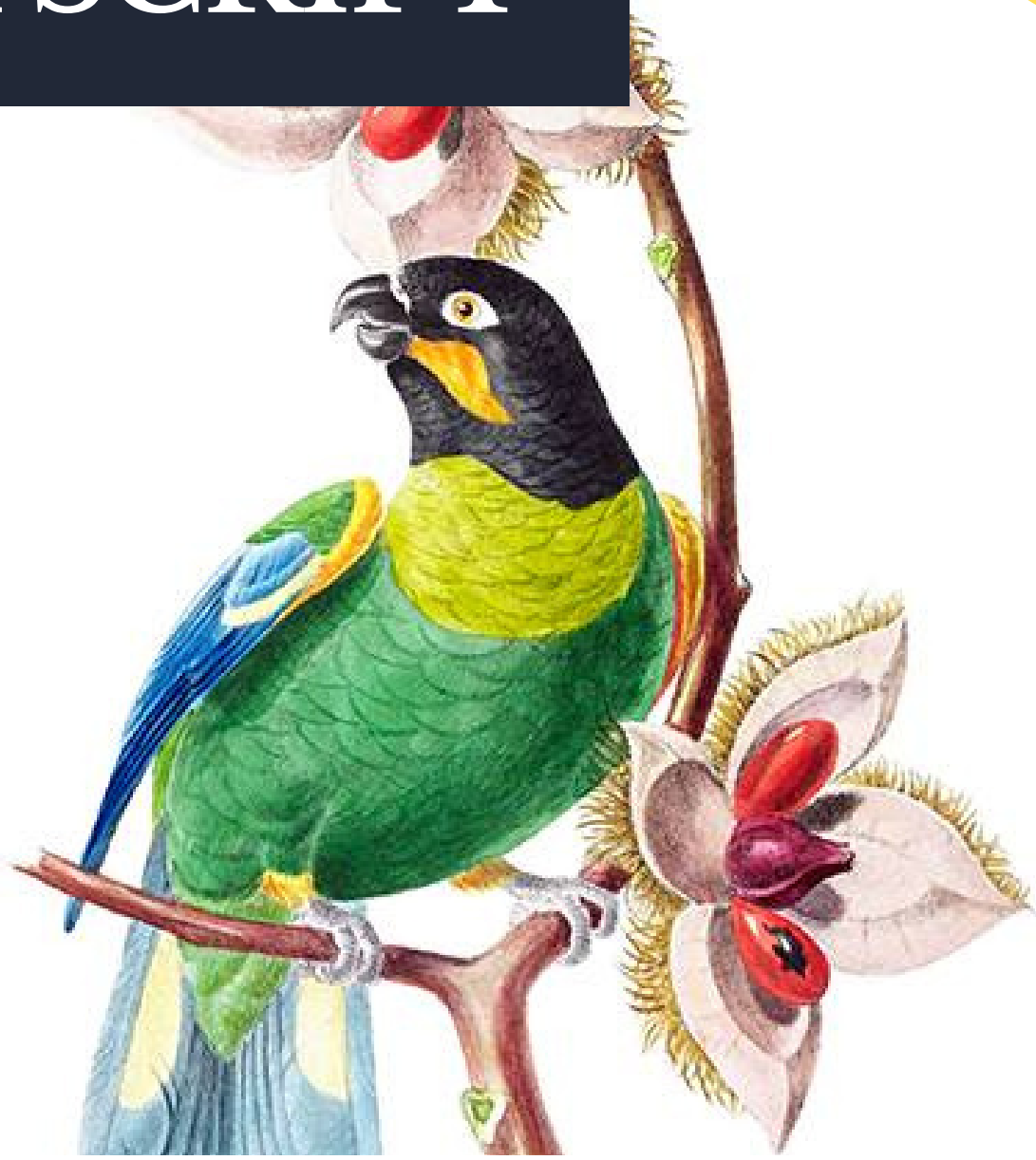
BRUNNA CROCHES

PARTE 3

JAVA SCRIPT



Artist Image :Descourtilz, Jean-Théodore | Dates:179?-1855

Guia Avançado: JavaScript

ABOUT ME

Brunna Croches

Developer Full Stack

Brunna Croches é Dev FullStack, advogada e empreendedora.

Apaixonada por tech, vem adquirido vasto conhecimento na área.

Desenvolveu projetos ricos em diversidade, buscando captar as próximas tendências e necessidades do mercado.

Neste e-book você aprenderá ou recapitulará de forma simplificada e otimizados conceitos de programação feito por ela.

let's share

SUMMARY

5.4 - APIS DA WEB > INTERFACES > EVENTARGET

EventTargeté uma interface DOM implementada por objetos que podem receber eventos DOM e tem que ouvir estes.

EventTarget

Resumo

`EventTarget`é uma interface DOM implementada por objetos que podem receber eventos DOM e tem que ouvir estes.

`Element`, `document`, e `window`são os mais comuns disparadores de eventos, mas outros objetos podem disparar eventos também, por exemplo `XMLHttpRequest`, `AudioNode`, `AudioContext`e outros.

Muitos disparadores de eventos (incluindo elements, documents, e windows) também suportam definir event handlersatravés `on...`propriedades e atributos.

Métodos

`EventTarget.addEventListener()`
Registra um tratamento para um tipo específico de evento sobre o `EventTarget`.

`EventTarget.removeEventListener()`
Remove um *event listener*do `EventTarget`.

