      "Título"), e seu próprio primeiro  
criança.  
  React.createElement("hr", null), // O segundo filho do  
  <div/>.  
  React.createElement("p", nulo, E o terceiro filho.  
                    "Este é o conteúdo da barra lateral"));
```

O valor retornado por `React.createElement()` é um objeto JavaScript comum que é usado pelo React para renderizar a saída em uma janela do navegador. Como esta seção é sobre a sintaxe JSX e não sobre React, não entraremos em detalhes sobre os objetos `ElementObject` retornados ou o processo de renderização. Vale a pena notar que você pode configurar o Babel para compilar elementos JSX para invocações de uma função diferente, portanto, se você acha que a sintaxe JSX seria uma maneira útil de expressar outros tipos de estruturas de dados aninhadas, você pode adotá-la para seus próprios usos não React.

Um recurso importante da sintaxe JSX é que você pode incorporar expressões regular JavaScript em expressões JSX. Dentro de uma expressão JSX, o texto entre chaves é interpretado como JavaScript simples. Por exemplo:

```
function sidebar(className, title, content, drawLine=true) {  
  retornar (  
    <div className={className}>  
      <h1>{title}</h1>{ drawLine  
      && <hr/> }<p>{content}</p>  
    </div>);}
```

A função `sidebar()` retorna um elemento JSX. São necessários quatro argumentos que ele usa dentro do elemento JSX. A sintaxe de chaves pode lembrá-lo de literais de modelo que usam `${}` para incluir expressões JavaScript em cadeias de caracteres. Como sabemos que as expressões JSX são compiladas em invocações de função, não deve ser surpreendente que

expressões JavaScript arbitrárias podem ser incluídas porque functioninvocations também podem ser escritas com expressões arbitrárias. Este código de exemplo é traduzido pelo Babel para o seguinte:

```
function sidebar(className, title, content, drawLine=true) {  
  return React.createElement("div", { className: className },  
    React.createElement("h1", null,  
      título),  
      drawLine &&  
    React.createElement("hr", null),  
      React.createElement("p", null,  
        conteúdo)  
);}
```

Este código é fácil de ler e entender: as chaves desapareceram e o código resultante passa os parâmetros de função de entrada para `React.createElement()` de maneira natural. Observe o truque legal que fizemos aqui com o parâmetro `drawLine` e o operador `&&` de curto-circuito. Se você chamar `sidebar()` com apenas três argumentos, o padrão de `drawLine` será `true` e o quarto argumento para a chamada `createElement()` externa será o `<hr>` elemento. Mas se você passar `false` como o quarto argumento para `sidebar()`, então o quarto argumento para a chamada `createElement()` externa será avaliado como `false` e nenhum `<hr>` elemento será criado. Esse uso do operador `&&` é um idioma comum no JSX para incluir ou excluir condicionalmente um elemento filho, dependendo do valor de alguma outra expressão. (Esta expressão funciona com o React porque o React simplesmente ignora os filhos que são falsos ou nulos e não produz nenhuma saída para eles.)

Ao usar expressões JavaScript em expressões JSX, você não está limitado a valores simples, como a string e os valores booleanos no

exemplo anterior. Qualquer valor JavaScript é permitido. Na verdade, é bastante comum na programação React usar objetos, arrays e funções. Considere a seguinte função, por exemplo:

Dado um array de strings e uma função de retorno de chamada retorna um elemento JSX// representando uma lista HTML com um array de elementos como seu filho.function list(items, callback) {

```
    retornar (
      <ul style={ {padding:10, border:"solid red 4px"} }>
        {items.map((item,index) => {
          <li onClick={() => callback(index)} key={index}>
{item}</li>
        ))}
      </ul>);}
```

Essa função usa um literal de objeto como o valor do atributo de estilo no elemento. (Observe que chaves duplas são necessárias aqui.) O elemento tem um único filho, mas o valor desse filho é uma matriz. A matriz filho é a matriz criada usando a função map() na matriz de entrada para criar uma matriz de elementos. (Isso funciona com o React porque a biblioteca React nivela os filhos de um elemento quando os renderiza. Um elemento com um filho de matriz é o mesmo que esse elemento com cada um desses elementos de matriz como filhos.) Por fim, observe que cada um dos elementos aninhados tem um atributo do manipulador onClick cujo valor é uma função de seta. O código JSX é compilado para o seguinte código JavaScript puro (que formatei com Prettier):

lista de funções (itens, retorno de chamada) {

```
return React.createElement(  
  "ul", { style: { padding: 10, border: "vermelho  
 sólido 4px" } }, items.map((item, index) =>  
  
    React.createElement(  
      "li", { onClick: () => callback(index), chave: índice  
    }, item));}
```

Um outro uso de expressões de objeto no JSX é com o operador de propagação de objeto (§6.10.4) para especificar vários atributos de uma só vez. Suponha que você esteja escrevendo muitas expressões JSX que repetem um conjunto comum de atributos. Você pode simplificar suas expressões definindo os atributos como propriedades de um objeto e "espalhando-os" em seus elementos JSX:

```
seja hebraico = { lang: "ele", dir: "rtl" }; Especifique o  
idioma e a direção shalom = <span className="emphasis" {...  
hebraico}⟩שלום/>
```

Babel compila isso para usar uma função `_extends()` (omitida aqui) que combina esse atributo `className` com os atributos contidos no objeto `hebraico`:

```
let shalom = React.createElement("span",  
  _extends({className:  
    "ênfase"}, hebraico),  
  "\u05E9\u05DC\u05D5\u05DD");
```

Finalmente, há mais uma característica importante do JSX que não temos

coberto ainda. Como você viu, todos os elementos JSX começam com um identificador imediatamente após o colchete angular de abertura. Se a primeira letra deste identificador for minúscula (como foi em todos os exemplos aqui), então o identificador é passado para createElement() como uma string. Mas se a primeira letra do identificador estiver em maiúsculas, ele será tratado como um actualidentifier e é o valor JavaScript desse identificador que é passado o primeiro argumento para createElement(). Isso significa que a expressão <Math/> JSX é compilada para o código JavaScript que passa o objeto Math global para React.createElement().

Para o React, essa capacidade de passar valores não string como o primeiro argumento tocreateElement() permite a criação de componentes. Acomponent é uma maneira de escrever uma expressão JSX simples (com um nome de componente em maiúsculas) que representa uma expressão mais complexa (usando nomes de tags HTML em minúsculas).

A maneira mais simples de definir um novo componente no React é escrever uma função que recebe um "objeto props" como argumento e retorna uma expressão JSX. Um objeto props é simplesmente um objeto JavaScript que representa valores de atributos, como os objetos que são passados como o segundo argumento createElement(). Aqui, por exemplo, está outra versão da função oursidebar():

```
function Barra lateral (props) {  
  retornar (  
    <div>  
      <h1>{props.title}</h1>{  
        props.drawLine && <hr/> }<p>  
        {props.content}</p></div>);
```

```
}
```

Esta nova função Sidebar() é muito parecida com a função sidebar() anterior. Mas este tem um nome que começa com uma letra maiúscula e recebe um único argumento de objeto em vez de argumentos separados. Isso o torna um componente React e significa que ele pode ser usado no lugar de um nome de tag HTML em expressões JSX:

```
barra lateral fácil = <Barra lateral  
title="Algo rápido"content="Algo sábio"/>;
```

Este <Sidebar> elemento é compilado assim:

```
let barra lateral = React.createElement(Barra lateral, {  
  title: "Algo  
rápido", content: "Algo  
sábio"});
```

É uma expressão JSX simples, mas quando o React a renderiza, ele passará o segundo argumento (o objeto Props) para o primeiro argumento (a função Sidebar()) e usará a expressão JSX retornada por essa função no lugar da <Sidebar> expressão.

17.8 Verificação de tipo com fluxo

Flow é uma extensão de linguagem que permite anotar seu código JavaScript com informações de tipo e uma ferramenta para verificar seu código JavaScript (anotado e não anotado) em busca de erros de tipo. Para usar o Flow, você começa a escrever código usando a extensão de linguagem Flow para adicionar anotações de tipo. Em seguida, execute a ferramenta Flow para analisar seu código e relatar erros de tipo. Depois de corrigir os erros e estar pronto para

executar o código, você usa o Babel (talvez automaticamente como parte do processo de agrupamento de código) para remover as anotações do tipo Flow do seu código. (Uma das coisas boas sobre a extensão da linguagem Flow é que não há nenhuma sintaxe nova que o Flow precise compilar ou transformar. Você usa a extensão de linguagem Flow para adicionar anotações ao código, e tudo o que o Babel precisa fazer é remover essas anotações para retornar seu código ao JavaScript padrão.)

TEXTO DATILOGRAFADO VERSUS FLUXO

O TypeScript é uma alternativa muito popular ao Flow. TypeScript é uma extensão do JavaScript que adiciona tipos, bem como outros recursos de linguagem. O compilador TypeScript "tsc" compila programas TypeScript em programas JavaScript e, no processo, analisa-os e relata erros de tipo da mesma forma que o Flow. tsc não é um plugin Babel: é seu próprio compilador autônomo.

As anotações de tipo simples no TypeScript geralmente são gravadas de forma idêntica às mesmas anotações no Flow. Para digitação mais avançada, a sintaxe das duas extensões diverge, mas a intenção e o valor das duas extensões são os mesmos. Meu objetivo nesta seção é explicar os benefícios das anotações de tipo e da análise estática de código. Farei isso com exemplos baseados no Flow, mas tudo o que foi demonstrado aqui também pode ser alcançado com o TypeScript com mudanças de sintaxe relativamente simples.

O TypeScript foi lançado em 2012, antes do ES6, quando o JavaScript não tinha uma palavra-chave de classe ou loop for/of ou módulos ou promessas. Flow é uma extensão de linguagem estreita que adiciona typeannotations ao JavaScript e nada mais. O TypeScript, por outro lado, foi projetado como uma nova linguagem. Como o próprio nome indica, adicionar tipos ao JavaScript é o objetivo principal do TypeScript e é a razão pela qual as pessoas o usam hoje. Mas os tipos não são o único recurso que o TypeScript adiciona ao JavaScript: a linguagem TypeScript tem palavras-chave enum e namespace que simplesmente não existem em JavaScript. Em 2020, o TypeScript tem melhor integração com IDEs e editores de código (particularmente VSCode, que, como o TypeScript, é da Microsoft) do que o Flow.

Em última análise, este é um livro sobre JavaScript, e estou cobrindo o Flow aqui em vez do TypeScript porque não quero tirar o foco do JavaScript. Mas tudo o que você aprende aqui sobre como adicionar tipos a JavaScript será útil para você se decidir adotar o TypeScript para seus projetos.

Usar o Flow requer comprometimento, mas descobri que, para projetos de médio e grande porte, o esforço extra vale a pena. Leva mais tempo para adicionar anotações de tipo ao seu código, executar o Flow sempre que você editar o código e corrigir os erros de tipo relatados. Mas, em troca, o Flow irá

impõe uma boa disciplina de codificação e não permitirá que você corte custos que podem levar a bugs. Quando trabalhei em projetos que usam o Flow, fiquei impressionado com o número de erros encontrados em meu próprio código. Ser capaz de corrigir esses problemas antes que eles se tornem bugs é uma ótima sensação e me dá confiança extra de que meu código está correto.

Quando comecei a usar o Flow, descobri que às vezes era difícil entender por que ele estava reclamando do meu código. Com alguma prática, porém, passei a entender suas mensagens de erro e descobri que geralmente era fácil fazer pequenas alterações no meu código para torná-lo mais seguro e satisfazer o Flow. Eu não recomendo usar o Flow se você ainda sentir que está aprendendo o próprio JavaScript. Mas quando você estiver confiante com a linguagem, adicionar o Flow aos seus projetos JavaScript o levará a levar suas habilidades de programação para o próximo nível. E é por isso que estou dedicando a última seção deste livro a um tutorial do Flow: porque aprender sobre sistemas de tipos JavaScript oferece um vislumbre de outro nível, ou outro estilo, de programação.

Esta seção é um tutorial e não tenta cobrir o FlowComprehensively. Se você decidir experimentar o Flow, quase certamente acabará gastando tempo lendo a documentação em <https://flow.org>. Por outro lado, você não precisa dominar o sistema de tipos Flow antes de começar a usá-lo na prática em seus projetos: os usos simples do Flow descritos aqui o levarão longe.

17.8.1 Instalando e executando o Flow

Como as outras ferramentas descritas neste capítulo, você pode instalar a ferramenta de verificação de tipo de fluxo usando um gerenciador de pacotes, com um comando como

npm install -g flow-bin ou npm install --save-dev flow-bin. Se você instalar a ferramenta globalmente com -g, poderá executá-la com fluxo. E se você instalá-lo localmente em seu projeto com --save-dev, poderá executá-lo com o fluxo npx. Antes de usar o Flow para fazer verificação de tipos, execute-o pela primeira vez como flow --init no diretório raiz do seu projeto para criar um arquivo de configuração .flowconfig. Talvez você nunca precise adicionar nada a esse arquivo, mas o Flow precisa dele para saber onde está a raiz do seu projeto.

Quando você executa o Flow, ele encontrará todo o código-fonte JavaScript em seu projeto, mas só relatará erros de tipo para os arquivos que "optaram" por verificar o tipo adicionando um comentário // @flow na parte superior do arquivo. Esse comportamento de aceitação é importante porque significa que você pode adotar o Flow para projetos existentes e, em seguida, começar a converter seu código um arquivo de cada vez, sem ser incomodado por erros e avisos em arquivos que ainda não foram convertidos.

O Flow pode encontrar erros em seu código, mesmo que tudo o que você faça seja aceitar com um comentário // @flow. Mesmo que você não use a extensão de linguagem Flow e não adicione nenhuma anotação de tipo ao seu código, a ferramenta Typechecker Flow ainda pode fazer inferências sobre os valores em seu programa e alertá-lo quando você os usa de forma inconsistente.

Considere a seguinte mensagem de erro do Flow:

Erro

variableReassignment.js:6:3

Não é possível atribuir 1 a i.r porque:
• A propriedade R está ausente no número [1].

```
2| let i = { r: 0, i: 1 }; // 0 número complexo 0+1i[1] 3|
for(i = 0; i < 10; i++) { // Opa! A variável de loop
substitui i
4|     console.log(i);
5| }6| i.r
= 1;                                     0 Flow detecta o erro
aqui
```

Nesse caso, declaramos a variável `i` e atribuímos um objeto a ela.

Então usamos `i` novamente como uma variável de loop, substituindo o objeto. O fluxo percebe isso e sinaliza um erro quando tentamos usar `i` como se ainda contivesse um objeto. (Uma correção simples seria escrever `for(let i = 0;` tornando a variável de loop local para o loop.)

Aqui está outro erro que o Flow detecta mesmo sem anotações de tipo:

```
Error.....  
size.js:3:14
```

Não é possível obter `x.length` porque a propriedade `length` está ausente em `Number[1]`.

```
1| // @flow2| tamanho da
função(x) {3| return
x.comprimento; 4| }[1] 5|
seja s = tamanho(1000);
```

O Flow vê que a função `size()` recebe um único argumento. Ele não sabe o tipo desse argumento, mas pode ver que se espera que o argumento tenha uma propriedade `length`. Quando ele vê essa função `size()` sendo chamada com um argumento numérico, ele sinaliza corretamente isso como um erro porque os números não têm propriedades de comprimento.

17.8.2 Usando anotações de tipo

Ao declarar uma variável JavaScript, você pode adicionar uma anotação de tipo de fluxo a ela seguindo o nome da variável com dois-pontos e o tipo:

```
let message: string = "Olá mundo";
let flag: boolean = false; let n:
número = 42;
```

O Flow conheceria os tipos dessas variáveis mesmo que você não as anotasse: ele pode ver quais valores você atribui a cada variável e acompanha isso. No entanto, se você adicionar anotações de tipo, o Flow saberá o tipo da variável e que você expressou a intenção de que a variável seja sempre desse tipo. Portanto, se você usar a anotação de tipo, o Flow sinalizará um erro se você atribuir um valor de um tipo diferente a essa variável. As anotações de tipo para variáveis também são particularmente úteis se você tende a declarar todas as suas variáveis no topo de uma função antes de serem usadas.

As anotações de tipo para argumentos de função são como anotações para variáveis: siga o nome do argumento da função com dois pontos e o nome do tipo. Ao anotar uma função, você normalmente também adiciona uma anotação para o tipo de retorno da função. Isso vai entre oparênteses fechados e a chave aberta do corpo da função. As funções que não retornam nada usam o tipo Flow void.

No exemplo anterior, definimos uma função size() que esperava um argumento com uma propriedade length. Veja como poderíamos alterar essa função para especificar explicitamente que ela espera um argumento de string e retorna um número. Observe que o Flow agora sinaliza um erro se passarmos um array para a função, mesmo que a função funcione nesse caso:

```
Error.....  
size2.js:5:18
```

Não é possível chamar size com array literal associado a s porque arrayliteral [1]é incompatível com string [2].

```
[2] 2| tamanho(s) da função: string): number {  
  3|   retornar s.length;  
4| }[1] 5|  
  console.log(tamanho([1,2,3]));
```

Usar anotações de tipo com funções de seta também é possível, embora possa transformar essa sintaxe normalmente sucinta em algo mais detalhado:

```
const tamanho = (s: string): número => s.comprimento;
```

Uma coisa importante a entender sobre o Flow é que o JavaScript value null tem o tipo Flow null e o JavaScript value undefined tem o tipo Flow void. Mas nenhum desses valores é membro de qualquer outro tipo (a menos que você o adicione explicitamente). Se você declarar um parâmetro de função como uma string, ele deve ser uma string, e é um erro passar null ou passar undefined ou omitir o argumento (que é basicamente a mesma coisa que passar undefined):

```
Error.....  
size3.js:3:18
```

Não é possível chamar size com null vinculado a s porque null [1] é incompatível com a cadeia de caracteres [2].

```
1| // @flow[2] 2| const tamanho = (s: string):  
número => s.comprimento; [1] 3|  
  console.log(tamanho(nulo));
```

Se você quiser permitir null e undefined como valores legais para um argumento de variável ou função, basta prefixar o tipo com um ponto de interrogação. Por exemplo, use ?string ou ?number em vez de string ornumber. Se alterarmos nossa função size() para esperar um argumento do tipo ?string, o Flow não reclamará quando passarmos null para a função. Mas agora tem algo mais do que reclamar:

```
Error.....  
size4.js:3:14
```

Não é possível obter s.length porque o comprimento da propriedade está ausente innull ou undefined [1].

```
1| // @flow[1] 2| tamanho(s) da função: ?  
  string): number {  
    3|     retornar s.length;  
    4| }5|  
    console.log(tamanho(nulo));
```

O que o Flow está nos dizendo aqui é que não é seguro escrever s.length porque, neste local em nosso código, s pode ser nulo ou indefinido, e esses valores não têm propriedades de comprimento. É aqui que o Flow garante que não cortamos nenhum canto. Se um valor for nulo, o Flow insistirá que verifiquemos esse caso antes de fazermos qualquer coisa que dependa do valor não ser nulo.

Nesse caso, podemos corrigir o problema alterando o corpo da função da seguinte maneira:

```
tamanho(s) da função: ?string): number {  
  Neste ponto do código, s pode ser uma cadeia de caracteres  
  ou null ou indefinido.  
  if (s === null || s === indefinido) {
```

```
Neste bloco, o Flow sabe que s é nulo ou
undefined. return
-1;
} else {
E neste bloco, o Flow sabe que s é uma string. return
s.length;}}
```

Quando a função é chamada pela primeira vez, o parâmetro pode ter mais de um tipo. Mas, ao adicionar código de verificação de tipo, criamos um bloco dentro do código em que o Flow sabe com certeza que o parâmetro é uma string. Quando usamos s.length dentro desse bloco, o Flow não reclama.

Observe que o Flow não exige que você escreva um código detalhado como este. Flow também ficaria satisfeito se apenas substituíssemos o corpo da função size() pelo retorno s ? s.comprimento : -1;

A sintaxe de fluxo permite que um ponto de interrogação antes de qualquer especificação de tipo indique que, além do tipo especificado, null e undefined também são permitidos. Os pontos de interrogação também podem aparecer após um nome de parâmetro para indicar que o parâmetro em si é opcional. Então, se mudássemos a declaração do parâmetro s de s: ? string para s? :string, isso significaria que não há problema em chamar size() sem argumentos (ou com o valor indefinido, que é o mesmo que omiti-lo), mas se o chamarmos com um parâmetro diferente de indefinido, então esse parâmetro deve ser uma string. Nesse caso, null não é um valor legal.

Até agora, discutimos os tipos primitivos string, number, boolean, null e void e demonstramos como você pode usá-los com declarações de variáveis, parâmetros de função e valores de retorno de função. As subseções a seguir descrevem alguns tipos mais complexos

suportado pelo Flow.

17.8.3 Tipos de classe

Além dos tipos primitivos que o Flow conhece, ele também conhece todas as classes integradas do JavaScript e permite que você use classname como tipos. A função a seguir, por exemplo, usa typeannotations para indicar que ela deve ser invocada com um objeto Date e um objeto RegExp:

```
@flow// Retorna true se a representação ISO da data
especificada// corresponder ao padrão especificado, ou
false caso contrário.// Por exemplo: const
isTodayChristmas = dateMatches(new Date(),/^d{4}-12-
25T/); export function dateMatches(d: Date, p: RegExp):
boolean {
    return p.test(d.toISOString());}
```

Se você definir suas próprias classes com a palavra-chave class, essas classes se tornarão automaticamente tipos de fluxo válidos. Para fazer isso funcionar, no entanto, o Flow exige que você use anotações de tipo na classe. Em particular, cada propriedade da classe deve ter seu tipo declarado. Aqui está uma classe de número complexo simples que demonstra isso:

```
@flowexport classe padrão
Complex {
    O fluxo requer uma sintaxe de classe estendida que inclui
    anotações de tipo
        para cada uma das propriedades usadas pelo
        número class.i;; r: número; estático i:
        Complexo;
}

construtor(r: número, i:número) {
```

```
Todas as propriedades inicializadas pelo construtor devem
tem tipo de fluxo
anotações acima.this.r =
r; this.i = i;}
```

```
add(that: Complexo) {
return new Complex(this.r + that.r, this.i + that.i);}}
```

*Esta tarefa não seria permitida pelo Flow se não houvesse
uma a// anotação de tipo para i dentro da classe.
Complexo.i = novo Complexo(0,1);*

17.8.4 Tipos de objeto

O tipo de fluxo para descrever um objeto se parece muito com um literal de objeto, exceto que os valores de propriedade são substituídos por tipos de propriedade. Aqui, por exemplo, está uma função que espera um objeto com propriedades numéricas x e y:

```
@flow// Dado um objeto com propriedades numéricas x e y,
retorna// distância da origem ao ponto (x,y) como um
número.export função padrão distance(point:
{x:number,y:number}): number {
```

```
return Math.hypot(ponto.x, ponto.y);}
```

Neste código, o texto `{x:number, y:number}` é um tipo Flow, assim como string ou Date é. Como acontece com qualquer tipo, você pode adicionar um ponto de interrogação na frente para indicar que null e undefined também devem ser permitidos.

Dentro de um tipo de objeto, você pode seguir qualquer um dos nomes de propriedade com um ponto de interrogação para indicar que essa propriedade é opcional e pode ser omitida. Por exemplo, você pode escrever o tipo para um objeto que representa um ponto 2D ou 3D assim:

```
{x: número, y: número, z?: número}
```

Se uma propriedade não estiver marcada como opcional em um tipo de objeto, ela será necessária e o Flow relatará um erro se uma propriedade apropriada não estiver presente no valor real. Normalmente, no entanto, o Flow tolera propriedades extras. Se você passasse um objeto que tivesse uma propriedade w para a função distance() acima, o Flow não reclamaria.

Se você quiser que o Flow imponha estritamente que um objeto não tenha propriedades diferentes daquelas explicitamente declaradas em seu tipo, você pode declarar um tipo de objeto exato adicionando barras verticais às chaves:

```
{| x: número, y: número |}
```

Os objetos do JavaScript às vezes são usados como dicionários ou mapas de string para valor. Quando usados dessa forma, os nomes de propriedade não são conhecidos antecipadamente e não podem ser declarados em um tipo de fluxo. Se você usar objetos dessa maneira, ainda poderá usar o Flow para descrever a estrutura de dados. Suponha que você tenha um objeto em que as propriedades são os nomes das principais cidades do mundo e os valores dessas propriedades são objetos que especificam a localização geográfica dessas cidades. Você pode declarar essa estrutura de dados assim:

```
@flowconst cityLocations : {[string]:  
{longitude:number,latitude:number}} = {
```

```
"Seattle": { longitude: 47.6062, latitude: -122.3321 },//  
TODO: se houver outras cidades importantes, adicione-as  
aqui.}; export defaultcityLocations;
```

17.8.5 Aliases de tipo

Os objetos podem ter muitas propriedades, e o tipo de fluxo que descreve esse objeto será longo e difícil de digitar. E mesmo tipos de objetos relativamente curtos podem ser confusos porque se parecem muito com literais de objetos. Uma vez que vamos além de tipos simples como number e ?string, geralmente é útil poder definir nomes para nossos Flowtypes. E, de fato, o Flow usa a palavra-chave type para fazer exatamente isso. Siga a palavra-chave type com um identificador, um sinal de igual e um tipo de fluxo. Depois de fazer isso, o identificador será um alias para o tipo. Aqui, por exemplo, é como poderíamos reescrever a função distance() da seção anterior com um Pointtype explicitamente definido:

```
@flowexport tipo  
Ponto = {  
  x: número, y:  
  número};
```

```
Dado um objeto Point retorna sua distância da função  
padrão originexport distance(point: Point): number {  
  return Math.hypot(ponto.x, ponto.y);}
```

Observe que este código exporta a função distance() e também exporta o tipo Point. Outros módulos podem usar o tipo de importação

Aponte de './distance.js' se eles quiserem usar essa definição de tipo. Lembre-se, porém, de que o tipo de importação é uma extensão Flowlanguage e não uma diretiva de importação JavaScript real. As importações e exportações de tipo são usadas pelo verificador de tipo do Flow, mas, como todas as outras extensões de linguagem do Flow, elas são removidas do código antes mesmo de serem executadas.

Por fim, vale a pena notar que, em vez de definir um nome para um tipo de objeto Flow que representa um ponto, provavelmente seria mais simples e mais limpo apenas definir uma classe Point e usar essa classe como o tipo.

17.8.6 Tipos de matriz

O tipo Flow para descrever uma matriz é um tipo composto que também inclui o tipo dos elementos da matriz. Aqui, por exemplo, está uma função que espera um array de números, e o erro que o Flow relata se você tentar chamar a função com um array que tenha elementos não numéricos:

Erro

average.js:8:16

Não é possível chamar average com o literal de matriz vinculado a dados porque a string [1] é incompatível com o número [2] no elemento de matriz.

```
[2]2| function average(data: Array<number>)
{3|let sum = 0; 4|for(let x de dados) soma +=
x;5|return sum/data.length; 6| }7|[1]8|
média([1, 2, "três"]);
```

O tipo de fluxo para uma matriz é Matriz seguido pelo tipo de elemento entre colchetes de emaranhamento. Você também pode expressar um tipo de matriz seguindo o

com colchetes de abertura e fechamento. Então, neste exemplo, poderíamos ter escrito `number[]` em vez de `Array<number>`. Eu prefiro a notação de colchete angular porque, como veremos, existem outros tipos de Flow que usam essa sintaxe de colchete angular.

A sintaxe do tipo Array mostrada funciona para arrays com um número arbitrário de elementos, todos com o mesmo tipo. O Flow tem uma sintaxe diferente para descrever o tipo de uma tupla: uma matriz com um número fixo de elementos, cada um dos quais pode ter um tipo diferente. Para expressar o tipo de uma tupla, basta escrever o tipo de cada um de seus elementos, separá-los com vírgulas e colocá-los todos entre colchetes.

Uma função que retorna um código de status HTTP e uma mensagem pode ter a seguinte aparência, por exemplo:

```
function getStatus():[número, string] {  
    return [getStatusCode(), getStatusMessage()];}
```

As funções que retornam tuplas são difíceis de trabalhar, a menos que você tenha usado uma atribuição de estruturação:

```
let [código, mensagem] = getStatus();
```

Atribuição de desestruturação, além dos recursos de aliasing de tipo do Flow, maketuples fáceis de trabalhar que você pode considerá-los como uma alternativa às classes para tipos de dados simples:

@flowexport tipo Cor = [número, número, número, número]; [r,
a, b, opacidadel

```
function gray(nível: número): Cor {  
    return [nível, nível, nível, 1];}  
  
function fade([r,g,b,a]: Cor, fator: número): Cor {  
    return [r, g, b, a/fator];}  
  
seja [r, g, b, a] = desvanecer(cinza(75), 3);
```

Agora que temos uma maneira de expressar o tipo de um array, vamos retornar à função size() anterior e modificá-la para esperar um argumento arrayargument em vez de um argumento string. Queremos que a função seja capaz de aceitar uma matriz de qualquer comprimento, portanto, um tipo de tupla não é apropriado. Mas não queremos restringir nossa função a funcionar apenas para arrays em que todos os elementos têm o mesmo tipo. A solução é o typeArray<mixed>:

```
@flowfunction tamanho(s): <mixed>Array:  
number {  
    return  
    s.length;}console.log(tamanho([1, ve  
rdadeiro,"três"]));
```

O tipo de elemento mixed indica que os elementos da matriz podem ser de qualquer tipo. Se nossa função realmente indexasse o array e tentasse usar qualquer um desses elementos, o Flow insistiria que usássemos typeofchecks ou outros testes para determinar o tipo do elemento antes de realizar qualquer operação insegura nele. (Se você estiver disposto a desistir da verificação de tipo, também poderá usar qualquer um em vez de misto: ele permite que você faça o que quiser com os valores da matriz sem garantir que os valores sejam do tipo esperado.)

17.8.7 Outros tipos parametrizados

Vimos que, quando você anota um valor como um Array, o Flow exige que você também especifique o tipo dos elementos do array dentro de anglebrackets. Esse tipo adicional é conhecido como um parâmetro de tipo, e `Array` não é a única classe JavaScript parametrizada.

A classe `Set` do JavaScript é uma coleção de elementos, como um array, e você não pode usar `Set` como um tipo por si só, mas precisa incluir um parâmetro `type` entre colchetes angulares para especificar o tipo dos valores contidos no conjunto. (Embora você possa usar misto ou qualquer um se o conjunto puder conter valores de vários tipos.) Aqui está um exemplo:

```
@flow// Retorna um conjunto de números com membros que
// são exatamente duas vezes aqueles// do conjunto de
// números de entrada.função do(s): <number>Set:<number>
Set {
```

```
let dobrado: Set<number> = new Set(); for(let n of s)
doubled.add(n * 2); retorno dobrado;}console.log(double(new
set([1,2,3])));// Imprime "Set {2, 4,6}"
```

`Map` é outro tipo parametrizado. Nesse caso, há dois typeparameters que devem ser especificados; O tipo das chaves e os tipos dos valores:

```
@flowimport digite { Color } from
"./Color.js";

let colorNames: Map<string, Color> = new Map([
  ["vermelho", [1, 0, 0, 1]],
  ["verde", [0, 1, 0, 1]],
```

```
["azul", [0, 0, 1, 1]]);
```

O Flow também permite definir parâmetros de tipo para suas próprias classes. O código a seguir define uma classe Result, mas parametriza essa classe com um tipo Error e um tipo Value. Usamos espaços reservados E e V no código para representar esses parâmetros de tipo. Quando o usuário desta classe declara uma variável do tipo Result, ele especificará os tipos reais para substituir E e V. A declaração da variável pode ter esta aparência:

```
let result: Result<TypeError, Set<string>>;
```

E aqui está como a classe parametrizada é definida:

```
@flow// Esta classe representa o resultado de uma operação
que pode// lançar um erro do tipo E ou um valor do tipo
V.exportar classe Result<E, V> {

    erro:? E;
    valor:? V;

    construtor(erro: ? E, valor: ? V) {
        this.error = erro;
        this.value = valor; }

    jogou (): ? E { return this.error;
    }retornado(): ? V { return this.value; }

    get():V {
        if (this.error) {throw this.error;} else if
        (this.value === null || this.value ===

    indefinido) {
            throw new TypeError("Erro e valor não devem
ambos sejam nulos");
```

```
    } else {
      return this.value;}}
```

}

E você pode até definir parâmetros de tipo para funções:

```
@flow// Combine os elementos de duas matrizes em uma matriz
de paresfunction zip<A,B>(a:Array<A>, b:Array<B>): Array<[?
A,?B]> {
  let resultado:Matriz<[?A,?B]> = []; seja len
  = Math.max(a.comprimento, b.comprimento);
  for(let i = 0; i < len; i++) {
    resultado.push([a[i],
    b[i]]);}resultado de retorno;}
```

```
Crie o array [[1,'a'], [2,'b'], [3,'c'],
[4,undefined]]let pairs: Array<[?number,?string]> =
zip([1,2,3,4],['a','b','c'])
```

17.8.8 Tipos somente leitura

O Flow define alguns "tipos de utilitários" parametrizados especiais que têm nomes que começam com \$. A maioria desses tipos tem casos de uso avançados que não abordaremos aqui. Mas dois deles são bastante úteis na prática. Se você tiver um tipo de objeto T e quiser fazer uma versão somente leitura desse tipo, basta escrever \$ReadOnly<T>. Da mesma forma, você pode escrever \$ReadOnlyArray<T> para descrever uma matriz somente leitura com elementos do tipo T.

A razão para usar esses tipos não é porque eles podem oferecer qualquer

garantir que um objeto ou array não possa ser modificado (consulte `Object.freeze()` em §14.2 se você quiser objetos somente leitura verdadeiros), mas porque permite que você capture bugs causados por modificações não intencionais. Se você escrever uma função que usa um objeto ou array argumento e não altera nenhuma das propriedades do objeto ou os elementos do array, poderá anotar o parâmetro da função com um dos tipos somente leitura do Flow. Se você fizer isso, o Flow relatará um erro se você esquecer e modificar acidentalmente o valor de entrada. Aqui estão dois exemplos:

```
@flowtype Ponto = {x:número,  
y:número};
```

*Esta função pega um objeto Point, mas promete não modificá-lo! distância da função(p: \$ReadOnly<Point>): número {
 return Math.hypot(p.x, p.y);}*

```
seja p: Ponto = {x:3, y:4};  
distância (p) // => 5
```

*Esta função recebe uma matriz de números que não modificará average da função (dados: \$ReadOnlyArray<number>): número {
 seja soma = 0; for(let i = 0; i < data.length; i++)
 soma += data[i]; return sum/data.length;}*

```
let data: Array<number> =  
[1,2,3,4,5]; média(dados) // => 3
```

17.8.9 Tipos de função

Vimos como adicionar anotações de tipo para especificar os tipos de um

parâmetros da função e seu tipo de retorno. Mas quando um dos parâmetros de uma função é em si uma função, precisamos ser capazes de especificar o tipo desse parâmetro de função.

Para expressar o tipo de uma função com Flow, escreva os tipos de cada parâmetro, separe-os com vírgulas, coloque-os entre parênteses e, em seguida, siga com uma seta e digite o tipo de retorno da função.

Aqui está um exemplo de função que espera receber uma função de retorno de chamada. Observe como definimos um alias de tipo para o tipo da função de retorno de chamada:

```
@flow// O tipo da função de retorno de chamada usada em
fetchText()belowexport type FetchTextCallback = (? Erro, ?
número, ?string) =>void;

exportar função padrão fetchText(url: string,
callback:FetchTextCallback) {
    let status =
    null; fetch(url)
        .then(resposta => {
            status = resposta.status; return
            response.text()}).then(body =>
            {callback(null, status,
            body);}). catch(erro => {

                callback(erro, status, nulo);});