`EventTarget.dispatchEvent()`
Dispatch an event to this `EventTarget`.

Especificações

Especificação	Status	Comentário
DOM The definition of 'EventTarget' in that specification.	Padrão em tempo real	Sem mundanças.
Document Object Model (DOM) Level 3 Events Specification The definition of 'EventTarget' in that specification.	Obsoleto	Alguns parâmetros agora são opcionais (`listener`), ou aceitam o valor `null`(`useCapture`).
Document Object Model (DOM) Level 2	Obsoleto	Definição inicial.

Events Specification The definition of 'EventTarget' in that specification.	

Compatibilidade com navegadores

Estamos convertendo nossos dados de compatibilidade para o formato JSON. Esta tabela de compatibilidade ainda usa o formato antigo, pois ainda não convertemos os dados que ela contém. **Descubra como você pode ajudar! (en-US)**

- Desktop
- Dispositivo móvel

Funcionalidade	Chrome	Firefox (Gecko)	Internet Explorer	Opera
Suporte Básico	1.0	1.0 (1.7 or earlier)	9.0	7

Funcionalidade	Android	Firefox Mobile (Gecko)	IE Mobile	Opera Mob
Suporte Básico	1.0	1.0 (1)	9.0	6.0

Additional methods for Mozilla chrome code

Mozilla extensions for use by JS-implemented event targets to implement on* properties. See also WebIDL bindings.

- void **setEventHandler**(DOMString type, EventHandler handler)
- EventHandler **getEventHandler**(DOMString type)

Veja também

- Referência de eventos (en-US)- os eventos disponíveis na plataforma.
- Guia do desenvolvedor sobre Eventos
- Interface `Event`

Element.addEventListener()

addEventListener() registra uma única espera de evento em um único alvo. O alvo do evento (en-US) pode ser um único elemento (en-US) em um documento, o documento (en-US) em si, uma janela (en-US), ou um XMLHttpRequest.

HTML Content

Element.addEventListener()

`addEventListener()` registra uma única espera de evento em um único alvo. O alvo do evento (en-US) pode ser um único elemento (en-US) em um documento, o `documento (en-US)` em si, uma `janela (en-US)`, ou um `XMLHttpRequest`.

Para registrar mais de uma espera de evento como alvo, chame `addEventListener()` para o mesmo alvo mas com diferentes tipos de evento ou captura de parâmetros.

Sintaxe

```
alvo.addEventListener(type,listener[, options]);
alvo.addEventListener(type,listener[, useCapture, wantUntrusted ]); // Gecko/M
```

`type`
Uma linha de texto que representa o tipo de evento (en-US) a ser esperado.

listener
O objeto que recebe uma notificação quando um evento do tipo especificado ocorre. Esse objeto precisa implementar a interface do `EventListener`, ou simplesmente executar uma função (en-US) JavaScript.

`useCapture` **Optional**
Se `true`, `useCapture` indica que o usuário deseja iniciar uma captura. Depois de iniciada a captura, todos os eventos do tipo especificado serão enviados à `listener` registrada antes de serem enviados à qualquer `EventTarget` abaixo dela na hierarquia de DOMs. Eventos que borbulharem para cima na hierarquia não acionarão a escuta designada a usar a captura. Veja Eventos DOM Nível 3 para uma explicação detalhada. Perceba que esse parâmetro não é opcional em todos os navegadores. Se não for especificado, `useCapture` é `false`.

`wantsUntrusted`
Se `true`, o evento pode ser acionado por conteúdo não-confiável. Veja Interação entre páginas com e sem privilégios.

> **Nota:** `useCapture` tornou-se opcional somente nas versões mais recentes dos principais navegadores; não era opcional antes do Firefox 6, por exemplo. Você deve especificar esse parâmetro para obter uma maior compatibilidade.

Exemplo

```
<!DOCTYPE html>
<html>
<head>
<title>Exemplo de Evento DOM</title>

<style>
#t { border: 1px solid red }
#t1 { background-color: pink; }
```

```
</style>

<script>
// Função para mudar o conteúdo de t2
function modifyText() {
  var t2 = document.getElementById("t2");
  t2.firstChild.nodeValue = "three";
}

// Função para adicionar uma espera de evento em t
function load() {
  var el = document.getElementById("t");
  el.addEventListener("click", modifyText, false);
}

document.addEventListener("DOMContentLoaded", load, false);
</script>

</head>
<body>

<table id="t">
   <tr><td id="t1">one</td></tr>
   <tr><td id="t2">two</td></tr>
</table>

</body>
</html>
```

View on JSFiddle

No exemplo acima, `modifyText()` é uma escuta para eventos de `click` registrados usando `addEventListener()`. Um clique em qualquer lugar da tabela irá borbulhar para cima até o manipulador e executar `modifyText()`.

Se você deseja passar parâmetros para a função de escuta, você deve usar uma função anônima.

```
<!DOCTYPE html>
<html>
<head>
<title>Exemplo de Evento DOM</title>

<style>
#t { border: 1px solid red }
#t1 { background-color: pink; }
</style>

<script>

// Função para mudar o conteúdo de t2
function modifyText(new_text) {
  var t2 = document.getElementById("t2");
  t2.firstChild.nodeValue = new_text;
}
```

```
// Função para adicionar uma espera de evento em t
function load() {
  var el = document.getElementById("t");
  el.addEventListener("click", function(){modifyText("four")}, false);
}
</script>

</head>
<body onload="load();">

<table id="t">
  <tr><td id="t1">one</td></tr>
  <tr><td id="t2">two</td></tr>
</table>

</body>
</html>
```

Notas

Por que usar `addEventListener`?