    }
```

17.8.10 Tipos de União

Vamos voltar mais uma vez para a função size(). Realmente não faz sentido ter uma função que não faz nada além de retornar o comprimento de uma matriz. As matrizes têm uma propriedade de comprimento perfeitamente boa para isso. Mas size() pode ser útil se puder pegar qualquer tipo de objeto de coleção (um array ou um Set ou um Map) e retornar o número de elementos na coleção. Em JavaScript normal não tipado, seria fácil escrever uma função size() como essa. Com o Flow, precisamos de uma maneira de expressar um tipo que permita matrizes, conjuntos e mapas, mas não permita valores de nenhum outro tipo.

O Flow chama tipos como este Union types e permite que você os expresse simplesmente listando os tipos desejados e separando-os com caracteres de barra vertical:

```
@flowfunction
size(collection:Array<mixed>|Conjunto<mixed>|Mapa<
misturado,misturado>): número {
    if (Array.isArray(coleção)) {
        return collection.length;}
    else {
        return coleção.size;} }tamanho([1,verdadeiro,"três"]) +
tamanho(novo Set([verdadeiro,falso])) // => 5
```

Os tipos de união podem ser lidos usando a palavra "ou" — "uma matriz ou um Set ou aMap" — portanto, o fato de essa sintaxe Flow usar o mesmo caractere de barra vertical que os operadores OR do JavaScript é intencional.

Vimos anteriormente que colocar um ponto de interrogação antes de um tipo permite valores nulos e indefinidos. E agora você pode ver que um ? prefixo é

simplesmente um atalho para adicionar um sufixo |null|void a um tipo.

Em geral, quando você anota um valor com um tipo Union, o Flow não permite que você use esse valor até que você tenha feito testes suficientes para descobrir qual é o tipo do valor real. No exemplo size() que acabamos de ver, precisamos verificar explicitamente se o argumento é um array antes de tentarmos acessar a propriedade length do argumento. Observe que não precisamos distinguir um argumento Set de um argumento Map, no entanto: ambas as classes definem uma propriedade size, portanto, o código na cláusula else é seguro, desde que o argumento não seja um array.

17.8.11 Tipos enumerados e uniões discriminadas

O Flow permite que você use literais primitivos como tipos que consistem em um único valor. Se você escrever let x:3;, o Flow não permitirá que você atribua nenhum valor a essa variável além de 3. Muitas vezes não é útil definir tipos que têm apenas um único membro, mas uma união de tipos literais pode ser útil. Você provavelmente pode imaginar um uso para tipos como estes, por exemplo:

```
digite Resposta = "sim" | "não";
digite Dígito = 0|1|2|3|4|5|6|7|8|9;
```

Se você usar tipos compostos de literais, precisará entender que apenas valores literais são permitidos:

```
let a: Resposta = "Sim".toLowerCase(); Erro: não é
possível atribuir string ao Answerlet d: Dígito = 3+4;//
Erro: não é possível atribuir
```

número para dígito

Quando o Flow verifica seus tipos, ele não faz os cálculos: ele apenas verifica os tipos dos cálculos. O Flow sabe que o `toLowerCase()` retorna uma string e que o operador `+` em números retorna um número.

Embora saibamos que ambos os cálculos retornam valores que estão dentro do tipo, o Flow não pode saber disso e sinaliza erros em ambas as linhas.

Um tipo de união de tipos literais como `Answer` e `Digit` é um exemplo de um tipo enumerado ou enumeração. Um caso de uso canônico para tipos de enumeração é representar os naipes de cartas de baralho:

```
tipo Terno = "Paus" | "Diamantes" | "Corações" | "Espadas";
```

Um exemplo mais relevante pode ser os códigos de status HTTP:

```
digite HttpStatus =
| 200// OK| 304// Não
modificado| 403// Não
Proibido| 404// Não
encontrado
```

Um dos conselhos que os novos programadores costumam ouvir é evitar o uso de literais em seu código e, em vez disso, definir constantes simbólicas para representar esses valores. Uma razão prática para isso é evitar o problema de erros de digitação: se você digitar incorretamente um literal de string como "Diamantes", o JavaScript pode nunca reclamar, mas seu código pode não funcionar corretamente. Se você digitar incorretamente um identificador, por outro lado, é provável que o JavaScript gere um erro que você notará. Com o Flow, esse conselho nem sempre se aplica. Se você anotar uma variável com o tipo `Suit` e, em seguida, tentar atribuir um naipe incorreto a ela, o Flow irá alertá-lo sobre o erro.

Outro uso importante para tipos literais é a criação de discriminatedunions. Quando você trabalha com tipos de união (compostos de tipos realmente diferentes, não de literais), você normalmente precisa escrever código para discriminar entre os tipos possíveis. Na seção anterior, escrevemos uma função que poderia receber um array ou um Set ou um Map como seu argumento e teve que escrever código para discriminar a entrada do array da entrada Set orMap. Se você quiser criar uma união de tipos de objeto, poderá tornar esses tipos fáceis de discriminar usando um tipo literal dentro de cada um dos tipos de objeto individuais.

Um exemplo deixará isso claro. Suponha que você esteja usando um thread de trabalho no nó (§16.11) e esteja usando postMessage() e "message"events para enviar mensagens baseadas em objeto entre o thread principal e o thread de trabalho. Há vários tipos de mensagens que theworker pode querer enviar para o thread principal, mas gostaríamos de escrever um tipo de união de fluxo que descreva todas as mensagens possíveis. Considere este código:

```
@flow// O trabalhador envia uma mensagem deste tipo quando é feito// reticulando os splines que enviamos.export type ResultMessage = {  
  
  messageType: "result", result: Array<ReticulatedSpline>, //  
  Suponha que este tipo esteja definido em outro lugar.};  
  
O worker envia uma mensagem desse tipo se seu código falhar com uma exceção. tipo de exportação ErrorMessage = {  
  
  messageType:  
  "erro", erro: Erro,};
```

O worker envia uma mensagem desse tipo para relatar
usagestatistics.export type StatisticsMessage = {

```
messageType:  
"stats",splinesReticulated:  
number,splinesPerSecond:  
number};
```

Quando recebermos uma mensagem do trabalhador, ela será
umWorkerMessage.export type WorkerMessage = ResultMessage
| Mensagem de erro |Mensagem de Estatísticas;

O thread principal terá uma função de manipulador de eventos
que é passada// um WorkerMessage. Mas como definimos
cuidadosamente cada um dos // tipos de mensagem para ter uma
propriedade messageType com um tipo literal,// o manipulador
de eventos pode discriminar facilmente entre as mensagens
possíveis:function handleMessageFromReticulator(message:
WorkerMessage){

```
if (message.messageType === "resultado") {  
    Somente ResultMessage tem uma propriedade messageType com  
Este valor  
    para que o Flow saiba que é seguro usá-lo  
message.result aqui.  
    E o Flow reclamará se você tentar usar qualquer outro  
propriedade.  
    console.log(message.result);} else if  
(message.messageType === "error") {  
    Somente ErrorMessage tem uma propriedade messageType com  
valor "erro"  
    então sabe que é seguro usar message.error  
aqui.  
    lançar mensagem.erro;} else if  
(message.messageType === "stats") {  
    Somente StatisticsMessage tem uma propriedade messageType  
com valor "estatísticas"  
    então sabe que é seguro usar  
message.splinesPerSecond aqui.  
    console.info(message.splinesPerSecond);}
```

}

17.9 Resumo

JavaScript é a linguagem de programação mais usada no mundo hoje. É uma linguagem viva - que continua a evoluir e melhorar - cercada por um ecossistema florescente de bibliotecas, ferramentas e extensões. Este capítulo apresentou algumas dessas ferramentas e extensões, mas há muito mais para aprender. O ecossistema JavaScript floresce porque a comunidade de desenvolvedores JavaScript é ativa e vibrante, cheia de colegas que compartilham seus conhecimentos por meio de postagens em blogs, vídeos e apresentações em conferências. Ao fechar este livro e se juntar a esta comunidade, você não encontrará escassez de fontes de informação para mantê-lo envolvido e aprendendo sobre JavaScript.

Muitas felicidades, David Flanagan, março de 2020

1 Se você programou com Java, pode ter experimentado algo assim na primeira vez que escreveu uma API genérica que usava um parâmetro de tipo. Eu encontrei o aprendizado processo para que o Flow fosse notavelmente semelhante ao que passei em 2004, quando os genéricos foram adicionados ao Java.

Índice

SÍMBOLOS

! (Operador booleano NOT), Lógico NÃO

(!)!= (operador de desigualdade não estrita)

expressões relacionais, conversões do tipo Operadores de

Igualdade e Desigualdade, Conversões de operadores de
casos especiais!== (operador de desigualdade)

valores booleanos, Valores booleanosvisão geral de, Operadores de
igualdade e desigualdadecomparação de strings, Trabalhando com
strings" (aspas duplas), Literais de strings\$ (cifrão), Identificadores e
palavras reservadas% (operador de módulo), Aritmética em JavaScript,
Expressões aritméticas& (operador AND bit a bit), Operadores bit a
bit&& (operador booleano AND), Valores booleanos, Logical AND
&&' (aspas simples), Literais de string* (operador de multiplicação),
Aritmética em JavaScript, Expressões eOperadores, Aritmética
Expressões** (operador de exponenciação), Aritmética em JavaScript,
ArithmetricExpressions

+ (sinal de mais)

operador de adição e atribuição ($+=$), Atribuição com operador Operationaddition, Aritmética em JavaScript, A concatenação + Operatorstring, Literais de cadeia de caracteres, Trabalhando com cadeias de caracteres, As conversões +Operatortype, Conversão de operador de caso especialOperador aritmético sunário, Operadores aritméticos unários++ (operador de incremento), Operadores aritméticos unários, (operador de vírgula), O operador de vírgula (,)

- (sinal de menos)operador de subtração, aritmética em JavaScript, aritméticaExpressãooperador aritmético sunário, operadores aritméticos unários

-- (operador de decremento), Operadores Aritméticos Unários. (operador de ponto), Um tour pelo JavaScript, Consultando e definindo propriedades/ (operador de divisão), Aritmética em JavaScript, Expressões aritméticas/* */ caracteres, Comentários// (barra dupla), Um tour pelo JavaScript, Um tour pelo JavaScript, ComentáriosGráficos 3D, Gráficos em um <canvas>; (ponto e vírgula), ponto e vírgula opcional-ponto e vírgula opcional< (menor que o operador)

visão geral de, Operadores de comparação e comparação de strings, Trabalhando com conversões do tipo Strings, Conversões de operador de caso especial << (operador shift left), Operadores bit a bit <= (menor ou igual a operador)

visão geral de, Operadores de comparação e comparação de strings, Trabalhando com conversões do tipo Strings, Conversões de operador de caso especial = (operador de atribuição), Um tour por JavaScript, Operadores de igualdade e desigualdade, Expressões de atribuição == (operador de igualdade)

visão geral de, Conversões de tipo de operadores de igualdade e desigualdade, Visão geral e definições, Conversões e igualdade, Conversões de operadores de casos especiais === (operador de igualdade estrita)

valores booleanos, Valores booleanos visão geral de, Operadores de igualdade e desigualdade comparação de strings, Trabalhando com conversões do tipo Strings, Visão geral e definições, Conversões e Igualdade => (setas), Um tour pelo JavaScript, Ponto-e-vírgula opcional, Arrow Functions > (operador maior que)

visão geral de, Comparação de strings de operadores de comparação, Trabalhando com conversões de tipo Strings, Conversões de operador de caso especial>= (maior ou igual a operador)

visão geral de, Operadores de comparação de strings, Trabalhando com conversões do tipo Strings, Conversões de operador de maiúsculas e minúsculas>> (deslocar para a direita com operador de sinal), Operadores bit a bit>>> (deslocar para a direita com operador de preenchimento zero), Operadores bit a bit?. (operador de acesso condicional), Um tour pelo JavaScript, FunctionInvocation?: (operador condicional), O operador condicional (?:)??: (operador definido pela primeira vez), Definido pela primeira vez (??)[] (colchetes), Um tour pelo JavaScript, Trabalhando com strings, inicializadores de objetos e matrizes, Consultando e definindo propriedades, Lendo e gravando elementos de matriz, Strings como matrizes\ (barra invertida), Literais de string-Sequências de escape em literais de string\n (nova linha), Literais de string, Sequências de escape em literais de string\u (escape de caracteres Unicode), Sequências de escape Unicode, EscapeSequences em literais de string\xA9 (símbolo de direitos autorais), Sequências de escape em literais de string\' (acento grave ou apóstrofo) escape, Sequências de escape em literais de cadeia de caracteres^ (operador XOR bit a bit), operadores bit a bit

_ (sublinhado), Identificadores e Words_ reservados (sublinhados, como separadores numéricos), Literais de ponto flutuante (acento grave), Literais de cadeia de caracteres, Literais de modelo{} (chaves), Um tour por JavaScript, Inicializadores de objetos e matrizes|| (operador booleano OR), valores booleanos, OR lógico (||)~ (operador NOT bit a bit), Operadores bit a bitl (operador OR bit a bit), Operadores bit a bit... (operador de propagação), Operador de propagação, O operador de propagação, Operador de propagação para chamadas de função, Iteradores e geradores

Um

classes abstratas, hierarquias de classe e classes abstratas- resumo acelerômetros, APIs de dispositivos móveis propriedades do acessador, método Property Getters e Setters addEventListener(), operador de adição addEventListener(), aritmética em JavaScript, o operador + recursos avançados

extensibilidade de objeto, Extensibilidade de objetovisão geral de, Metaprogramação atributos de propriedade, Atributos de propriedade-Atributos de propriedade atributo de protótipo, O atributo protótipo Objetos proxy, Objetos proxy-Invariantes de proxy API de reflexão, A API de reflexão-A API de reflexão

tags de modelo, tags de modelo-tags de
modelosímbolos bem conhecidos

símbolos de correspondência de padrões, Símbolos de
correspondência de padrõesSymbol.asyncIterator, Symbol.iterator
andSymbol.asyncIteratorSymbol.hasInstance,
Symbol.hasInstanceSymbol.isConcatSpreadable,
Symbol.isConcatSpreadableSymbol.iterator, Well-Known
SymbolsSymbol.species, Symbol.species-
Symbol.speciesSymbol.toPrimitive,
Symbol.toPrimitiveSymbol.toStringTag,
Symbol.toStringTagSymbol.unscopables,
Symbol.unscopablesalphabetization, Comparando stringscaracteres
âncora, Especificando correspondência positionapostrophes, String
Literalsapply() método, Invocação indireta, O call() e apply()
Métodosarc() método, CurvesarcTo() método, Curvesarguments

tipos de argumento, Tipos de
ArgumentoObjeto de argumentos, O objeto
de argumentosdefinição do termo, Funções

desestruturação de argumentos de função em parâmetros,

Desestruturação Argumentos de função em parâmetros

Desestruturação de argumentos de função em parâmetros listas de argumentos de comprimento variável, parâmetros REST e listas de argumentos de comprimento variável operadores aritméticos, um tour pelo JavaScript, aritmética em JavaScript-aritmética em JavaScript, expressões aritméticas-operadores bit a bit, leitura e gravação de elementos de matriz métodos de iterador de array

every() e some(), every() e some() filter(), filter() find() e findIndex(), find() e findIndex() forEach(), forEach() map(), map() visão geral de, Array Iterator Methods reduce() e reduceRight(), reduce() e reduceRight() literais de array,

Inicializadores de Objeto e Matriz, Construtor Array

LiteralsArray(), Função Array() ConstructorArray.from(),

Array.from(), Função Static Array

FunctionsArray.isArray(), Função Static Array

FunctionsArray.of(), Array.of(), Static Array

FunctionsArray.prototype, Arrays, método

ArrayArray.sort() semelhante a array, funções como valores

método `arrayBuffer()`, analisando corpos de
resposta arrays

adicionando e excluindo, adicionando e excluindo
elementos de matrizcomprimento da matriz, métodos de
matriz de comprimento da matriz

adicionando arrays, Adicionando arrays com `concat()`array para
conversões de strings, Array para String Conversionsflattening
arrays, Achate arrays com `flat()` e `flatMap()`aplicação genérica de,
Arraysiterator, Array Iterator Methods-reduce()
andreduceRight()visão geral de, Métodos de arraypesquisando e
classificando, Métodos de pesquisa e classificação de arrays-
`reverse()`stacks e filas, Pilhas e filas com `push()`, `pop()`,`shift()` e
`unshift()`funções de array estático, Funções de matriz
estáticassubmatrizes, Submatrizes com `slice()`, `splice()`, `fill()` e
`copyWithin()`objetos semelhantes a matrizes, objetos semelhantes
a matrizes-objetos semelhantes a matrizesmatrizes associativas,
introdução a objetos, objetos como matrizes associativascriando,
criando matrizes-matriz.`from()`definição de termo, visão geral e
definições

expressões do inicializador, Um tour pelo JavaScript, Objeto e matrizInicializadorsitematizes de classificação, Matrizes de iteraçãoomatrizes multidimensionais, Matrizes multidimensionaisaninhadas, Inicializadores de objetos e matrizesvisão geral de, Um tour pelo JavaScript, Matrizesprocessamento com funções, Processamento de matrizes com funçõesleitura e gravação de elementos de matriz, Leitura e gravação de matrizElementosmatrizes esparsas, Matrizes esparsascadeias de caracteres como matrizes, Cadeias de caracteres como matrizesmatrizes digitadas

criando, Criando Arrays Tipados DataView e endianness, DataView e Endiannessmétodos e propriedades, Métodos e Propriedades de Arrays Tipadosvisão geral de, Arrays Tipados e Dados Bináriostipos de arrays tipados, Tipos de Arrays Tipadosusando, Usando Arrays Tipadosfunções de seta, Um Tour de JavaScript, Definindo Funções, ArrowFunctionssetas (=>), Um Tour de JavaScript, Ponto e Vírgula Opcional, ArrowFunctionsCaracteres de controle ASCII, O Texto de um Programa JavaScript

asserções, Um tour pelo JavaScriptoperador de atribuição (=), Um tour pelo JavaScript, Igualdade e desigualdadeOperadores, Expressões de atribuiçãomatrizes associativas, Introdução a objetos, Objetos como associativosArraysassociatividade, Associativityasync do operador palavra-chave, async e await-Detalhes da implementaçãoprogramação assíncrona (consulte também Nó)

palavras-chave assíncronas e await, async e await-
ImplementationDetailsiteração assíncrona

geradores assíncronos, Geradores assíncronositeradores
assíncronos, Iteradores assíncronospara/await, Iteração
assíncrona com for/await,Iteração
assíncronaimplementação, Implementando Iteradores
Assíncronos-Implementando Iteradores Assíncronosretornos
de chamada

retornos de chamada e eventos no Nó, Retornos de chamada
e eventos no Nódefinição de termo, Programação
assíncrona com retornoseventos, Eventoseventos de rede,
Eventos de redetemporizadores, Temporizadores

definição de termo, JavaScript

assíncronoSuporte a JavaScript para, JavaScript

assíncronoPromessas

encadeamento de promessas, encadeamento de promessas -

encadeamento de promessas tratamento de erros com, tratamento de

erros com promessas, mais emPromessas e erros - os métodos catch e

finally fazendo promessas, fazendo promessas - promessas em

sequênciavisão geral de, promessasoperações paralelas, promessas em

paralelo promessas em sequência, promessas em sequência -

promessas em sequênciaresolvendo promessas, resolvendo promessas

- mais sobre promessas e erros retornando de retornos de chamada de

promessa, A captura e finalmentemétodoterminologia, Manipulando

erros com Promisesusing, usando promessas - Manipulando erros com

APIs Promisesaudio

Construtor Audio(), O Construtor Audio() visão geral

de, APIs de áudioAPI WebAudio, A API

WebAudioPalavra-chave await, async e await-Detalhes

da implementaçãoOperador await, O operador await

B

Babel, Transpilação com Babelbackend JavaScript, JavaScript em navegadores da Webbackpressure, Gravando em fluxos e manipulando Backpressure-Writingto Streams e Manipulando Backpressurebackslash (\), String, Literals-Escape Sequences em String Literalsbacktick (`), String Literals, Template Literalsbare catch, método try/catch/finallybezierCurveTo(), ordenação de bytes Curvesbig-endian, DataView e EndiannessBigInt tipo, Inteiros de precisão arbitrária com dados binários BigInt, processamento, Matrizes tipadas e Binary DataView andEndianness, APIs binárias (veja também matrizes tipadas)literais inteiros binários, Literais inteirosoperadores binários, Número de operandosbind() método, O método bind(), Aplicação parcial de funçõesoperadores bit a bit, Operadores bit a bit(), Análise de corpos de respostaescopo de bloco, Escopo de bloqueio de escopo variável e constante, Quando os scripts são executados: operador AND booleano assíncrono e diferido (&&), Valores booleanos, Operador AND lógico (&&)Operador NOT booleano (!), Lógico NÃO (!)

Operador booleano OR (||), Valores booleanos, OR lógico
(||)valores booleanos, valores booleanos-valores
booleanosfunção Boolean(), Conversões
explícitasinstruções break, ferramentas de
desenvolvimento breakbrowser, explorando
JavaScripthistórico de navegação
gerenciando com eventos de hashchange, Gerenciamento de histórico
com eventos de hashchange gerenciando com pushState(),
Gerenciamento de histórico com pushState()visão geral de, Histórico
de navegaçãoalgoritmo de clone estruturado, Gerenciamento de
histórico com pushState()Classe de buffer (Nó), Buffers

C

API de cache, Progressive Web Apps e Service
Workerscalendários, formatação de datas e horáriosmétodo call(),
invocação indireta, métodos call() e apply() retornos de chamada

retornos de chamada e eventos no Nó, Retornos de chamada e
eventos no Nóddefinição de termo, Programação assíncrona
com retornos de chamadaeeventos, Eventoseventos de rede,
Eventos de rede temporizadores, Temporizadores

API de tela, gráficos em uma <canvas>manipulação de -pixel dimensões e coordenadas da tela, Dimensões da tela e Coordenada transformações do sistema de coordenadas, Transformação do sistema de coordenadas - Função example drawImage(), APIs de mídia operações de desenho

curvas, Curvas imagens,
Imagens retângulos,
Retângulo texto,
Texto Atributos gráficos

cores, padrões e gradientes, Cores, padrões e gradientes estilos de linha, Estilos de linha visão geral de, Atributos gráficos salvando e restaurando o estado gráfico, Salvando e restaurando estado gráficos sombras, Sombras estilos de texto, Estilos de texto Translucidez e composição, Translucidez e composição visão geral de, Gráficos em <canvas>caminhos e polígonos, Caminhos e polígonos

manipulação de pixels, Manipulação de pixelsdiferenciação de casos, O texto de um programa JavaScriptcláusulas catch, try/catch/finally-Instruções diversasinstruções catch, Classes de erro.método catch(), Os métodos catch e finally-As classes catch e finallymethods (expressões regulares), Classes de caractereshistogramas de frequênciade caracteres, Exemplo: Método Character FrequencyHistograms-SummarycharAt(), Strings como função Arrayscheckscope(), Processos Closureschild (Node), Trabalhando com Child Processos-fork()

benefícios de, Trabalhando com Processos Filhosexec() e execFile(), exec() e execFile()execSync() e execFileSync(), execSync() e execFileSync()fork(), fork()options, execSync() e execFileSync()spawn(), spawn()declaração de classe, palavra-chave classclass, Classes com a classe Palavra-chave-Exemplo: Um ComplexNumber Classclass métodos, Static Methodsclasses

adicionando métodos a classes existentes, Adicionando métodos a classes e construtores ExistingClasses, Classes e construtores-Propriedade do construtor

constructor property, O construtor Propertyconstructors, class identity e instanceof, Constructors, Class Identity e instanceofnew.target expression, Classes e Constructorsclasses e protótipos, Classes e Prototypesclasses com palavra-chave de classe, Classes com a classe Palavra-chave Exemplo: Uma classe de número complexo

exemplo de classe de número complexo, Exemplo: Uma Classe de Número Complexo-Exemplo: Uma Classe de Número Complexogetters, setters e outras formas de método, Getters, Setters e outras formas de método campos públicos privados e estáticos, Público, Privado e Estático Campos-Campos Públicos, Privados e Estáticosmétodos estáticos, Métodos estáticosprogramação modular com, Módulos com Classes, Objetos e Encerramentosnomenclatura, Palavras Reservadasvisão geral de, Visão Geral e Definições, Classes subclasses

hierarquias de classe e classes abstratas, hierarquias de classe e classes abstratas-resumo

delegação versus herança, Delegação em vez de Herançavisação geral de Subclasses subclasses e protótipos, Subclasses e Protótipos com cláusula extenss, Subclasses com extensões e super-Subclasses com extensões e JavaScript do lado do supercliente, JavaScript em navegadores da Webarmazenamento do lado do cliente, Armazenamento recorte, Método Clippingclosest(), Selecionando elementos com seletores

CSSCLOSURES

combinando com getters e setters de propriedade, Closures erros comuns, Closures definição de termo, Closures regras de escopo lexical e, Closures programação modular com, Automatizando Closure- Based Modularity encerramentos de função aninhada, Closures estado privado compartilhado, Closures agrupamento de código, Agrupamento de código exemplos de código

sintaxe de comentário em, Um tour pelo JavaScript obtendo e usando, Código de exemplo

experimentando o código JavaScript, Explorando
JavaScriptordem de agrupamento, Comparando
Stringscores, Cores, padrões e gradientesoperador de
vírgula (,), O operador de vírgula (,)comentários

sintaxe para, Um tour pelo JavaScript, Comentáriossyntax em
exemplos de código, Um tour pelo método JavaScriptcompare(),
Comparando stringsoperadores de comparação, Operadores de
comparaçãoocompositando, Translucidez e compositinginstruções
compostas, Instruções compostas e vaziaspropriedades computadas,
Nomes de propriedades computadasmétodo concat(), Adicionando
matrizes com operador de acesso condicional concat()(?), Um tour
pelo JavaScript, FunctionInvocationinvocação condicional,
Invocação condicional, Função Invocationcondicional operator (?),
O Operador Condicional (?:)instruções condicionais, Instruções,
Atributo configurável de comutação condicional, Introdução a
objetos, API de console de atributos de propriedade

console.log(), A API do ConsoleSaída formatada com,
Saída formatada com Console

funções definidas por, A API do Console-A API do Consolesuporte para, A função APIconsole.log() do Console, Hello World, Palavra-chave Output const, Declarações com let e constconstants

declaring, Visão geral e definições, Declarações com let andconst, const, let e vardefinição de termo, Declaração de variável e atribuição de nomenclatura, Palavras reservadasconstrutores

Construtor Array(), O construtor Array() ConstrutorAudio(), As classes do construtor Audio() e, Classes e construtores - A propriedade do construtor