`addEventListener` é a maneira de registrar uma espera de evento como especificada no W3C DOM. Seus benefícios são os seguintes:

- Permite mais de um manipulador por evento. Isso é particularmente útil em bibliotecas DHTML (en-US) ou em extensões Mozilla que precisam trabalhar bem mesmo com outras bibliotecas/extensões sendo usadas.
- Te dá um pente-fino do estágio em que a espera de evento é ativada (captura ou borbulha).
- Funciona em qualquer elemento DOM, não só para elementos HTML.

Existe outra alternativa, uma maneira ultrapassada de registrar esperas de evento.

Adicionando uma espera de evento durante um disparo de evento

Se um EventListener for somado a um EventTarget enquanto está processando um evento, ele não será ativado pelas ações atuais, mas poderá ser ativado em um período posterior no fluxo de eventos, como na fase de borbulha.

Múltiplas esperas de evento idênticas

Se múltiplas esperas de evento idênticas forem registradas no mesmo EventTarget com os mesmos parâmetros, as versões duplicadas serão descartadas. Elas não fazem o EventListener ser disparado mais de uma vez, e, como as duplicatas são descartadas, elas não precisam ser removidas manualmente com o método removeEventListener.

O valor de this no manipulador

É preferível referenciar o elemento do qual a espera de evento foi disparada, como quando é usado um manipulador genérico para uma série de elementos similares. Quando anexar uma função

usando addEventListener(), o valor de this é mudado — perceba que o valor de this é passado para uma função a partir do disparador.

Nos exemplos acima, o valor de this em modifyText(), quando disparado pelo evento de clique, é uma referência à tabela 't'. Isso é um contraste do comportamento que acontece se o manipulador é adicionado ao HTML fonte:

```
<table id="t" onclick="modifyText();">
  . . .
```

O valor de this em modifyText(), quando disparado pelo evento de clique no HTML, será uma referência ao objeto global (no caso, a janela).

Nota: JavaScript 1.8.5 introduz o método `Function.prototype.bind() (en-US)`, que permite especificar o valor que deve ser usado como `this` para todas as chamadas à uma determinada função. Isso evita problemas quando não é claro o que `this` será, dependendo do contexto do qual a sua função for chamada. Perceba, entretanto, que é preciso manter uma referência da escuta à mão, para que depois você possa removê-la.

Este é um exemplo com e sem `bind`:

```
var Algo = function(elemento)
{
  this.nome = 'Algo bom';
  this.onclick1 = function(evento) {
    console.log(this.nome); // indefinido, porque this é a função de escuta do
  };
  this.onclick2 = function(evento) {
    console.log(this.nome); // 'Algo bom', porque this está como objeto Algo a
  };
  elemento.addEventListener('click', this.onclick1, false);
  elemento.addEventListener('click', this.onclick2.bind(this), false); // Truc
}
```

Outra solução é usar uma função especial chamada `handleEvent` para capturar quaisquer eventos:

```
var Algo = function(elemento)
{
  this.nome = 'Algo bom';
  this.handleEvent = function(evento) {
    console.log(this.nome); // 'Algo bom', porque this é o objeto Algo
    switch(evento.type) {
      case 'click':
        // seu codigo aqui...
        break;
      case 'dblclick':
        // seu codigo aqui...
        break;
    }
  };
  elemento.addEventListener('click', this, false); // Não this.handleEvent, so
```

```
    elemento.addEventListener('dblclick', this, false); // Não this.handleEvent,
  }
```

Internet Explorer antigos e attachEvent

Em versões do Internet Explorer anteriores ao IE9, você precisa usar `attachEvent` em vez do padrão `addEventListener`. Para dar suporte ao IE, o exemplo acima pode ser modificado para:

```
if (el.addEventListener) {
  el.addEventListener('click', modifyText, false);
} else if (el.attachEvent)  {
  el.attachEvent('onclick', modifyText);
}
```

Existe um porém com `attachEvent:` o valor de `this` será a referência ao objeto `window` em vez do elemento do qual foi disparado.

Uma maneira ultrapassada de registrar esperas de evento

`addEventListener()` foi introduzido com as especificações de Eventos DOM 2. Antes disso, esperas de evento eram registradas assim:

```
// Passe uma função de referência — não adicione '()' depois dela, o que chama
el.onclick = modifyText;

// Usando uma expressão de função
element.onclick = function() {
    // ... lógica da função ...
};
```

Esse método substitui as esperar de evento de `click` no elemento, se houve alguma. Igualmente para outros outros eventos e manipuladores de evento associados, como `blur` (`onblur`), `keypress` (`onkeypress`), e assim por diante.

Porque era essencialmente uma parte do DOM 0, esse método era largamente suportado e não necessitava de códigos entre-navegadores especiais; logo é normalmente usado para registrar esperas de evento dinâmicamente, a menos que atributos extras do `addEventListener()` sejam necessários.