constructor property, A propriedade do construtor-A propriedade do construtorconstrutores, identidade de classe e instanceof, Constructors,Class Identity e instanceofnew.target expression, Classes e Construtoresinvocação do construtor, Invocação do construtordefinição de termo, Funçõesexemplos de, Criando objetos com o construtor newFunction(), O construtor Function() ConstructorSet(), A classe Set

Cabeçalho HTTP Content-Security-Policy, instruções Securitycontinue, estruturas de controle, um tour pelo JavaScript-Um tour pelo JavaScript, Statementscookies

API para manipulação, Cookiesdefinição de termo, Atributos de tempo de vida e escopo de cookies, Atributos de cookie: tempo de vida e escopolimitações de, Atributos de cookie: tempo de vida e escopoorigem do nome, Cookiesleitura, Armazenamento de cookiesarmazenamento de cookies, Transformações do sistema de coordenadas, Exemplo de transformação do sistema de coordenadasassímbolo de direitos autorais (\xA9), Sequências de escape no método String LiteralscopyWithin(), copyWithin()API de gerenciamento de credenciais, criptografia e APIs relacionadasRecurso de origem cruzada Compartilhamento (CORS), A política de mesma origem,Solicitações de origem cruzadascript entre sites (XSS), Criptografia de script entre sites, Inteiros de precisão arbitrária com BigInt, Criptografia e APIs relacionadasPixels CSS, Coordenadas do documento e Coordenadas da janela de visualizaçãohorizonte

Folhas de estilo CSS

estilos CSS comuns, Scripts de CSSestilos computados, Estilos computadosAnimações e eventos CSS, Animações e eventos CSSConformações CSS, Classes CSSsyntax do seletor CSS, Selecionando elementos com seletores CSSESTILOS embutidos, Estilos embutidosconvenções de nomenclatura, Estilos embutidosFolhas de estilo de script, Folhas de estilo de scriptObjetos CSSStyleDeclaration, Estilos embutidochaves ({}), Um tour por JavaScript, Inicializadores de objetos e matrizesmoeda, Formatando númeroscurvas, Curvas

D

propriedades de dados, classe Property Getters e SettersDataView, tipo DataView e EndianessDate, Visão geral e definições, Datas e horasdatas e horas

aritmética de data, aritmética de dataformatando e analisando strings de data, formatando e analisando dateStringsformatação para internacionalização, formatando datas e horas-

Formatação de datas e horas
carimbos de data/hora
de alta resolução, carimbos de data/hora
visão geral
de, datas e horas, datas e horas
carimbos de data/hora
instruções do
depurador, declarações do depurador

class, classconst, let e var, const, let e var
function, functionimport
and export, import and export
overview de,
DeclarationsdecodeURI() function, Legacy URL
FunctionsdecodeURIComponent() function, Legacy URL
Functionsdecrement operator (--), Unary Arithmetic
Operatorsdelegation, Delegation Instead of Inheritance
delete operator, The delete Operator, Deleting Properties
ataques de negação de serviço, Gravação em fluxos e
manipulação
Backpressure
desestruturação de atribuição,
Desestruturação Atribuição-DesestruturaçãoAtribuição,
Desestruturação de argumentos de função em parâmetros-
Desestruturação de argumentos de função em
parâmetros
ferramentas de desenvolvimento, Explorando
JavaScriptdispositivoevento de movimento, APIs de dispositivo
móvel

evento deviceorientation, APIs de dispositivo móvelpropriedade devicePixelRatio, Coordenadas do documento e ViewportCoordinatesdictionaries, Introdução a objetos, Objetos como matrizes associativasdiretórios (nó), Trabalhando com a função Directoriesdistance(), Declarações de funçãooperador de divisão (/), Aritmética em JavaScript, Expressões aritméticasdo/while loops, do/whiledocument geometry e rolagem, Geometria do documento e tamanho da janela de visualização de rolagem, tamanho do conteúdo e posição de rolagem

Pixels CSS, Coordenadas do documento e coordenadas da janela de visualizaçãodeterminando o elemento em um ponto, Determinando o elemento em um pontoCoordenadas do documento e coordenadas da janela de visualização, DocumentCoordinates e Coordenadas da janela de exibiçãoonstalando a geometria dos elementos, Consultando a geometria de umElementorolagem, RolagemTamanho da janela de exibição, tamanho do conteúdo e posição de rolagem, Tamanho da janela de visualização, Tamanho do conteúdo e posição de rolagemModelo de objeto de documento (DOM), O modelo de objeto de documento - Exemplo: Gerando um sumário

estrutura e passagem do documento, nós Document Structure e TraversalDocumentFragment, usando componentes da Web

gerando índices dinamicamente, Exemplo: Gerando um Índice
elementos de quadros, Coordenadas do documento e Coordenadas da
janela de exibição modificando conteúdo, Conteúdo do elemento como
HTML modificando estrutura, Criando, Inserindo e Excluindo
Nósvisão geral de, Documentos de script consultando e definindo
atributos, Atributos selecionando elementos do documento,
Selecionando elementos do documento DOM de sombra, Shadow
DOM-API do DOM de sombra Documento Fragmento de nós, Usando
componentes da Web documentos, carregando novo, Carregando novo
Documentos cifrão (\$), Identificadores e palavras reservadas Evento
DOMContentLoaded, Execução de programas JavaScript, Operador
de ponto de linha do tempo JavaScript do lado do cliente (.), Um tour
pelo JavaScript, Consulta e configuração de propriedades aspas duplas
("), Literais de cadeia de caracteres barras duplas (//), Um tour pelo
JavaScript, Um tour pelo JavaScript, Função Comments drawImage(),
APIs de mídia operações de desenho

curvas, Curves images,
Imagens

retângulos, Retângulo

Textmatrizes

dinâmicas,
Matrizes

E

ECMA402 padrão, A Internacionalização APIECMA Script (ES),

Introdução ao método JavaScript elementFromPoint(),

Coordenadas do documento e ViewportCoordinates

Elementos de matriz

definição de termo, Matrizes

leitura e escrita, Leitura e

Gravação de Elementos de Matriz

elementos personalizados, Elementos personalizados determinando o

elemento em um ponto, Determinando o elemento em um Pointiframe,

Coordenadas do documento e coordenadas da janela de

visualização consultando a geometria dos elementos, Consultando a

geometria de um elemento selecionando, Selezionando elementos do

documento método ellipse(), Curvas else if instruções, else if emojis,

Unicode, Sequências de escape em literais de string

instruções vazias, Instruções compostas e vazias cadeias de caracteres vazias, TextencodeFunção URI(), Funções de URL legadas encodeURIComponent(), Funções de URL herdadas Contrações em inglês, Literais de cadeia de caracteres atributo enumerável, Introdução a objetos, Atributos de propriedade operador de igualdade (==)

visão geral de, Operadores de igualdade e desigualdade conversões de tipo, Visão geral e definições, Conversões e igualdade, Conversões de operadores de casos especiais operadores de igualdade, Um tour por JavaScript Classes de erro, Classes de erro tratamento de erros

usando Promessas, Lidando com erros com promessas, Mais sobre promessas e Erros ambiente de host do navegador da web, Erros de programa ES2016

operador de exponenciação (**), Aritmética em JavaScript, Método ArithmeticExpressionsincludes(), includes() ES2017, palavras-chave assíncronas e await, O operador await, JavaScript assíncrono, async e await-Implementation Details ES2018

iterador assíncrono, iteração assíncrona com
for/await, Iteração assíncrona de estruturação com parâmetros
rest, desestruturação de FunctionArguments em
Parameters.finally() método, Os métodos catch e
finally expressões regulares

asserções lookbehind, Especificando a posição de
correspondênci agrupos de captura nomeados, Alternância,
agrupamento e referênci ass sinalizador, Sinalizadores Classes de
caracteres Unicode, Classes de caracteres operador spread (...),
Operador de propagação, Desestruturação de função Argumentos em
parâmetros, iteradores e geradores ES2019

cláusulas catch simples, try/catch/finally flattening arrays,
Flattening arrays com flat() e flatMap() ES2020

?? operador, Primeiro Definido (??) Tipo BigInt, Inteiros de
precisão arbitrária com BigIntBigInt64Array(), Tipos de matriz
tipadaBigUint64Array(), Tipos de matriz tipada operador de
acesso condicional (?.), Um tour pelo JavaScript, Erros de
PropertyAccess, Invocação condicional de função, Invocação
condicional

globalThis, A função Global Objectimport(), Importações dinâmicas com import()lastIndex e API RegExp, método exec()matchAll(), matchAll(), exec(), Implementando precedência de operador IterableObjects, Operador PrecedencePromise.allSettled(), Promessas em expressões de acesso Parallelproperty, Propriedade condicional AccessES5

apply(), Os métodos call() e apply() quebrando cadeias de caracteres em várias linhas, Literais de String, EscapeSequences em Literais de Stringbugs endereçados por variáveis de escopo de bloco, Linha de base de Closurescompatibility, Introdução ao método JavaScriptFunction.bind(), O construtor Propertygetters e setters, Solução alternativa de Property Getters e SettersIE11, Módulos JavaScript na Webtranspilação com Babel, Transpilação com BabelES6

Função Array.of(), Funções de seta Array.of(), Funções de definição, Funções de setaInteiros binários e octais, Literais de inteiros embutidosFunção de tag integrada, Literais de modelo marcados

declaração de classe, palavra-chave classclass, Classes com
a classe Palavra-chave-Exemplo: AComplex Número
Classpropriedades computadas, Nomes de propriedades
computadassintaxe literal de objeto estendido, Propriedades
abreviadassintaxe para/de/de-loops, para/de-parágrafo emSolução
alternativa do IE11, Módulos JavaScript nas cadeias de
caracteres Webiteráveis em, Textiterando matrizes, Iterando
matrizesObjeto matemático, Aritmética em
JavaScriptmódulos em

importações dinâmicas com import(), Importações dinâmicas
comimport(exports, ES6 Exportsimport.meta.url,
import.meta.urlimports, ES6 Imports-ES6 Importsimports e
exportações com renomeação, Importações e exportações
comRenomeaçãoMódulos JavaScript na web, Módulos JavaScript
na Web-Módulos JavaScript na Webvisão geral de, Módulos no
ES6re-exports, ReexportaçõesPromessas

encadeamento de promessas, encadeamento de promessas—
encadeamento de promessastratamento de erros com, mais sobre
promessas e erros-A capturae finalmente métodosfazendo
promessas, fazendo promessas-promessas em sequênciavisão—
geral de, promessasoperações paralelas, promessas em
paraleloPromessas em sequência, promessas em sequência-
promessas em Sequênciaresolvendo promessas, resolvendo
promessas-mais sobre promessas e errosretornando de retornos de
chamada de promessas, A captura e, finalmentemétodosusando,
Usando promessas-Manipulando erros com a propriedade
Promises ordem de enumeração, Ordem de enumeração de—
propriedadeslançamento de, Introdução ao JavaScriptClasses Set
e Map, for/of com métodos Set e Mapshorthand, Shorthand
Methodsoperador de propagação (...), O operador de
propagaçãostings delimitadas com acentos graves, Literais de
string, Literais de modelossubclasses com cláusula extends,
Subclasses com extends e super-Subclasses com extends e tipo
superSymbol, Visão geral e Definiçõessímbolos como nomes de
propriedade, Símbolos como nomes de propriedadesmatrizes
tipadas, Matrizes

declaração de variável em, declaração e atribuição de variávelpalavra-chave `yield*`, `yield*` e geradores recursivossequências de escape

apóstrofos, Literais de cadeia de caracteresem literais de cadeia de caracteres, Sequências de escape em literais de cadeia de caracteresUnicode, Função de fuga Unicode `escape()`, Funções de URL legadasESLint, Linting com função ESLint `eval()`, Expressões de avaliação-Strict `eval()`

global `eval()`, Global `eval()`strict `eval()`, Strict `eval()`evaluation expressions, Evaluation Expressions-Strict `eval()`event listeners, Events, Eventsevent-driven programming model, Asynchronous JavaScript, Events-Dispatching Custom Events, Server-Sent Events

definição de termo, JavaScript assíncronodespachando eventos personalizados, despachando eventos personalizadoscancelamento de eventos, Cancelamento de eventoscategorias de eventos, Categorias de eventosinvocação do manipulador de eventos, Invocação do manipulador de eventospropagação de eventos, Propagação de eventosvisão geral de, Eventos

registrando manipuladores de eventos, Registrando manipuladores de eventos enviados pelo servidor, Eventos enviados pelo servidor recursos da plataforma web para investigar, classe EventsEventEmitter, método Events e EventEmitter every(), cada() e algumas() exceções, lançando e capturando, método throwexec(), notação exponencial exec(), operador Floating-Point Literalsexponentiation (**), Aritmética em JavaScript, declaração ArithmeticExpressionsexport, palavra-chave import and exportexport, módulos em instruções ES6expression, expressão Instruçõesexpressões

expressões aritméticas, expressões aritméticas-operadores bit a bitexpressões de atribuição, expressões de atribuição-atribuiçãocom operação definição de termo, expressões e operadoresincorporando em literais de cadeia de caracteres, literais de cadeia de caracteresexpressões de avaliação, expressões de avaliação-avaliação estrita eval()formando com operadores, um tour pelo JavaScript, expressões e operadoresexpressões de definição de função, expressões de definição de função

expressões de função, Expressões de função, Funções comoNamespaces expressão inicializador, Um tour pelo JavaScript, Expressões de invocação de objeto e matrizInicializadores, Expressões de invocação-Invocação condicional, Invocação de função-Invocação do construtor expressões lógicas, Expressões lógicas-Lógicas NÃO (!)new.target expression, Classes e Construtoresinicializadores de objetos e matrizes, Inicializadores de objetos e matrizesexpressões de criação de objetos, Expressões de criação de objetosexpressões primárias, Expressões primáriasexpressões de acesso a propriedades, Expressões de acesso a propriedades-Acesso a propriedades condicionaisexpressões relacionais, Expressões relacionais-A instânciadeinstruções Operatorversus, Um tour por JavaScript, Statementsextensibilidade, Extensibilidade de objetos

F

função factorial(), Declarações de função, Função Invocationfunções de fábrica, Classes e Protótiposvalores falsos, Função booleana Valuesfetch(), Método Network Eventsfetch()

anulando solicitações, Abortando uma solicitação
solicitações de origem cruzada, Solicitações de origem cruzada
exemplos de, fetch()upload de arquivo, Upload de arquivo com fetch()
Códigos de status HTTP, cabeçalhos de resposta e erros de rede, HTTP
códigos de status, cabeçalhos de resposta e erros de rede
opções de solicitação diversas, Opções de solicitação diversas
analizando corpos de resposta, Analisando corpos de resposta
configurando cabeçalhos de solicitação, Configurando cabeçalhos de
solicitação, Configurando cabeçalhos de solicitação
configurando parâmetros de solicitação, Configurando parâmetros de
solicitação especificando método de solicitação e solicitação body,
Especificando o método de solicitação e o corpo da solicitação de,
fetch()corpos de resposta de streaming, corpos de resposta de
streamingcampos, público, privado e estático, Campos públicos,
privados e estáticosmanipulação de arquivos (nó), Trabalhando com
arquivos-Trabalhando com diretórios

diretórios, Trabalhando com diretórios
metadados de arquivo, Metadados de arquivo
cadeias de caracteres de modo de arquivo, Gravando arquivos
operações de arquivo, Operações de arquivo
visão geral de, Trabalhando com arquivos
caminhos, descritores de arquivo e FileHandles, Caminhos, Descritores de
arquivo e FileHandles

lendo arquivos, Lendo arquivosgravando arquivos, Escrevendo o método Filefill(), método fill(filter()), método filter().finally(), Os métodos catch e finally-Os métodos catch e finally, números de contas financeiras, método Storagefind(), find() e findIndex()método findIndex(), find() e findIndex()Firefox Developer Tools, Explorando JavaScriptoperador primeiro definido (??), primeiro definido (??)método flat(), Nivelamento de matrizes com flat() e flatMap()flatMap() método, Nivelamento de matrizes com flat() e flatMap()literais de ponto flutuante, Literais de ponto flutuante, Ponto flutuante binário e Erros de arredondamentoExtensão de linguagem Flow, Verificação de tipo com Flow-EnumeratedTypes e Uniões discriminadas

tipos de matriz, tipos de matriztipos de classe, tipos de classetipos enumerados e uniões discriminadas, tipos enumerados e uniões discriminadastipos de função, tipos de funçãoinstalando e executando, instalando e executando fluxo

tipos de objeto, tipos de objeto outros tipos parametrizados, outros tipos parametrizadosvisão geral de, Verificação de tipo com fluxotipos somente leitura, tipos somente leitura aliases de tipo, aliases de tipoScript versus fluxo, Verificação de tipo com tipos de união de fluxo, tipos de união usando anotações de tipo, usando anotações de tipo para loops, para, Iterando matrizespara/aguardar loops, Iteração assíncrona com para/aguardar, Iteração assíncronapara/em loops, para/em, Enumerando propriedadespara/de loops, Texto, para/de-para/em, Iterando matrizes, Iteradores eGeneratorsforEach() método, Iterando matrizes, forEach()format() método, Formatando númerosfrações, Formatando números do método Data(), Analisando corpos de respostaJavaScript front-end, JavaScript em navegadores da Webmódulo fs (Nó), Trabalhando com arquivos - Trabalhando com diretóriosdeclaração de função, funçãoexpressões de função, expressões de função, funções como namespacespalavra-chave de função, definindo funções

Construtor Function(), A palavra-chave Function()

Constructorfunction*, Generatorsfunctions

funções de seta, Um tour pelo JavaScript, Definindo funções, ArrowFunctionsdiferenciação de maiúsculas e minúsculas, O texto de um programa JavaScriptencerramentos, Encerramentos-Encerramentosdefinindo, Definindo funções-Funções aninhadasdefinindo suas próprias propriedades de função, Definindo suas própriasPropriedades de funçãofunções de fábrica, Classes e Protótiposargumentos e parâmetros de função

tipos de argumento, Tipos de argumentosobjeto de argumentos, O Objeto de Argumentosdesestruturação de argumentos de função em parâmetros,Desestruturação de argumentos de função em parâmetros-Desestruturação de argumentos de função em parâmetrosparâmetros opcionais e padrões, Parâmetros opcionais ePadrõesvisão geral de, Argumentos de função e parâmetrosparâmetros de descanso, Parâmetros de descanso e comprimento variávelListas de argumentosoperador de propagação para chamadas de função, O operador de propagação paraChamadas de função

listas de argumentos de comprimento variável, Parâmetros Rest e Listas de Argumentos de Comprimento Variávelexpressões de definição de função, Expressões de Definição de Funçãoinvocação de função, Criando Objetos com novapropriedades, métodos e construtor de função, Propriedades de Função, Métodos e Construtor - O Construtor Function()

bind(), Os métodos bind() Methodcall() e apply(), O construtor MethodsFunction() call() e apply(), A propriedade Constructorlength Function(), A propriedade Propertyname de comprimento, A propriedade Propertyprototype do nome, O método prototype PropertytoString(), O método toString() Programação funcional

explorando, Programação Funcionalfunções de ordem superior, Funções de Ordem Superiormemoização, Memorizaçãooaplicação parcial de funções, Aplicação parcial deFunçõesprocessando matrizes com função, Processando Matrizes comFunçõesfunções como namespaces, Funções como Namespacesfunções como valores, Funções como Valores-Definindo Suas PrópriasPropriedades da Função

Invocar

abordagens para, Invocando funçõesinvocação do construtor, Invocação do construtorexemplos, Um tour pelo JavaScriptinvocação de função implícita, Invocação implícita de funçãoinvocação indireta, Invocação indiretaexpressões de invocação, Invocação de funçãoinvocação de método, Invocação de método-Invocação de métodonomenclatura, Palavras reservadasvisão geral de, Visão geral e definições, Funçõesfunções recursivas, Invocação de funçõesyntaxe abreviada para, Um tour pelo JavaScriptfunções de matriz estática, Matriz estática Funções

G

coleta de lixo, Visão geral e definiçõesfunções geradoras, Geradores, O valor retornado de uma função geradora (consulte também iteradores e geradores)API de geolocalização, Método APIs de dispositivo móvelgetBoundingClientRect(), Coordenadas do documento e método Viewport CoordinatesgetRandomValues(), Criptografia e APIs relacionadasmétodos getter, Getters e Setters de propriedade, Getters, Setters e outros formulários de método

global eval(), Global eval()objeto global, O objeto global, O objeto global em navegadores da Webvariáveis globais, Escopo variável e constantegradientes, Cores, padrões e gradientesgráficos

3D, gráficos em uma <canvas>API de tela, gráficos em uma <canvas>manipulação de pixels

dimensões e coordenadas da tela, Dimensões e coordenadas da telarecorte, Recortetransformações do sistema de coordenadas, Sistema de coordenadasTransformações-Exemplo de transformaçãooperações de desenho, Operações de desenho da telaatributos gráficos, Atributos gráficosvisão geral de, <canvas>Gráficos em caminhos e polígonos, Caminhos e polígonosmanipulação de pixels, Manipulação de pixelssalvando e restaurando o estado gráfico, Salvando e restaurandoestados gráficosgráficos vetoriais escaláveis (SVG), SVG: Vetor escalável Criação de gráficos de imagens SVG com JavaScriptmaior que (>)

visão geral dos Operadores de Comparaçāo

comparação de cadeia de caracteres, Trabalhando com conversões de tipo Strings, Conversões de operador de maiúsculas e minúsculas maiores ou iguais a (\geq)

visão geral de, Operadores de Comparação de strings, Trabalhando com conversões do tipo Strings, Conversões de operadores de maiúsculas e minúsculas especiais

H

eventos de hashchange, Gerenciamento de histórico com eventos de hashchangehashtables, Introdução a objetos, Objetos como matriz associativasOperador OwnProperty, Propriedades de testeOlá, Mundo, Olá, Mundo, Saída do consoleLiterais hexadecimais, Literais inteiros, Sequências de escape em StringLiteralsfunções de ordem superior, Funções de ordem superiorhistogramas, frequência de caracteres, Exemplo: Frequência de caracteresHistogramas-Summaryhistory.pushState() método, Gerenciamento de histórico com pushState()history.replaceState(), Gerenciamento de Histórico com pushState()hoisting, Declarações de Variáveis com tags varHTML <script>, JavaScript em <script> HTML Tags-Loadingscripts sob demanda

diretivas de importação e exportação, móduloscarregando scripts sob demanda, carregando scripts sob demanda

especificando o tipo de script, Especificando o tipo de scriptexecução de script síncrona, Quando os scripts são executados: propriedade async edeferredtext, Conteúdo do elemento como texto <template> simplesTag HTML, Modelos HTMLCodio HTML, aspas simples e duplas em, Literais de stringClientes e servidores HTTP, Clientes e servidores HTTP-Cientes e servidores HTTP-Cientes e servidores HTTP

Eu

Identificadores

diferenciação de maiúsculas e minúsculas, O texto de um programa JavaScriptfinalidade de, Identificadores e palavras reservadas, Declaração de variável e Atribuiçãopalavras reservadas, Identificadores e palavras reservadas, PrimaryExpressionssintaxe para, Identificadores e palavras reservadassideografias, Instruções Unicodeif, instrução if-ifif/else, Valores booleanosimagens

desenho no Canvas, Imagespixel manipulation, Pixel Manipulationexpressão de função imediatamente invocada, Funções como Namespaces

imutabilidade, Trabalhando com Strings, Valores Primitivos
Imutáveis e Referências de Objetos Mutáveis
invocação de função implícita, Invocação de Função Implícita
declaração de importação, palavra-chave import e export
import, Módulos na função ES6
import(), Importações Dinâmicas com import()
import.meta.url, operador import.meta.url
in, O Operador in, Testando o método Properties
includes(), includes() increment operator (++), Operadores Aritméticos
Unários Posição do índice, Matrizes, Leitura e Gravação
de Elementos de Matriz IndexedDB, IndexedDB-Worker Threads e
método Messaging indexOf(), indexOf() e lastIndexOf() invocação
indireta, operador de invocação indireta (!==)

valores booleanos, Valores booleanos
visão geral de, Operadores de igualdade e desigualdade
comparação de strings, Trabalhando com strings
valor infinito, Aritmética em JavaScript
herança, Introdução a objetos, Herança, Delegação em vez de
Herança expressão inicializador, Um tour por JavaScript, Objeto e matriz
Inicializadores

métodos de instância, Static Methods`instanceof` operator, The
`instanceof` Operator, Constructors, ClassIdentity e
instanceofinteger literals, API de internacionalização de Integer
Literals

classes incluídas em, A API de Internacionalização comparando
cadeias de caracteres, Comparando cadeias de caracteres-
Comparando cadeias de caracteresformatando datas e horas,
Formatando datas e horas-Formatando datas e horasformatando
números, Formatando números-Formatando númerossuporte para no
Nº, A API de InternacionalizaçãoTexto traduzido, A API de
InternacionalizaçãoInterpolação, Literais de Cadeia de
CaracteresIntl.DateTimeClasse Format, Formatando Datas e Horas-
FormataçãoDatas e HorasIntl.NumberClasse Format, Formatação
Numbers-Formatação de Númerosexpressões de invocação

invocação condicional, Invocação condicional,
FunctionInvocationinvocação de método, Expressões de invocação,
Invocação de métodovisão geral de, função InvocationisFinite(),
Aritmética em JavaScript`isNaN()` função, aritmética em JavaScript

Iteradores e geradores (consulte também Métodos de iterador de matriz)

Recursos avançados do gerador

valor de retorno de funções geradoras, O valor de retorno de uma função geradora métodos return() e throw(), Os métodos return() e throw() Métodos de um gerador valor de expressões de rendimento, O valor de uma expressão de rendimento assíncrono, Iteradores assíncronos - Implementando Iteradores assíncronos fechando iteradores, "Fechando" um iterador: O método de retorno geradores

benefícios de, Uma Nota Final Sobre Geradores criando, Geradores definição de termo, Geradores exemplos de, Exemplos de Geradores rendimento* e geradores recursivos, rendimento* e Recursivo Gerador mostrar o trabalho dos iteradores, Como Funcionam os Iteradores implementando objetos iteráveis, Implementando Objetos Iteráveis - Implementando Objetos Iteráveisvisão geral de, Iteradores e Geradores

J

JavaScript

benefícios de, Introdução ao JavaScript,
Resumo

visões gerais do capítulo, Um tour pelo JavaScript, Um tour pelo
JavaScript histogramas de frequência de caracteres, Exemplo:
PersonagemHistogramas de frequência-ResumoOlá, Mundo, Olá
mundohistória de, JavaScript em navegadores da
WebInterpretadores de JavaScript, Explorando
JavaScriptestrutura lexical, Estrutura lexical-Resumonomes,
versões e modos, Introdução ao JavaScriptsyntax e recursos, Um
tour pelo JavaScript-Um tour pelo JavaScriptdocumentação de
referência, PrefácioBiblioteca padrão JavaScript

API do console, saída formatada pela API do console com
consoledatas e horas, datas e horas - formatação e análise de
dataStringsclasses de erro, classes de erro-classes de erroAPI de
internacionalização, API de internacionalização da API de
internacionalização, serialização e análise JSON, serialização
JSON e análise-personalizações JSONvisão geral de, Biblioteca
padrão JavaScript

correspondência de padrões, correspondência de padrões com expressões regulares-exec()conjuntos e mapas, conjuntos e mapas- WeakMap e WeakSettimers, temporizadores-temporizadoresmatrizes digitadas e dados binários, matrizes tipadas e APIs binárias DataView e EndiannessURL, APIs de URL-funções de URL legadasJest, teste de unidade com o método Jestjoin(), conversões de matriz para stringSerialização e análise JSON, função JSON e Parsing-JSONCustomizationsJSON.parse(), Serialização de objetos, serialização JSON e ParsingJSON.stringify() função, Serialização de objetos, O método toJSON(), Serialização JSON e AnáliseExtensão de linguagem JSX, JSX: Expressões de marcação em JavaScript- JSX:Expressões de marcação em instruções JavaScriptjump, Instruções, Jumps-try/catch/finally

instruções break, breakcontinue,
continuedefinição de termo,
Instruçõesrotuladas, Instruções
rotuladasvisão geral de,
Jumpsinstruções return, return

K

Keywords

async, async e await-Implementation Detailsawait palavra-chave, async e await-Implementation Detailscase sensitivity, O texto de um programa JavaScriptpalavra-chave de classe, Classes com a palavra-chave class Exemplo: AComplex Number Classconst palavra-chave, Declarações com let e constexport palavra-chave, Módulos na palavra-chave ES6function, Definindo funções de função* palavra-chave, Generatorsimport palavra-chave, Módulos na palavra-chave ES6let, Um tour pelo JavaScript, Declarações com let e const, const, let e varnew palavra-chave, Criando objetos com new, Invocação do construtorpalavras reservadas, Palavras reservadas, Expressões primáriasesta palavra-chave, Um tour pelo JavaScript, Expressões primárias,Invocação de função palavra-chave var, Declarações de variáveis com var, const, let e varyield* palavra-chave, yield* e geradores recursivosFlocos de neve de Koch, exemplo de transformação

L

instruções rotuladas, propriedade Labeled
StatementslastIndex, método exec()lastIndexOf(),
indexOf() e lastIndexOf()menor que o operador
(<)

visão geral de, Operadores de comparaçãocomparação
de strings, Trabalhando com conversões de tipo Strings,
Conversões de operador de caso especialless ou igual a
(<=)

visão geral de, Operadores de comparaçãocomparação de strings,
Trabalhando com conversões de tipo Strings, Conversões de
operador de caso especialpalavra-chave let, Um tour pelo
JavaScript, Declarações com escopo let e const, const, let e
varlexical, Closuresestrutura lexical, Estrutura lexical-Resumo

diferenciação de maiúsculas e minúsculas, O texto de um
programa JavaScriptcomentários,
Comentáriosidentificadores, O texto de um programa
JavaScript - Identificadores e palavras reservadas quebras
de linha, O texto de um programa JavaScript literais,
Literais palavras reservadas, Palavras reservadas,
Expressões primáriasponto e vírgula, Ponto e vírgula
opcionais - Ponto e vírgula opcional

espaços, o texto de um programa

JavaScriptConjunto de caracteres Unicode

sequências de escape, Sequências de escape Unicode
normalização, Normalização Unicodevisão geral de, Unicode
quebras de linha, O texto de um programa JavaScript, Ponto-e-vírgula opcional-Ponto e
vírgula opcional estilos de linha, Estilos de linha terminadores de
linha, O texto de uma programação JavaScriptFerramentas de
linting, Linting com ESLint literais

numérico

literais de ponto flutuante, literais de ponto flutuante, binárioErros
de ponto flutuante e arredondamento literais inteiros, literais
inteiros números negativos, Números separadores em, literais de
ponto flutuante expressões regulares, correspondência de padrões,
correspondência de padrões com Expressões regulares string,
literais de string literais de modelo, literais de modelo, tags de
modelo arquitetura little-endian, evento DataView e
Endianness load, execução de programas JavaScript

propriedade localStorage, propriedade localStorage e...
sessionStorage, Location, Navigation e History
operadores lógicos, Um tour pelo JavaScript, Expressões lógicas-Logical NOT (!) lookbehind, especificando instruções de loop de posição de correspondência

loops do/while, loops do/while for, for, Iterating Arrays for/await loops, Iteração assíncrona com for/await, Loops de iteração assíncrona for/in, for/in, Enumerando propriedades para/de loops, for/of-for/in, Iterating Arrays, Iterators and Generators finalidade de, Statements while loops, while value, Operand Result Type

M

magnetômetros, APIs de dispositivos móveis Conjunto de Mandelbrot, Exemplo: O conjunto de Mandelbrot - Resumo e sugestões para leitura adicional Classe de mapa, para/de com conjunto e mapa, a classe de mapa - a classe de mapa de classe de mapas, visão geral e definições, o método de mapa de classe de mapa(), map()

marshaling, Serialização JSON e método Parsingmatch(), método match()matchAll(), método matchAll()matches(), Seleção de elementos com seletores CSSfunção Math.pow, Expressões aritméticaoperações matemáticas, Aritmética em JavaScript-Aritmética em JavaScriptSite MDN, Prefácio APIs de mídia, APIs de mídia memoização, Memoização gerenciamento de memória, Visão geral e definições eventos de mensagem, Modelo de threading JavaScript do lado do cliente, Eventos, Eventos enviados pelo servidor, Objetos de trabalho - O objeto global em Workers, Worker Execution Model-postMessage(), MessagePorts, andMessageChannels, Mensagens de origem cruzada com postMessage(), fork()-Worker Threads, Canais de comunicação e MessagePorts, Tipos enumerados e uniões discriminadasMessageChannels, postMessage(), MessagePorts e objetos MessageChannelsMessagePort, postMessage(), MessagePorts e MessageChannels, Canais de comunicação e MessagePortsmessaging

WebSocket API

recebimento de mensagens, recebimento de mensagens de um WebSocket envio de mensagens, envio de mensagens por meio de um WebSocket

worker threads and messaging, Worker Threads e Messaging-Cross-Origin Messaging with postMessage()

mensagens de origem cruzada, mensagens de origem cruzada withpostMessage(), mensagens de origem cruzada com postMessage() modelo de execução, modelo de execução de trabalho importando código, importando código para um trabalhador Exemplo de conjunto de Mandelbrot, exemplo: o resumo do conjunto de Mandelbrot e sugestões para leitura adicionalmódulos, importando código para um trabalhadorvisão geral de, Worker Threads e Messaging postMessage(), MessagePorts e MessageChannels, postMessage(), MessagePorts e MessageChannels Objetos de trabalhador, Worker Objeto Objects Worker Global Scope, O Objeto Global em Workers metaprogramming, Metaprogramming methods

adicionando métodos a classes existentes, adicionando métodos a métodos Existing Classes array

aplicação genérica de, Matrizesvisão geral de, Métodos de matriz classe versus métodos de instância, Métodos estáticos criando, Um tour pelo JavaScript definição de termo, Funções, Invocação de método

encadeamento de métodos, Invocação de método invocação de método, Expressões de invocação, Invocação de método-Invocação de método métodos abreviados, Getters, Setters e outras formas de métodos sintaxe abreviada, Métodos abreviados métodos estáticos, Métodos estáticos métodos de matriz tipada, Métodos de matriz tipada e Propriedades essenciais (-)

operador de subtração, Aritmética em JavaScript, Aritmética Expressão Operador aritmético unário, Operadores Aritméticos Unários APIs de dispositivos móveis, APIs de dispositivos móveis módulos

automatizando a modularidade baseada em fechamento, automatizando a modularidade baseada em fechamento no ES6

importações dinâmicas com import(), Importações dinâmicas com import() exports, ES6 Exports import.meta.url, import.meta.url imports, ES6 Imports-ES6 Imports imports e exportações com renomeação, Importações e exportações com Renomeando Módulos JavaScript na web, Módulos JavaScript no

Módulos Web-JavaScript na Webvisão geral de, Módulos em ES6re-exports, Re-Exportsmódulo sfs (Node), Trabalhando com arquivos-Trabalhando com diretóriosdiretivas de importação e exportação, Módulosno Nó, Módulos em Node-Node-Style Modules na Web,Módulos de nó

Exportações de nós, Exportações de nósImportações de nós,
Importações de nósMódulos de estilo de nó na Web, Módulos de estilo de nó na Webvisão geral de, Módulosfinalidade de, Módulosusando em trabalhadores, Importando código para um trabalhadorcom classes, objetos e encerramentos, Módulos com classes, objetos e encerramentos-Automatizando a modularidade baseada em fechamentooperador de módulo (%), Aritmética em JavaScript, Expressões aritméticasoperador de multiplicação (*), Aritmética em JavaScript, Expressões eOperadores, Expressões aritméticasmultithreaded programação, o nó é assíncrono por padrão,Worker threadsmutability, visão geral e definições, introdução aos objetos

N

grupos de captura nomeados, Alternância, agrupamento e referênciasNaN (valor diferente de um número), Aritmética em JavaScriptnavigator.mediaDevices.getUserMedia() função, Método Media APIsnavigator.vibrate(), APIs de dispositivos móveisValor infinito negativo, Aritmética em JavaScriptzero negativo, Aritmética em JavaScriptFunções aninhadas, Funções aninhadaseventos de rede, Eventos de rede, Negociação de protocolo de rede

método fetch()

anulando solicitações, Abortando uma solicitação solicitações de origem cruzada, Solicitações de origem cruzada exemplos de, fetch()upload de arquivo, Upload de arquivo com fetch()Códigos de status HTTP, cabeçalhos de resposta e erros de rede,Códigos de status HTTP, cabeçalhos de resposta e erros de redeOpções de solicitação diversas, Opções de solicitação diversas analisando corpos de resposta, Analisando corpos de resposta configurando cabeçalhos de solicitação, Configurando cabeçalhos de solicitaçãoconfigurando parâmetros de solicitação, Configurando parâmetros de solicitação especificando método de solicitação e solicitação body, Especificando o método request e o corpo da solicitação de, fetch()

corpos de resposta de streaming, Corpos de resposta de streamingvisão geral de, Redeeventos enviados pelo servidor, Eventos enviados pelo servidorWebSocket API, WebSocketsXMLHttp API de solicitação (XHR), fetch()new palavra-chave, Criando objetos com new, Constructor Invocationnew.target expression, Classes e construtoresnewline (\n), Literais de string, Sequências de escape em literais de stringnewlines, Ponto e vírgula opcionais - Ponto e vírgula opcionais

usando para formatação de código, o texto de um ProgramNode JavaScript

iteração assíncrona em, O loop for/await, Node IsAsynchronous by Default-Node Is Asynchronous by Defaultbenefits de, Introdução ao JavaScript, JavaScript do lado do servidorcom tipo NodeBigInt, Inteiros de precisão arbitrária com BigIntbuffers, Buffersretornos de chamada e eventos em, Retornos de chamada e eventos em processos Nodechild, Trabalhando com processos filhos-fork()definindo o recurso de, JavaScript do lado do servidor com Nodeevents e EventEmitter, Manipulação de arquivos Events e EventEmitter, Trabalhando com arquivos-Working com diretórios

diretórios, Trabalhando com diretórios

metadados de arquivo, metadados de arquivocadeias de caracteres de modo de arquivo, Gravando arquivosoperações de arquivo, Operações de arquivovisão geral de, Trabalhando com arquivoscaminhos, descritores de arquivo e FileHandles, Caminhos, Descritores de arquivo e FileHandleslendo arquivos, Lendo arquivosgravando arquivos, Gravando arquivosClientes e servidores HTTP, Clientes e servidores HTTP-HTTPClients e servidoresinstalando, Explorando JavaScript, JavaScript do lado do servidor comAPI NodeIntl, A API de internacionalizaçãomódulos em, Módulos em módulos de estilo nó no Webservidores e clientes de rede não-HTTP, rede não-HTTPservidores e clientesparalelismo com, O nó é assíncrono por padrãodetalhes do processo, Processo, CPU e detalhes do sistema operacionalnoções básicas de programação, Noções básicas de programação de nó - o NodeGerenciador de pacotes

argumentos de linha de comando, argumentos de linha de comando e variáveis de ambienteaída do console, saída do consolevariáveis de ambiente, argumentos de linha de comando e

Variáveis de ambiente módulos, Módulos de
nó gerenciador de pacotes, O gerenciador de pacotes
de nó ciclo de vida do programa, Ciclo de vida do
programa documentação de referência,
Prefácio fluxos, Modo de fluxos pausados

iteração assíncrona em, Iteração assíncronavisão geral de,
Streams pipes, Pipes lendo com eventos, Lendo Streams com
Eventos tipos de, Fluxos gravando e manipulando contrapressão,
Gravando em Streams e Manipulando Backpressure thread de
trabalhador

canais de comunicação e MessagePorts,
CommunicationChannels e MessagePorts criando
trabalhadores e passando mensagens, Criando Workers e
passando mensagensvisão geral de, Worker
Threads compartilhando matrizes tipadas entre threads,
Compartilhando matrizes tipadas Entre threads transferindo
MessagePorts e matrizes tipadas, Transferindo MessagePorts
e Arrays Tipados

ambiente de execução do trabalhador, A execução do
trabalhadorEnvironmentNodeLists, Selecionando elementos
com seletores CSSpropriedades não herdadas, Introdução a
objetosoperador de desigualdade não estrita (!=)

expressões relacionais, Operadores de Igualdade e
Desigualdadeconversões de tipo, Conversões de operadores de
casos especiaisnormalização, Normalização Unicodevalor não
numérico (NaN), Aritmética em JavaScriptAPI de notificações,
Aplicativos Web Progressivos e Service WorkersGerenciador
de pacotes npm, Gerenciamento de pacotes com valores
npmnull, operador de coalescência nulo e indefinidonullish
(??), Primeiro Definido (??)Tipo de número

Formato de ponto flutuante de 64 bits, NúmerosNúmeros inteiros
de precisão arbitrária com BigInt, Precisão ArbitráriaInteiros com
BigIntaritmética e matemática complexa, Aritmética em
JavaScript-Aritméticaem JavaScriptErros binários de ponto
flutuante e arredondamento, Ponto Flutuante Binário e Erros de