Problemas de memória

```
var i;
var els = document.getElementsByTagName('*');

// Caso 1
for(i=0 ; i<els.length ; i++){
  els[i].addEventListener("click", function(e){/*fazer algo*/}, false});
```

```
}

// Caso 2
function processarEvento(e){
  /*fazer algo*/
}

for(i=0 ; i<els.length ; i++){
  els[i].addEventListener("click", processarEvento, false});
}
```

No primeiro caso, uma nova função (anônima) é criada em cada turno do loop. No segundo caso, a mesma função previamente declarada é usada como um manipulador de evento. Isso resulta em um consumo menor de memória. Além do mais, no primeiro caso, já que nenhuma referência à função anônima é mantida, não é possível chamar `element.removeEventListener (en-US)` porque não há uma referência ao manipulador, enquanto no segundo caso é possível fazer `myElement.removeEventListener("click", processEvent, false)`.

Compatiblidade de navegadores

> **Estamos convertendo nossos dados de compatibilidade para o formato JSON**. Esta tabela de compatibilidade ainda usa o formato antigo, pois ainda não convertemos os dados que ela contém. **Descubra como você pode ajudar! (en-US)**

- Desktop
- Dispositivo móvel

Característica	Chrome	Firefox (Gecko)	Internet Explorer	Opera	Safari (WebKit)
Suporte básico	1.0	1.0 (1.7 or earlier)	9.0	7	1.0
`useCapture` é opcional	1.0	6.0	9.0	11.60	(Yes)

Característica	Android	Firefox Mobile (Gecko)	IE Mobile	Opera Mobile	Safari Mobile
Suporte básico	1.0	1.0 (1.0)	9.0	6.0	1.0

Notas Gecko

- Antes do Firefox 6, o navegador daria um erro se o parâmetro `useCapture` não fosse especificado `false`. Antes do Gecko 9.0 (Firefox 9.0 / Thunderbird 9.0 / SeaMonkey 2.6),

`addEventListener()` daria um erro se o parâmetro `escuta` fosse `null`; agora o método retorna sem erros, mas também sem fazer nada.

Notas Webkit

- Apesar do WebKit ter explicitamente adicionado `[optional]` ao parâmetro `useCapture` como recentemente anunciado em Junho de 2011, já funcionava antes do anúncio da mudança. Ela foi anunciada no Safari 5.1 e no Chrome 13.

HTML Content

EventTarget.removeEventListener()

Remove o event listener anteriormente registrado com `EventTarget.addEventListener()`.

Sintaxe

```
target.removeEventListener(type, listener[, useCapture])
```

`type`
Uma string indicando o tipo de evento a ser removido.

`listener`
A função `EventListener` a ser removida do event target.

`useCapture` **Optional**
Indica quando o `EventListener` a ser removido foi registrado ou não como *capturing listener*. Caso este parâmetro seja omitido, o valor *false* será assumido por padrão.

Se um listener foi registrado duas vezes, uma com o parâmetro *capture* especificado e outra sem, cada um deve ser removido separadamente. A remoção de um *capturing listener* não afeta a versão *non-capturing* do mesmo listener, e vice versa.

> **Nota:** `useCapture` era obrigatório em versões mais antigas dos navegadores. Para ampla compatibilidade, sempre informe o parâmetro `useCapture.`

Notas

Se um `EventListener` é removido de um `EventTarget` enquanto *este* está processando um evento, esse não será disparado pelas *current actions*. Um `EventListener` não será invocado para o evento o qual foi registrado depois de ter sido removido, porém pode ser registrado novamente.

Chamar `removeEventListener()` com argumentos que não identifiquem nenhum `EventListener` registrado no `EventTarget` não tem qualquer efeito.

Exemplo

Este é um exemplo de como associar e remover um event listener.

```
var div = document.getElementById('div');
var listener = function (event) {
  /* faça alguma coisa... */
};
div.addEventListener('click', listener, false);
div.removeEventListener('click', listener, false);
```

Compatibilidade com navegadores

EventTarget.removeEventListener()

HTML Content

EventTarget.removeEventListener()

Remove o event listener anteriormente registrado com `EventTarget.addEventListener()`.

Sintaxe

```
target.removeEventListener(type, listener[, useCapture])
```

`type`
Uma string indicando o tipo de evento a ser removido.

`listener`
A função `EventListener` a ser removida do event target.

`useCapture` **Optional**
Indica quando o `EventListener` a ser removido foi registrado ou não como *capturing listener*. Caso este parâmetro seja omitido, o valor *false* será assumido por padrão.

Se um listener foi registrado duas vezes, uma com o parâmetro *capture* especificado e outra sem, cada um deve ser removido separadamente. A remoção de um *capturing listener* não afeta a versão *non-capturing* do mesmo listener, e vice versa.

Nota: `useCapture` era obrigatório em versões mais antigas dos navegadores. Para ampla compatibilidade, sempre informe o parâmetro `useCapture.`

Notas

Se um `EventListener` é removido de um `EventTarget` enquanto *este* está processando um evento, esse não será disparado pelas *current actions*. Um `EventListener` não será invocado para o evento o

qual foi registrado depois de ter sido removido, porém pode ser registrado novamente.

Chamar `removeEventListener()` com argumentos que não identifiquem nenhum `EventListener` registrado no `EventTarget` não tem qualquer efeito.

Exemplo

Este é um exemplo de como associar e remover um event listener.

```
var div = document.getElementById('div');
var listener = function (event) {
  /* faça alguma coisa... */
};
div.addEventListener('click', listener, false);
div.removeEventListener('click', listener, false);
```

Compatibilidade com navegadores

EventTarget.dispatchEvent()

Dispara um Event para o EventTarget especificado, invocando os EventListeners especificados, em uma ordem apropriada. O processamento normal das regras (including the capturing and optional bubbling phase) aplica-se a eventos disparados manualmente com dispatchEvent().

Sintaxe

```
cancelled = !target.dispatchEvent(event)
```

- event é o objeto Event a ser disparado.
- target é utilizado para inicializar o Event.target e determinar quais event listeners serão invocados.
- O valor retornado é false se ao menos um dos event handlers o qual manipulou o evento chamou Event.preventDefault(). De outro modo, isto retorna true.

O método dispatchEvent joga UNSPECIFIED_EVENT_TYPE_ERR se o tipo do evento não foi especificado pela inicialização do evento antes do método ser chamado, ou se o tipo do evento for is null ou uma string vazia. Exceções jogadas por event handlers são reportadas como exceções não-capturáveis; os event handlers são executados em uma callstack aninhada; eles bloqueiam o chamador até que a rotina tenha sido totalmente executada, mas as execeções não se propagam para o chamador.

Notas

dispatchEvent é a última fase do processo create-init-dispatch, a qual é usada para disparar eventos na implementação do event model. O evento pode ser criado utilizando o método document.createEvent e pode ser inicializado com initEvent ou outro método de inicialização mais específico, como initMouseEvent ou initUIEvent.

Veja também a referência Event object.

6.0 - JAVASCRIPT ASSÍNCRONO

JavaScript síncrono

HTML Content

Apresentando JavaScript assíncrono

Anterior

Visão geral: assíncrono

Próximo

Neste artigo, recapitulamos brevemente os problemas associados ao JavaScript síncrono e observamos algumas das diferentes técnicas assíncronas que você encontrará, mostrando como elas podem nos ajudar a resolver esses problemas.

Pré-requisitos:	Conhecimento básico de informática, uma compreensão razoável dos fundamentos do JavaScript.
Objetivo:	Para ganhar familiaridade com o que é JavaScript assíncrono, como ele difere do JavaScript síncrono e quais casos de uso ele tem.

JavaScript síncrono

Para nos permitir entender o que é JavaScript **assíncrono**, devemos começar certificando-nos de que entendemos o que é JavaScript **síncrono**. Esta seção recapitula algumas das informações que vimos no artigo anterior.

Muitas das funcionalidades que vimos nos módulos da área de aprendizagem anteriores são síncronas - você executa algum código e o resultado é retornado assim que o navegador puder fazer isso. Vejamos um exemplo simples (veja ao vivo aqui, e veja a fonte):

```
const btn = document.querySelector('button');
btn.addEventListener('click', () => {
  alert('You clicked me!');

  let pElem = document.createElement('p');
  pElem.textContent = 'This is a newly-added paragraph.';
  document.body.appendChild(pElem);
});
```

Neste bloco, as linhas são executadas uma após a outra:

1. Pegamos uma referência a um `<button>` elemento que já está disponível no DOM.
2. Adicionamos um `click` ouvinte de evento para que, quando o botão for clicado:
    1. Uma `alert()` mensagem aparece.

2. Depois que o alerta é descartado, criamos um `<p>` elemento.
3. Em seguida, fornecemos algum conteúdo de texto.
4. Finalmente, acrescentamos o parágrafo ao corpo do documento.

Enquanto cada operação está sendo processada, nada mais pode acontecer - a renderização é pausada. Isso ocorre porque, como dissemos no **artigo anterior**, o JavaScript é de encadeamento único. Apenas uma coisa pode acontecer por vez, em um único encadeamento principal, e todo o resto é bloqueado até que uma operação seja concluída.

JavaScript Assíncrono

HTML Content

JavaScript assíncrono

Por razões ilustradas anteriormente (por exemplo, relacionadas ao bloqueio), muitos recursos da API da Web agora usam código assíncrono para executar, especialmente aqueles que acessam ou buscam algum tipo de recurso de um dispositivo externo, como buscar um arquivo da rede, acessar um banco de dados e retornar dados dele, acessar um fluxo de vídeo de uma webcam ou transmitir a tela para um fone de ouvido VR.

Por que é difícil começar a trabalhar usando código assíncrono? Vejamos um exemplo rápido. Quando você busca uma imagem de um servidor, não pode retornar o resultado imediatamente. Isso significa que o seguinte (pseudocódigo) não funcionaria:

```
let response = fetch('myImage.png'); // fetch is asynchronous
let blob = response.blob();
// display your image blob in the UI somehow
```

Isso porque você não sabe quanto tempo levará para fazer o download da imagem, então quando você executar a segunda linha, ela gerará um erro (possivelmente de forma intermitente, possivelmente todas as vezes) porque o `response` ainda não está disponível. Em vez disso, você precisa que seu código espere até que `response` seja retornado antes de tentar fazer qualquer outra coisa com ele.

Existem dois tipos principais de estilo de código assíncrono que você encontrará no código JavaScript: callbacks de estilo antigo e código de estilo de promessa mais recente. Nas seções a seguir, revisaremos cada um deles separadamente.

HTML Content

Callbacks assíncronos

Retornos de chamada assíncronos são funções especificadas como argumentos ao chamar uma função que iniciará a execução do código em segundo plano. Quando o código de segundo plano termina de ser executado, ele chama a função de retorno de chamada para informar que o trabalho foi concluído ou que algo de interesse aconteceu. O uso de retornos de chamada é um pouco antiquado agora, mas você ainda os verá em uso em uma série de APIs mais antigas, mas ainda comumente usadas.