Arredondamentodatas e horas, Datas e HorasLiterais de ponto
flutuante, Literais de Ponto Flutuante

literais inteiros, Literais inteirosseparadores em literais numéricos, Função Number() de literais de ponto flutuante, Conversões explícitas, Função Explicit ConversionsNumber.isFinite(), Aritmética em JavaScriptnumbers, formatação para internacionalização, Formatação de números-Formatação de númerosliterais numéricos, Números

O

literais de objeto sintaxe estendida para, Sintaxe Literal de Objeto Estendida- PropriedadeGetters e Settersvisão geral de, Inicializadores de Objeto e Matrizforma mais simples de, Literais de Objetonomes de propriedades de objeto, Leitura e Gravação de Elementos de Matrizprogramação orientada a objetos

definição de termo, Visão geral e Definiçõesexemplo de, Um tour pela função JavaScriptObject.assign(), Função Extending ObjectsObject.create(), Object.create(), Método Property AttributesObject.defineProperties(), Método Property AttributesObject.defineProperty(), Método Property AttributesObject.entries(), for/of com objetos

Object.getOwnPropertyNames(), Enumerando a função
PropertiesObject.getOwnPropertySymbols(), Enumerando o
método PropertiesObject.keys, for/of com a função
objectsObject.keys(), Enumerando PropertiesObject.prototype,
Protótipos, Objetos de métodosobjetos

Objeto Argumentos, O Objeto Argumentosobjetos semelhantes
a matrizes, Objetos semelhantes a Matriz-Objetos semelhantes a
matrizescriando, Criando Objetos-Objeto.create()excluindo
propriedades, Excluindo propriedadesenumerando propriedades,
Enumerando propriedadessintaxe literal de objeto estendida,
Sintaxe literal de objeto estendida-Getters e Setters de
propriedadesestendendo objetos, Estendendo
objetosimplementando objetos iteráveis, Implementando objetos
iteráveis-Implementando objetos iteráveisintrodução a,
Objetosprogramação modular com, Módulos com Classes,
Objetos eClosuresreferências de objetos mutáveis, Valores
Primitivos Imutáveis e Referências de Objetos Mutáveis,
Introdução a Objetosnomeando propriedades dentro, Palavras
Reservadasexpressões de criação de objetos, Expressões de
Criação de Objetos

extensibilidade de objeto, Extensibilidade de objetométodos de objeto, Métodos de objeto - O método `toJSON()` visão geral de, Um tour pelo JavaScript, Visão geral e definições - Visão geral e definições Consultando e definindo propriedades, Consultando e definindo propriedades - Erros de acesso a propriedadesserializando objetos, Serializando objetos testando propriedades, Testando propriedades no evento de mensagem, Recebendo mensagens de um WebSocket, WorkerObjects-O objeto global em trabalhadores, `postMessage()`, MessagePorts e MessageChannels, Mensagens de origem cruzada com `postMessage()` operadores

operadores aritméticos, Um tour pelo JavaScript, Aritmética em JavaScript-Aritmética em JavaScript, Expressões aritméticas-Operadores bit a bit, Expressões de atribuição-Atribuição com Operação operadores binários, Número de operandos operadores de comparação, Operadores de comparação operadores de igualdade e desigualdade, Igualdade e desigualdade Operadores operadores de igualdade, Um tour pelo JavaScript formando expressões com, Um tour pelo JavaScript, Expressões e operadores operadores lógicos, Um tour pelo JavaScript, Expressões Lógicas-

Lógico NÃO

(!)Operadores diversos

operador await, operador await Operador (,), Operador vírgula
(,)operador condicional (?:"), operador condicional (?:"operador delete,
operador delete operador primeiro definido (??), primeiro definido
(??)typeof operator, O typeof Operatoroperador vazio, O operador
voidNúmero de operandos, Número de operandosOperando e tipo de
resultado, Tipo de operando e resultadoassociatividade do operador,
Associatividade do operadorprecedência do operador, Precedência do
operadorefeitos colaterais do operador, Efeitos colaterais do
operadorordem de avaliação, Ordem de avaliaçãovisão geral de, Visão
geral do operadoroperadores de postfix, Ponto e vírgula
opcionaloperadores relacionais, Um tour pelo JavaScript, Expressões
relacionais-A instânciade Operadortabela de, Operador Visão
geraloperadores ternários, Número de operandosponto-e-vírgula
opcional, Ponto-e-vírgula opcional-Ponto-e-vírgula opcional

overflow, Aritmética em JavaScriptpropriedades
próprias, Introdução a Objetos, Herança

P

gerenciador de pacotes (Node), O Gerenciador de Pacotes do
Nó, PackageManagement com npmparallelization, O nó é
assíncrono por padrãoparametrização, Função
FunctionsparseFloat(), Função Explicit ConversionsparseInt(),
Explicit Conversionspasswords, Storagepaths, Paths e
Polygons-Paths e Polygonspattern matching

Definindo expressões regulares

alternação, agrupamento e referências, alternância,
agrupamento e referênciasclasses de caracteres, classes de
caracteressinalizadores, sinalizadorescaracteres literais,
caracteres literaisdeclarações lookbehind, especificando a
posição de correspondênciacapturas de grupo nomeado,
alternância, agrupamento e referênciasrepetição não
gananciosa, repetição não gananciosaespecificações de
padrão, definindo expressões regulares

caracteres de repetição, Repetição especificando posição de correspondência, Especificando posição de correspondência Classes de caracteres Unicode, Classes de caracteresvisão geral de, Correspondência de padrões com expressões regulares Símbolos de correspondência de padrões Classe RegExp exec(), propriedade exec()lastIndex e reutilização RegExp, visão geral exec()das propriedades RegExp Class RegExp, método RegExp properties test(), métodos test()string para

match(), match()matchAll(), matchAll()replace(), replace()search(), Métodos de string para correspondência de padrões split(), split()syntaxe para, correspondência de padrões cores, padrões e gradientes API de solicitação de pagamento, criptografia e APIs relacionadas APIs de desempenho, decapagem de desempenho, serialização JSON e análise

pixels, Coordenadas do documento e coordenadas da viewport,
PixelManipulations mais (+)

operador de adição e atribuição (+=), Atribuição com Operador de adição, Aritmética em JavaScript, A concatenação + Operator string, Literais de String, Trabalhando com Strings, As conversões + Operator type, Conversão de operador de caso especial Operator aritmético sunário, Operadores Aritméticos Unários polígonos, Caminhos e Polígonos-Caminhos e Método Polygon.pop(), Pilhas e Filas com push(), pop(), shift(), eunshift() evento popstate, Categorias de Eventos, Gerenciamento de Histórico withpushState() - Networking positive zero, Aritmética em JavaScript possessivos, Literais de cadeia de caracteres operadores de postfixo, Método opcional Ponto-e-vírgula postMessage(), postMessage(), MessagePorts e MessageChannels Mais bonito, Formatação JavaScript com expressões primárias, Expressões primárias tipos primitivos

Valores de verdade booleanos, Valores booleanos-Valores booleanos

valores primitivos imutáveis, Valores Primitivos Imutáveis e Referências de Objetos Mutáveis

Tipo de número, Números-Datas e Horas

visão geral e definições, Visão geral e definições

Tipo de string, Text-Tagged modelo literals

printprops(), Declarações de função

campos privados, Campos públicos, privados e estático

procedimentos, Funções

programas

tratamento de erros, Erros de Programa

execução de JavaScript, Execução de Programas JavaScript

Linha do tempo JavaScript do lado do cliente

modelo de threading do lado do cliente, threading JavaScript do lado do cliente

modelolinha do tempo do lado do cliente, linha do tempo do JavaScript do lado do cliente

entrada e saída, Entrada e saída do programa

PWAs (aplicativos da Web progressivos), aplicativos da Web progressivos e ServiceWorkers

Cadeias de promessas, Promessas, encadeamento de promessas - encadeamento da função

Promises

PromisesPromise.all(), Promessas em ParallelPromises

encadeamento de promessas, encadeamento de promessas - encadeamento de promessas

tratamento de erros com, tratamento de erros com promessas, mais em

Promessas e erros - os métodos catch e finally

Fazendo promessas, fazendo promessas - promessas em sequência com base em outras promessas, Promessas baseadas em outras promessas com base em valores síncronos, Promessas baseadas em valores síncronos do zero, Promessas do zerovisão geral de, Promessas operações paralelas, Promessas em paraleloPromessas em sequência, Promessas em sequência-Promessas em Sequênciaresolvendo promessas, resolvendo promessas-Mais sobre promessas e Errosretornando de retornos de chamada de promessas, Os métodos catch e finallyterminologia, Manipulando erros com promessasusando, Usando promessas-Manipulando erros com Promisesproperties

nomes de propriedades calculadas, nomes de propriedades calculadas acesso a propriedades condicionais, acesso a propriedades condicionais copiando de um objeto para outro, estendendo objetosdefinindo suas próprias propriedades de função, definindo suas própriaspropriedades de função definição de termo, visão geral e definiçõesexcluindo, excluindo propriedadesenumerando propriedades, enumerando propriedades

herdando, Herançanomeação, Símbolos, Introdução a Objetos, Símbolos como Nomes de Propriedadespropriedades não herdadas, Introdução a Objetoserros de acesso à propriedade, Erros de Acesso à Propriedadeexpressões de acesso à propriedade, Expressões de Acesso à Propriedadeatributos de propriedade, Introdução a Objetos, Atributos de Propriedade-Atributos de Propriedadedescritores de propriedade, Atributos de Propriedadegetters e setters de propriedade, Getters e Setters de Propriedadeconsultando e configurando, Consultando e Configurando Propriedades-PropertyAccess Errotestando, testando propriedadespropriedades de matriz tipada, métodos de matriz tipada e propriedadespropertyIsEnumerable() método, testando propriedadesherança prototípica, introdução a objetos, herançacadeias de protótipos, protótipoprotótipos, protótipos, herança, a propriedade do protótipo, classes e protótipos, o atributo do protótipoinvariantes de proxy, invariantes de proxy, objetos de proxy-invariantes de proxynúmeros pseudoaleatórios, criptografia e APIs relacionadascampos públicos, Público, Campos privados e estáticos

API Push, aplicativos Web progressivos e método Service Workers
push(), um tour pelo JavaScript, pilhas e filas com push(), pop(), shift() e unshift()

Q

método quadraticCurveTo(), método Curves
querySelector(), seleção de elementos com seletores CSS
método querySelectorAll(), seleção de elementos com seletores CSS
ASPAS

aspas duplas ("), Literais de cadeia de caracteres aspas simples ('), Literais de cadeia de caracteres

R

React, JSX: Expressões de marcação em JavaScript
rectangles, Rectangles
funções recursivas, Function Invocation
geradores recursivos, método yield* e Recursive Generators
reduce(), tipos de referência reduce() e reduceRight(), Immutable Primitive Values e Mutable Object References
Reflect API, The Reflect API-The Reflect API
Reflect.ownKeys(), Enumerando a classe Properties
RegExp

método exec(), propriedade exec().lastIndex e reutilização de RegExp, visão geral de exec()As propriedades RegExp ClassRegExp, propriedades RegExpmétodo test(), tipo test()RegExp, Visão geral e definições, Correspondência de padrões, Correspondência de padrões com expressões regulares (consulte também correspondência de padrões)expressões regulares, Correspondência de padrões com expressões regulares

(veja também correspondência de padrões)expressões relacionais, Expressões relacionais-A instância de operadores relacionais Operator, Um tour pelo método JavaScriptreplace(), Trabalhando com a função Stringsrequire(), Importações de nóspalavras reservadas, Palavras reservadas, Expressões primáriasparâmetros de descanso, Parâmetros de descanso e listas de argumentos de comprimento variávelinstruções de retorno, valores de retorno, método Functionsreturn(), "Fechando" um iterador: o método de retorno, métodos Thereturn() e throw() de um método Generatorreverse(), um Tour de JavaScript, erros de arredondamento reverse(), erros de ponto flutuante binário e erros de arredondamento

same-origin policy, A política de mesma origem gráficos vetoriais escaláveis (SVG), SVG: Gráficos vetoriais escaláveis - Criando imagens SVG com JavaScript

criando imagens SVG com JavaScript, Criando imagens SVG com JavaScriptvisão geral de, SVG: Gráficos vetoriais escaláveis scripting SVG, Script SVG SVG em HTML, SVG em HTML Screen API de orientação, APIs de dispositivos móveis deslocamentos de rolagem, Coordenadas do documento e coordenadas da janela de visualização rolagem, método ScrollingscrollTo(), método Scrollingsearch(), Métodos de string para correspondência de padrões segurança

armazenamento do lado do cliente, Armazenamento objetivos concorrentes da programação da Web, O modelo de segurança da Web Compartilhamento de recursos de origem cruzada (CORS), A política de mesma origem, Solicitações de origem cruzadas script entre sites (XSS), Script entre sites APIs de criptografia, Criptografia e APIs relacionadas defesa contra código mal-intencionado, O que o JavaScript não pode fazer ataques de negação de serviço, Gravando em fluxos e manipulando Backpressure

política de mesma origem, a política de mesma origem recursos da plataforma da web para investigar, segurança ponto-e-vírgula (;), ponto-e-vírgula opcional-ponto e vírgula opcional informações confidenciais, API StorageSensor, APIs de dispositivo móvel serialização, serialização de objetos, serialização e análise de JSON, gerenciamento de histórico com pushState() eventos enviados pelo servidor, eventos enviados pelo servidor - eventos enviados pelo servidor JavaScript do lado do servidor, JavaScript em navegadores da Web, JavaScript do lado do servidor JavaScript com NodeServiceWorkers, Progressive Web Apps and Service Propriedade Workers sessionStorage, localStorage e sessionStorage Set classe, for/of com Set e Map, The Set Class-The Set ClassSet objetos, Overview e DefinitionsSet() construtor, The Set Class setInterval() função, Timers sets e mapas

definição de conjuntos, a classe Set ClassMap, a classe Map - a visão geral da classe Map de, a classe Sets e MapsSet, a classe Set - as classes Set ClassWeakMap e WeakSet, métodos WeakMap e WeakSet setter, Property Getters e Setters, Getters, Setters e outros

Função Method FormsetTimeout(), Timers, método
TimersetTransform(), Transformações do sistema de
coordenadas shadow DOM, Shadow DOM-Shadow DOM
APIshadowsshadows, Operador esquerdo Shadowshift (<<),
Operadores bit a bitshift para a direita com operador de sinal (>>),
Operadores bit a bitshift para a direita com operador de
preenchimento zero (>>>), Operadores bit a bitshift(), Pilhas e
filas com push(), pop(), shift() e métodos abreviados, Métodos
abreviados, Getters, Setters e outrosMethod Formsefeitos laterais,
Lado do operador Efeitosaspas simples ('), Método String
Literalsslice(), método slice()some(), ordem de classificação
cada() e some(), Comparando o método Stringssort(), Invocação
condicional, sort()matrizes esparsas, Matrizes, Array
esparsoMétodo splice(), método splice()split(), operador
split()spread (...), Operador Spread, O operador Spread, Operador
TheSpread para chamadas de função, Iteradores e geradores

colchetes ([]), Um tour pelo JavaScript, Trabalhando com strings, Inicializadores de objetos e matrizes, Consultando e definindo propriedades, Lendo e gravando elementos de matriz, Strings como matrizesbiblioteca padrão (consulte a biblioteca padrão JavaScript) blocos de instruções, instruções compostas e vaziasdeclarações (consulte também declarações)

instruções compostas e vazias, instruções condicionais Compound e EmptyStatements, Statements-switchcontrol structures, A Tour of JavaScript-A Tour of JavaScript, Statementsexpression statements, Expression Statementsversus expressions, A Tour of JavaScriptif/else statement, Boolean Valuesjump statements, Statements, Jumps-try/catch/finallyquebras de linha e, Ponto e vírgula opcional-Ponto e vírgula opcionallista de, Resumo de instruções JavaScriptloops, Instruções, Loops-for/ Declarações Diversas

instruções do depurador, depuradorusar diretiva estrita, "usar estrito"com instruções, Instruções diversasvisão geral de, Instruções

separando com ponto-e-vírgula, ponto-e-vírgula opcional-
opcionalPonto-e-vírgula instruções throw, instruções
throwtry/catch/finally, instruções try/catch/finally-
try/catch/finallyyyield, yield, O valor de uma expressão
yieldcampos estáticos, campos públicos, privados e
estáticosmétodos estáticos, métodos estáticosarmazenamento,
Storage-IndexedDB

cookies, CookiesIndexedDB, IndexedDBlocalStorage e
sessionStorage, localStorage e sessionStorage overview of,
Storage security and privacy, Storage streams (Node), Modo de fluxos
pausados

iteração assíncrona em, Iteração assíncronavisão geral de,
Streams pipes, Pipeslendo com eventos, Lendo Streams com
Eventos tipos de, Streamsgravando e manipulando contrapressão,
Gravando em Streams eManipulando Backpressureoperador de
igualdade estrita (====)

valores booleanos, valores booleanos

visão geral de, Operadores de igualdade e
desigualdade de comparação de strings, Trabalhando com conversões
do tipo Strings, Visão geral e definições, Conversões
e Igualdade de modo estrito

aplicação padrão de, Classes com a palavra-chave class, Modules in
ES6, Módulos JavaScript no operador Web delete e, O operador
delete excluindo propriedades, Excluindo propriedades eval()
função, Declarações de função eval() estritas, Declarações de
função invocação de função, Chamada de função versus modo não
estrito, "use estrito" - "use estrito" optando por, Introdução ao
JavaScript TypeError, Erros de acesso à propriedade, Extensibilidade
de objetos variáveis não declaradas e, Declarações de variáveis com
var with instrução e, com, Definindo literais de string de atributos do
manipulador de eventos

sequências de escape em, Sequências de escape em literais de
cadeia de caracteres visão geral de, Literais de cadeia de
caracteres função String(), Função explícita
ConversionsString.raw(), Literais de modelo marcados strings

conversões de matriz para string, conversões de matriz para string
caracteres e pontos de código, métodos de texto para correspondência de padrões

match(), match()matchAll(), matchAll()replace(),
replace()search(), Métodos de string para
correspondência de padrõessplit(), split()visão
geral de, literais de string, literais de stringstrings
como matrizes, strings como matrizes trabalhando
com

acessando caracteres individuais, Trabalhando com StringsAPI
para, Trabalhando com Stringscomparando, Trabalhando com
Strings, Comparando Stringsconcatenação, Trabalhando com
Stringsdeterminando comprimento, Trabalhando com
Stringsimmutability, Trabalhando com Stringsalgoritmo de clone
estruturado, Gerenciamento de Histórico com subarrays
pushState(), Subarrays com subclasses slice(), splice(), fill() e
copyWithin()subclasses

hierarquias de classe e classes abstratas, hierarquias de classe e

Classes abstratas-Resumo delegação versus herança, Delegação em vez de herança visão geral de, Subclasses protótipos e, Subclasses e Protótipos com cláusula extends, Subclasses com extensões e super-Subclasses com extensões e supersub-rotinas, Funções operador de subtração (-), Aritmética em JavaScript pares substitutos, TextSVG (consulte gráficos vetoriais escaláveis (SVG)) instruções switch, switch-switchSymbol.asyncIterator, Symbol.iterator e Symbol.asyncIteratorSymbol.hasInstance, Symbol.hasInstanceSymbol.isConcatSpreadable, Symbol.isConcatSpreadableSymbol.iterator, SymbolsSymbol.species, Symbol.species-Symbol.speciesSymbol.toPrimitive, Symbol.toPrimitiveSymbol.toStringTag, Symbol.toStringTagSymbol.unscopables, Symbol.unscopablesSymbols

definição de extensões de linguagem, Visão geral e definições nomes de propriedades, Símbolos, Símbolos como nomes de propriedades

Símbolos conhecidos, Símbolos conhecidos execução de script síncrona, Quando os scripts são executados: assíncrono e diferidos sintaxe

estruturas de controle, Um tour pelo JavaScript-Um tour pelo JavaScript declarando variáveis, Um tour pelo JavaScript Comentários em inglês, Um tour pelo JavaScript, Um tour pelo JavaScript igualdade e operadores relacionais, Um tour pelo JavaScript expressions

formando com operadores, Um tour pela expressão JavaScript initializer, Um tour pelo JavaScript extended para literais de objeto, Extended Object Literal Syntax-Property Getters and Setters functions, Um tour pela estrutura lexical JavaScript, Lexical Structure-Summary