Um exemplo de callback assíncrono é o segundo parâmetro do `addEventListener()` método (como vimos na ação acima):

```
btn.addEventListener('click', () => {
  alert('You clicked me!');

  let pElem = document.createElement('p');
  pElem.textContent = 'This is a newly-added paragraph.';
  document.body.appendChild(pElem);
});
```

O primeiro parâmetro é o tipo de evento a ser ouvido e o segundo parâmetro é uma função de retorno de chamada que é chamada quando o evento é disparado.

Quando passamos uma função de retorno de chamada como um argumento para outra função, estamos apenas passando a referência da função como um argumento, ou seja, a função de retorno de chamada **não**é executada imediatamente. É “chamado de volta” (daí o nome) de forma assíncrona em algum lugar dentro do corpo da função que o contém. A função de contenção é responsável por executar a função de retorno de chamada quando chegar a hora.

Você pode escrever sua própria função contendo um retorno de chamada com bastante facilidade. Vejamos outro exemplo que carrega um recurso por meio da `XMLHttpRequest` API(execute-o ao vivo, e veja a fonte):

```
function loadAsset(url, type, callback) {
  let xhr = new XMLHttpRequest();
  xhr.open('GET', url);
  xhr.responseType = type;

  xhr.onload = function() {
    callback(xhr.response);
  };

  xhr.send();
}

function displayImage(blob) {
  let objectURL = URL.createObjectURL(blob);

  let image = document.createElement('img');
  image.src = objectURL;
  document.body.appendChild(image);
}

loadAsset('coffee.jpg', 'blob', displayImage);
```

Aqui, criamos uma `displayImage()` função que representa um blob transmitido a ele como um URL de objeto e, em seguida, criamos uma imagem para exibir o URL, anexando-o ao do documento `<body>`. No entanto, criamos uma `loadAsset()` função que recebe um retorno de chamada como parâmetro, junto com uma URL a ser buscada e um tipo de conteúdo. Ele usa `XMLHttpRequest` (geralmente abreviado como "XHR") para buscar o recurso no URL fornecido e, em seguida, passa a resposta ao retorno de chamada para fazer algo com ele. Nesse caso, o retorno de chamada está aguardando a chamada XHR para concluir o download do recurso (usando o `onload` manipulador de eventos) antes de passá-lo para o retorno de chamada.

Os retornos de chamada são versáteis - não apenas permitem que você controle a ordem em que as funções são executadas e quais dados são transmitidos entre elas, mas também permitem que você transmita dados para funções diferentes, dependendo das circunstâncias. Então, você poderia ter ações diferentes para executar sobre a resposta baixado, como `processJSON()`, `displayText()`, etc.

Observe que nem todos os retornos de chamada são assíncronos - alguns são executados de forma síncrona. Um exemplo é quando usamos `Array.prototype.forEach()` para percorrer os itens em uma matriz (ver ao vivo, e a fonte):

```
const gods = ['Apollo', 'Artemis', 'Ares', 'Zeus'];

gods.forEach(function (eachName, index){
  console.log(index + '. ' + eachName);
});
```

Neste exemplo, percorremos um array de deuses gregos e imprimimos os números de índice e valores no console. O parâmetro esperado de `forEach()` é uma função de retorno de chamada, que por si só leva dois parâmetros, uma referência ao nome da matriz e valores de índice. No entanto, ele não espera por nada - ele é executado imediatamente.

HTML Content

Promessas

As promessas são o novo estilo de código assíncrono que você verá usado nas APIs da Web modernas. Um bom exemplo é a `fetch()` API, que é basicamente uma versão moderna e mais eficiente do `XMLHttpRequest`. Vejamos um exemplo rápido, de nosso artigo Buscando dados do servidor:

```
fetch('products.json').then(function(response) {
  return response.json();
}).then(function(json) {
  let products = json;
  initialize(products);
}).catch(function(err) {
  console.log('Fetch problem: ' + err.message);
});
```

> **Nota:** Você pode encontrar a versão final no GitHub (veja a fonte aqui, e também ver rodando ao vivo)

Aqui, vemos `fetch()` um único parâmetro - a URL de um recurso que você deseja buscar na rede - e retornando uma promessa. A promessa é um objeto que representa a conclusão ou falha da operação assíncrona. Ele representa um estado intermediário, por assim dizer. Em essência, é a maneira do navegador dizer "Prometo retornar a você a resposta assim que puder", daí o nome "promessa".

Esse conceito pode exigir prática para se acostumar; parece um pouco comgato de Schrodingerem ação. Nenhum dos resultados possíveis aconteceu ainda, portanto, a operação de busca está aguardando o

resultado da tentativa do navegador de concluir a operação em algum momento no futuro. Em seguida, temos mais três blocos de código encadeados no final do `fetch()`:

três blocos de código encadeados no final do fetch():

- Dois then() blocos. Ambos contêm uma **função de retorno de chamada que será executada se a operação anterior for bem-sucedida**, e cada retorno de chamada recebe como entrada o resultado da operação anterior bem-sucedida, para que você possa seguir em frente e fazer outra coisa. **Cada .then()bloco retorna outra promessa, o que significa que você pode encadear vários .then()blocos entre si, de modo que várias operações assíncronas possam ser executadas em ordem,** uma após a outra.

- O catch()bloco no final é executado se algum dos .then()blocos falhar - de forma semelhante aos try...catchblocos síncronos , **um objeto de erro é disponibilizado dentro do catch(), que pode ser usado para relatar o tipo de erro ocorrido.** Observe, entretanto, que síncrono try...catchnão funcionará com promessas, embora funcione com async / await , como você aprenderá mais tarde.

Nota: Você aprenderá muito mais sobre promessas posteriormente neste módulo, então não se preocupe se ainda não as entender completamente.

A fila de eventos

As operações assíncronas, como promessas, são colocadas em uma **fila de eventos** , que é executada depois que o thread principal termina o processamento, para que *não bloqueiem a* execução do código JavaScript subsequente. As operações enfileiradas serão concluídas o mais rápido possível e, em seguida, retornarão seus resultados ao ambiente JavaScript.

Promessas versus chamadas de retorno

As promessas têm algumas semelhanças com os retornos de chamada do estilo antigo. Eles são essencialmente um objeto retornado ao qual você anexa funções de retorno de chamada, em vez de passar os retornos de chamada para uma função.

No entanto **as promessas são feitas especificamente para lidar com operações assíncronas e têm muitas vantagens em relação aos retornos de chamada do estilo antigo**:

- Você pode **encadear várias operações assíncronas usando várias .then()operações, passando o resultado de uma para a próxima como uma entrada.** Isso é muito mais difícil de fazer com retornos de chamada, que muitas vezes termina com uma "pirâmide da desgraça" confusa (também conhecida como inferno de retorno)
- Os retornos de chamada promissores são sempre chamados na ordem estrita em que são colocados na fila de eventos.
- O tratamento de erros é muito melhor - **todos os erros são tratados por um único .catch()bloco no final do bloco, em vez de serem tratados individualmente em cada nível da "pirâmide".**
- As promessas evitam a inversão de controle, ao contrário dos retornos de chamada do estilo antigo, que perdem o controle total de como a função será executada ao passar um retorno de chamada para uma biblioteca de terceiros.

-

.then() : encadear várias operações assíncronas usando várias .then() operações, passando o resultado de uma para próxima como uma entrada.

- **.catch()** : todos os erros são tratados por um único .cath() bloco no final do bloco, em vez de serem tratados individualmente em cada nível da pirâmide.

HTML Content

A natureza do código assíncrono

Vamos explorar um exemplo que ilustra melhor a natureza do código assíncrono, mostrando o que pode acontecer quando não estamos totalmente cientes da ordem de execução do código e dos problemas de tentar tratar o código assíncrono como o código síncrono. O exemplo a seguir é bastante semelhante ao que vimos antes (ver ao vivo, e a fonte) Uma diferença é que incluímos várias `console.log()` instruções para ilustrar uma ordem na qual você pode pensar que o código seria executado.

```
console.log ('Starting');
let image;

fetch('coffee.jpg').then((response) => {
  console.log('It worked :)')
  return response.blob();
}).then((myBlob) => {
  let objectURL = URL.createObjectURL(myBlob);
  image = document.createElement('img');
  image.src = objectURL;
  document.body.appendChild(image);
}).catch((error) => {
  console.log('There has been a problem with your fetch o
});

console.log ('All done!');
```

O navegador começará a executar o código, verá a primeira `console.log()` instrução (`Starting`) e executará, e então criará a `image` variável.

Em seguida, ele irá para a próxima linha e começará a executar o `fetch()` bloco, mas, como é `fetch()` executado de forma assíncrona sem bloqueio, a execução do código continua após o código relacionado à promessa, alcançando assim a `console.log()` instrução final (`All done!`) e enviando-a para o console.

Somente quando o `fetch()` bloco terminar de ser executado e entregar seu resultado através dos `.then()` blocos, finalmente veremos a segunda `console.log()` mensagem (`It worked :)`) aparecer. Portanto, as mensagens apareceram em uma ordem diferente do que você esperava:

- Iniciando
- Tudo feito!
- Funcionou :)

Se isso o confunde, considere o seguinte exemplo menor:

```
console.log("registering click handler");

button.addEventListener('click', () => {
  console.log("get click");
});

console.log("all done");
```

Isso é muito semelhante em comportamento - a primeira e a terceira `console.log()` mensagens serão mostradas imediatamente, mas a segunda será bloqueada até que alguém clique no botão do mouse. O exemplo anterior funciona da mesma maneira, exceto que, nesse caso, a segunda mensagem é bloqueada na cadeia de promessa buscando um recurso e exibindo-o na tela, em vez de um clique.

Em um exemplo de código menos trivial, esse tipo de configuração pode causar um problema - você não pode incluir um bloco de código assíncrono que retorna um resultado, no qual você confia posteriormente em um bloco de código de sincronização. Você simplesmente não pode garantir que a função assíncrona retornará antes que o navegador tenha processado o bloco de sincronização.

Para ver isso em ação, tente fazer uma cópia local de nosso exemploe alterando a quarta `console.log()` chamada para o seguinte:

```
console.log ('All done! ' + image.src + 'displayed.');
```

Agora você deve obter um erro em seu console, em vez da terceira mensagem:

```
TypeError: image is undefined; can't access its "src"
property
```

Isso ocorre porque, no momento em que o navegador tenta executar a terceira `console.log()` instrução, o `fetch()` bloco não terminou de ser executado, portanto, a `image` variável não recebeu um valor.