diferenciação de maiúsculas e minúsculas, O texto de um programa JavaScript comentários, Comentários identificadores, O texto de um programa JavaScript - Identificadores e palavras reservadas quebras de linha, O texto de um programa JavaScript literais, Literais palavras reservadas, Palavras reservadas, Expressões primárias ponto e vírgula, Ponto e vírgula opcionais - Ponto e vírgula opcional

espaços, O texto de um programa JavaScriptConjunto de caracteres Unicode, Sequências de escape Unicode-UnicodeNormalizaçãooperadores lógicos, Um tour por JavaScriptmétodos, Um tour por JavaScriptobjects

acessando propriedades condicionalmente, Um tour por JavaScriptdeclarando, Um tour por JavaScriptmétodos abreviados, Métodos abreviadosinstruções, Um tour por JavaScriptvariáveis, atribuindo valores a, Um tour por JavaScript

T

guias, O texto de um programa JavaScriptLiterais de modelo marcados, Literais de modelo marcados, Tags de modeloliterais de modelo, Literais de modelo, Operadores de tagsternário de modelo, Número de operandostest() método, test()text

desenho em Canvas, sequências de escape de texto em literais de string, sequências de escape em StringLiteralscorrespondência de padrões, correspondência de padrões

literais de cadeia de caracteres, Literais de cadeia de caracteresrepresentação de tipo de string, Textliterais de modelo, Literais de modelotrabalhando com cadeias de caracteres, Trabalhando com cadeias de caractereseditores de texto

normalização, Normalização Unicodeusando com Node, Hello Worldtext styles, Text styles.then() método, Usando promessas, encadeando promessas, mais sobre promessas e erroesta palavra-chave, Um tour pelo JavaScript, Expressões primárias, FunçãoInvocationthreading, Worker Threads e mensagens, Progressive Web Apps e Service Workers (consulte também Worker API)Gráficos 3D, Gráficos em um <canvas>instruções throw, throw, Classes de errométodo throw(), Os métodos return() e throw() de um geradorfusos horários, Formatação de datas e horários, Temporizadores, Timerscarimbos de data/hora, Datas e horas, Método TimestamptoString(), Formatação e análise de data Método StringstoExponential(), Método Explicit ConversiontoFixed(), Conversões explícitas

método `toISOString()`, Formatação e análise de strings de data, método `JSONCustomizationstoJSON()`, método `toJSON()`, método `JSONCustomizationstoLocaleDateString()`, formatação e análise de strings de data, método de formatação de datas e horários `toLocaleString()`, método `toLocaleString()`, matriz para `StringConversions`, formatação e análise de strings de data `toLocaleTimeString()`, formatação e análise de strings de data, formatação de datas e horários ferramentas e extensões, ferramentas JavaScript e Extensions-EnumeratedTypes e uniões discriminadas

empacotamento de código, agrupamento de código Formatação JavaScript com Prettier, Formatação JavaScript com Prettier Extensão de linguagem JSX, JSX: Expressões de marcação em JavaScript-JSX: Expressões de marcação em JavaScript linting com ESLint, Linting com ESLint visão geral de, Ferramentas e extensões JavaScript gerenciamento de pacotes com npm, Gerenciamento de pacotes com npm transpilação com Babel, Transpilação com Babel verificação de tipo com Flow, Verificação de tipo com Flow-EnumeratedTypes e Uniões discriminadas

tipos de matriz, tipos de matriz tipos de classe, tipos de classe tipos enumerados e uniões discriminadas, tipos enumerados

Tipos e Uniões Discriminadas tipos de função, Tipos de função instalando e executando, Instalando e executando fluxo tipos de objeto, Tipos de objeto outros tipos parametrizados, Outros tipos parametrizadosvisão geral de, Verificação de tipo com fluxo tipos somente leitura, Tipos somente leitura aliases de tipo, Aliases de tipoScript versus fluxo, Verificação de tipo com fluxo tipos de união, Tipos de união usando anotações de tipo, Usando anotações de tipoTeste de unidade com Jest, Teste de unidade com JesttoMétodo Precision(), Conversões explícitas paraString() método, Valores Booleanos, Conversões Explícitas, Métodos ToString() e valueOf(), O Operador +, Igualdade com typeconversion, O Método toString(), O Método toString(), Formatação e Análise de Data StringstoTimeString() método, Formatação e Análise de Data StringstoTimeString() método, Trabalhando com o método StringstoUTCString(), Formatação e Análise de Data Stringstransformations, Sistema de Coordenadas Transformations Transformation example translate(), Transformações do Sistema de Coordenadas

translucidez, translucidez e composição transpilação, transpilação com valores de BabelTruthy, valores booleanos instruções try/catch/finally, try/catch/finally-try/catch/finally verificação de tipo, verificação de tipo com tipos enumerados por fluxo e uniões discriminadas

tipos de matriz, tipos de matriz tipos de classe, tipos de classe tipos enumerados e uniões discriminadas, tipos enumerados e uniões discriminadas tipos de função, tipos de função instalando e executando o fluxo, instalando e executando fluxo tipos de objeto, tipos de objeto outros tipos parametrizados, outros tipos parametrizados visão geral de, Verificação de tipo com fluxo tipos somente leitura, tipos somente leitura aliases de tipo, aliases de tipoScript versus fluxo, Verificação de tipo com fluxo tipos de união, tipos de união usando anotações de tipo, usando tipo Conversões de tipo de anotações

igualdade e, Conversões e igualdade, Igualdade com typeconversion

conversões explícitas, Conversões explícitas
dados financeiros e científicos, Conversões

explícitasConversões implícitas, Conversões
explícitasobjeto de conversões primitivas

algoritmos para, Conversões de objeto para primitivo, Algoritmos de
conversão de objeto para primitivosObjeto-para-booleano,
Conversões de objeto-para-booleanoObjeto-para-número,
Conversões de objeto-para-númeroObjeto-para-string, Conversões
de objeto-para-stringConversões de operador de caso especial,
Operador de caso especialConversões para String() e valorOf()
métodos, Os métodos toString() e evaluateOf() visão geral de,
Conversões de tipoMatrizes tipadas

criando, criando matrizes tipadasDataView e endianness,
DataView e endiannessmétodos e propriedades, métodos e
propriedades de matrizes tipadasvisão geral de, Matrizes tipadas
e dados bináriosversus matrizes regulares,
matrizescompartilhamento entre threads, compartilhamento de
matrizes tipadas entre threadstipos de matriz tipada, tipos de
matriz tipadausando, usando matrizes tipadas

typeof operator, O typeof

Operatortypes

objeto global, O Objeto Global-O Objeto GlobalTipo de número, Números-Datas e Horasobjetos (ver objetos)visão geral de, Tipos, Valores e Variáveis-Visão geral

eDefiniçõesprimitivo, Visão geral e DefiniçõesRegExp,

Correspondência de padrões-Correspondência de

padrõesstrings, Literais de modelo com marcação de

textoSímbolos, Símbolos-Símbolosconversões de tipo,

Conversões de tipo-Objeto-Primitivoalgoritmos de

conversãoTypeScript, Verificação de tipo com fluxo

No

Uint8Array, Tipos de Array Tipado, Corpos de resposta de

streaming, Operadores Buffersunary

operadores aritméticos, operadores aritméticos

unáriosOperador booleano NOT (!), valores

booleanosSupporte JavaScript para, número de

operandosvariáveis declaradas, declarações de

variáveis com valores indefinidos, nulos e indefinidos

underflow, Aritmética em JavaScript sublinhado (_), Identificadores e palavras reservadas sublinhados, como separadores numéricos (_), Função Literal `unescape()` de ponto flutuante, Funções de URL legadas e evento de rejeição não tratado, Erros de programa Conjunto de caracteres Unicode sequências de escape, sequências de escape Unicode, sequências de escape em literais de cadeia de caracteres Cadeias de caracteres JavaScript, Texto normalização, Normalização Unicodevisão geral de, Unicode correspondência de padrões, Classes de caracteres caracteres de espaço, O texto de um programa JavaScript teste de unidade, Teste de unidade com o método `shift()`, Pilhas e filas com `push()`, `pop()`, `shift()` e `unshift()` APIs de URL, APIs de URL - Funções de URL legadas usar diretiva estrita

aplicação padrão do modo estrito, Classes com a palavra-chave `class`, Módulos no ES6, Módulos JavaScript no operador `Webdelete` e, A função `delete` Operador `eval()`, Declarações de função `eval()` estritas, Declarações de função

função invocação, Função Invocaçãoooptando no modo estrito, Introdução ao JavaScriptmodo estrito versus não estrito, "use strict"- "use strict"TypeError, Erros de acesso à propriedade, Extensibilidade de objetosvariáveis não declaradas e, Declarações de variáveis com instrução varwith e, com, Configurando atributos do manipulador de eventosuse o modo estrito, Introdução ao JavaScript

e variáveis globais, Declarações de variáveis com propriedades varexwing, Excluindo propriedadesCodificação UTF-16, Texto

V

método valueOf(), os métodos toString() e valueOf(), os valores do método thevalueOf()

atribuindo, Um tour por JavaScriptvalores booleanos, Valores booleanos-Valores booleanosfalsy e verdadeiros, Valores booleanosfunções como valores, Funções como valores-Definindo seus própriosPropriedades de funçãovalores primitivos imutáveis, Valores primitivos imutáveis e referências de objetos mutáveisnulo e indefinido, nulo e indefinidovisão geral de, Tipos, Valores e Variáveis-Visão geral e

Definições tipos de, Um tour pelo JavaScript palavra-chave var,
Declarações de variáveis com var, const, let e varvarargs,
Parâmetros Rest e listas de argumentos de comprimento
variável funções de aridade variável, parâmetros Rest e
Variable-Length Lists de argumentos variáveis

diferenciação de maiúsculas e minúsculas, o texto
de um programa JavaScript declaração e atribuição
declarações com let e const, Declarações com let e const-
Declarações e tipos declarações com var, Declarações de variáveis
com var de atribuição de estruturação, Atribuição de
desestruturação-Atribuição de desestruturação visão geral de, Um
tour por JavaScript variáveis não declaradas, Declarações de
variáveis com var definição de termo, Declaração e atribuição de
variável, Declarações de variáveis com var naming, Palavras
reservadas visão geral de, Tipos, Valores e Variáveis-Visão geral
e Definições escopo de, Variável e escopo constante, funções
variadic aninhadas, parâmetros REST e Argument Lists de
comprimento variável

fluxos de vídeo, APIs de mídiaviewport, Coordenadas do documento e coordenadas da janela de visualização, ViewportSize, Tamanho do conteúdo e Posição de rolagemoperador void, O operador void

Em

Classe WeakMap, classe WeakMap e WeakSetWeakSet, API de autenticação WeakMap e WeakSetWeb, criptografia e APIs relacionadasambiente de host do navegador da Web

APIs assíncronas, EventosAPIs de áudio, APIs de áudio - A API WebAudioBenefícios do JavaScript, JavaScript em navegadores da WebAPI Canvas, Gráficos em uma <canvas>manipulação de pixels

dimensões e coordenadas da tela, Dimensões e coordenadas da telarecorte, Recortetransformações do sistema de coordenadas, Sistema de coordenadasTransformações- Exemplo de transformaçãooperações de desenho, Operações de desenho da telaatributos gráficos, Atributos gráficos, Salvando e restaurandoestado gráficovisão geral de, Gráficos em <canvas>caminhos e polígonos, Caminhos e polígonos

manipulação de pixels, manipulação de pixelsgeometria e rolagem de documentos, geometria do documento e tamanho da janela de visualização de rolagem, tamanho do conteúdo e posição de rolagem

Pixels CSS, Coordenadas do documento e coordenadas da janela de visualizaçãodeterminando o elemento em um ponto,
Determinando o elemento em um pontoCoordenadas do documento e coordenadas da janela de visualização, Coordenadas do documento e coordenadas da janela de visualizaçãoonconsultando a geometria dos elementos, Consultando a geometria de um elementorolagem, Rolagemtamanho da janela de exibição, tamanho do conteúdo e posição de rolagem,
Tamanho da janela de visualização, Tamanho do conteúdo e posição de rolagemeventos, Eventos-Despachando eventos personalizados

despachando eventos personalizados, Despachando eventos personalizadoscancelamento de eventos, Cancelamento de eventoscategorias de eventos, Categorias de eventosinvocação do manipulador de eventos, Invocação do manipulador de eventospropagação de eventos, Propagação de eventosvisão geral de, Eventosregistrando manipuladores de eventos, Registrando manipuladores de eventosAPIs legadas, JavaScript em navegadores da Weblocalização, navegação e histórico, Local, Navegação e

Gerenciamento de histórico-histórico com pushState()

histórico de navegação, Histórico de navegação carregando novos documentos, Carregando novos documentosvisão geral de, Localização, Navegação e HistóricoExemplo de conjunto de Mandelbrot, Exemplo: O Resumo do Conjunto de Mandelbrote Sugestões para Leitura Adicionalnavegadores com reconhecimento de módulos, Módulos JavaScript na Webrede, Negociação de protocolo de rede

fetch() método, fetch()visão geral de, Networkingeventos enviados pelo servidor, Eventos enviados pelo servidorWebSocket API, WebSocketsvisão geral de, JavaScript em navegadores da Webgráficos vetoriais escaláveis (SVG), SVG: Gráficos vetoriais escaláveis - Criando imagens SVG com JavaScript

criando imagens SVG com JavaScript, Criando imagens SVGcom JavaScriptvisão geral de, SVG: Gráficos vetoriais escaláveisScriptando SVG, Criando scripts SVGSVG em HTML, SVG em HTMLscriptando CSS, Criando scripts de animações e eventos CSS-CSS

estilos CSS comuns, estilos CSScomputados com scripts, estilos computados

Animações e eventos CSS, Animações e eventos CSS classes
CSS, Classes CSS estilos embutidos, Estilos embutidos convenções
de nomenclatura, Estilos embutidos folhas de estilo de script,
Folhas de estilo de scripts Documentos de script, Documentos de
script - Exemplo: Gerando um sumário

estrutura e passagem do documento, Estrutura do documento
e Travessia gerando tabelas de conteúdo, Exemplo: Gerando um
sumário modificando conteúdo, Conteúdo do elemento como
HTML modificando estrutura, Criando, Inserindo e Excluindo
Nósvisão geral de, Documentos de script consultando e definindo
atributos, Atributos selecionando elementos do documento,
Selecionando elementos do documento armazenamento, Storage-
IndexedDB

cookies, Cookies IndexedDB,
IndexedDB localStorage e sessão Armazenamento,
localStorage e sessão Armazenamento visão geral
de, Armazenamento segurança e privacidade,
Armazenamento

componentes da Web, Componentes da Web - Exemplo: um <search-box> componente da Web

elementos personalizados, Elementos

personalizados, Documento, Nós de fragmento, Usando

componentes da Web, Modelos HTML, Modelos

HTML, Visão geral de, Componentes da Web, exemplo de

caixa de pesquisa, Exemplo: uma <search-box>

WebComponents, shadow DOM, Shadow DOM, usando,

Usando componentes da Web, Recursos da plataforma da

Web para investigar

APIs binárias, APIs binárias, APIs de criptografia e

segurança, Criptografia e APIs relacionadas, eventos,

Eventos, HTML e CSS, HTML e CSS, APIs de mídia, APIs

de mídia, APIs de dispositivos móveis, APIs de dispositivos

móveis, APIs de desempenho, Desempenho, Aplicativos e

serviços da Web progressivos, Workers, Progressive Web

Apps e Service Workers, segurança,

Segurança, WebAssembly, WebAssembly

Recursos de objeto de janela e documento, Mais recursos de documento e janelanoções básicas de programação da Web

Modelo de objeto de documento (DOM), O modelo de objeto de documentoModelo - O modelo de objeto de documentoexecução de programas JavaScript, Execução de JavaScriptProgramas-Linha do tempo JavaScript do lado do clienteobjeto global em navegadores da web, O objeto global em navegadores da webJavaScript em <script> tags HTML, JavaScript em<script> tags HTML-Carregando scripts sob demandaerros de programa, erros de programaentrada e saída de programas, entrada e saída de programasscripts compartilhando namespaces, scripts compartilham um namespacesmodelo de segurança da web, The Web Security Model-Cross-sitescriptingworker threads e mensagens, Worker Threads e Messaging-Cross-Origin Messaging com postMessage()ferramentas de desenvolvedor da Web, Explorando JavaScriptWeb Manifest, Progressive Web Apps e Service WorkersWeb Workers API, Modelo de threading JavaScript do lado do cliente, WorkerThreadsWebAssembly, WebAssemblyWebAudio API, The WebAudio APIWebRTC, APIs de mídia

WebSocket API

criando, conectando e desconectando WebSockets, Criando, _____
conectando e desconectando WebSocketsvisão geral de,
WebSocketsnegociação de protocolo, Negociação de
protocolorecebendo mensagens, Recebendo mensagens de um
WebSocketenviando mensagens, Enviando mensagens por um
WebSocketwhile loops, instruções whilewith, Instruções
diversasAPI do trabalhador

mensagens de origem cruzada, mensagens de origem cruzada
compostMessage()erros, Erros no modelo de execução de
trabalhadores, modelo de execução de trabalhoimportando
código, importando código para um trabalhadorExemplo de
conjunto de Mandelbrot, exemplo: O conjunto de Mandelbrot -
resumo e sugestões para leitura adicionalmódulos, importando
código para um trabalhadorvisão geral de, Worker Threads e
MessagingpostMessage(), MessagePorts e
MessageChannels,postMessage(), MessagePorts e
MessageChannelsObjetos de trabalhador, objeto Worker
ObjectsWorkerGlobalScope, Objeto global em trabalhadores

atributo gravável, Introdução aos Objetos, Atributos de Propriedade

X

XMLHttpRequest API (XHR), fetch()XSS (cross-site scripting), A política de mesma origem

E

instruções yield, yield, O valor de um yield Expressionyield* palavra-chave, yield* e geradores recursivos

Com

zero

zero negativo, Aritmética em JavaScriptzero
positivo, Aritmética em JavaScriptmatrizes
baseadas em zero, Matrizes

Sobre o autor David Flanagan tem programado e escrito sobre JavaScript desde 1995. Ele mora com sua esposa e filhos no noroeste do Pacífico, entre as cidades de Seattle, Washington e Vancouver, Colúmbia Britânica. David é formado em ciência da computação e engenharia pelo Massachusetts Institute of Technology e trabalha como engenheiro de software na VMware.

ColofãoO animal na capa de JavaScript: O Guia Definitivo, Sétima Edição, é um rinoceronte-de-jáva (*Rhinoceros sondaicus*). Todas as cinco espécies de rinocerontes se distinguem por seu grande tamanho, pele grossa semelhante a uma armadura, pés de três dedos e chifre de focinho simples ou duplo. O Javanrhinoceros se assemelha ao rinoceronte indiano relacionado e, como acontece com essa espécie, os machos têm um único chifre. No entanto, os rinocerontes de Java são menores e têm texturas de pele únicas. Embora encontrados hoje apenas na Indonésia, os rinocerontes de Java já se espalharam por todo o sudeste da Ásia. Eles vivem em habitats de floresta tropical, onde pastam em folhas e gramíneas abundantes e se escondem de pragas de insetos, como moscas sugadoras de sangue, levantando-se até o focinho na água ou na lama.

O rinoceronte de Java tem em média cerca de 6 pés de altura e pode ter até 10 pés de comprimento, com adultos pesando até 3.000 libras. Como o rinoceronte-indiano, sua pele cinza parece ser separada em "placas", algumas delas texturizadas. A vida útil natural de um rinoceronte-de-jáva é estimada em 45 a 50 anos. As fêmeas dão à luz a cada 3-5 anos, após um período de gestação de 16 meses. Os bezerros pesam cerca de 100 libras quando nascem e ficam com suas mães protetoras por até 2 anos.

O rinoceronte é geralmente um animal um tanto abundante, sendo adaptável a uma variedade de habitats e na idade adulta não tendo predadores naturais. No entanto, os humanos os caçaram quase até a extinção. O folclore afirma que o chifre do rinoceronte possui poderes mágicos e afrodisíacos e, por causa disso, os rinocerontes são o principal alvo dos caçadores furtivos. O População de rinocerontes de Java é o mais precário: a partir de 2020, os cerca de 70 animais restantes desta espécie vivem, sob guarda, em Ujung Kulon

Parque Nacional, em Java, Indonésia. Essa estratégia parece estar ajudando a garantir a sobrevivência desses rinocerontes por enquanto, já que um censo de 1967 contou apenas 25.

Muitos dos animais nas capas da O'Reilly estão ameaçados de extinção; todos eles são importantes para o mundo.

A ilustração colorida na capa é de Karen Montgomery, baseada em uma gravura em preto e branco da Dover Animals. As fontes de capa são Gilroy e Guardian Sans. A fonte do texto é Adobe Minion Pro; a fonte do cabeçalho é Adobe Myriad Condensed; e a fonte do código é o Ubuntu Mono da Dalton Maag.