> **Nota:**Por razões de segurança, você não pode `fetch()` arquivos de seu sistema de arquivos local (ou executar outras operações localmente); para executar o exemplo acima localmente, você terá que executar o exemplo por meio de um servidor da web local.

Aprendizagem ativa: torne tudo assíncrono!

Para corrigir o `fetch()` exemplo problemático e fazer com que as três `console.log()` instruções apareçam na ordem desejada, você também pode fazer com que a terceira `console.log()` instrução seja executada de forma assíncrona. Isso pode ser feito movendo-o para dentro de outro `.then()` bloco encadeado no final do segundo, ou movendo-o para dentro do segundo `then()` bloco. Tente consertar isso agora.

> **Nota:**se você ficar preso, você podeencontre uma resposta aqui(Veja correndo ao vivotb). Você também pode encontrar muito mais informações sobre promessas em nosso guia de programação assíncrona Graceful com promessas, mais adiante no módulo.

Conclusão

Em sua forma mais básica, o JavaScript é uma linguagem síncrona, bloqueadora e de encadeamento único, na qual apenas uma operação pode estar em andamento por vez. Mas os navegadores da web definem funções e APIs que nos permitem registrar funções que não devem ser executadas de forma síncrona e, em vez disso, devem ser invocadas de forma assíncrona quando ocorre algum tipo de evento (a passagem do tempo, a interação do usuário com o mouse ou a chegada de dados pela rede, por exemplo). Isso significa que você pode permitir que seu código faça várias coisas ao mesmo tempo sem interromper ou bloquear seu thread principal.

Se queremos executar o código de forma síncrona ou assíncrona, dependerá do que estamos tentando fazer.

Há momentos em que queremos que as coisas carreguem e aconteçam imediatamente. Por exemplo, ao aplicar alguns estilos definidos pelo usuário a uma página da web, você deseja que os estilos sejam aplicados o mais rápido possível.

Se estivermos executando uma operação que leva tempo, no entanto, como consultar um banco de dados e usar os resultados para preencher os modelos, é melhor empurrar isso para fora da pilha e concluir a tarefa de forma assíncrona. Com o tempo, você aprenderá quando faz mais sentido escolher uma técnica assíncrona em vez de uma síncrona.

7.0 - PROGRAMAÇÃO ASSÍNCRONA

Conceitos gerais de programação assíncrona

https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Concepts#asynchronous

Não faz sentido ficar sentado esperando por algo quando você pode deixar a outra tarefa executar em outro núcleo do processador e avisar quando estiver concluída. Isso permite que você **faça outro trabalho nesse ínterim**, que é a base da **programação assíncrona**.
Conforme você usa APIs mais novas e poderosas, você encontrará mais casos em que a única maneira de fazer as coisas é de forma assíncrona.

- Visão geral: assíncrono
- Próximo

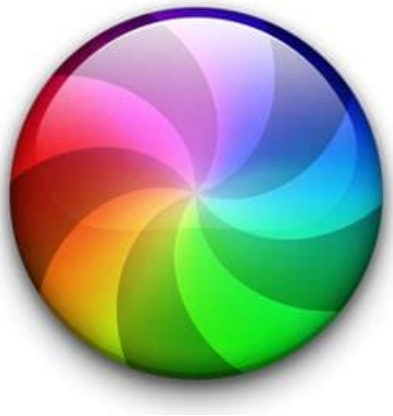
Neste artigo, examinaremos vários conceitos importantes relacionados à programação assíncrona e como isso se parece em navegadores da web e JavaScript. Você deve compreender esses conceitos antes de trabalhar com os outros artigos do módulo.

Pré-requisitos:	Conhecimento básico de informática, uma compreensão razoável dos fundamentos do JavaScript.
Objetivo:	Para entender os conceitos básicos por trás da programação assíncrona e como eles se manifestam em navegadores da web e JavaScript.

Assíncrono?

Normalmente, o código de um determinado programa é executado diretamente, com apenas uma coisa acontecendo ao mesmo tempo. Se uma função depende do resultado de outra função, ela deve esperar que a outra função termine e retorne e, até que isso aconteça, todo o programa é essencialmente interrompido da perspectiva do usuário.

Os usuários de Mac, por exemplo, às vezes experimentam isso como um cursor giratório com a cor do arco-íris (ou "bola de praia", como costuma ser chamada). Este cursor é como o sistema operacional diz "o programa atual que você está usando teve que parar e esperar algo terminar, e está demorando tanto que fiquei preocupado que você se perguntasse o que estava acontecendo".



Esta é uma experiência frustrante e não é um bom uso do poder de processamento do computador - especialmente em uma época em que os computadores têm vários núcleos de processador disponíveis. Não faz sentido ficar sentado esperando por algo quando você pode deixar a outra tarefa executar em outro núcleo do processador e avisar quando estiver concluída. Isso permite que você faça outro trabalho nesse ínterim, que é a base da **programação assíncrona**. Depende do ambiente de programação que você está usando (navegadores da web, no caso de desenvolvimento da web) fornecer APIs que permitem a execução dessas tarefas de forma assíncrona.

Código de bloqueio

As técnicas assíncronas são muito úteis, principalmente na programação web. Quando um aplicativo da web é executado em um navegador e executa uma grande quantidade de código sem retornar o controle ao navegador, o navegador pode parecer estar congelado. Isso é chamado de **bloqueio**; o navegador é impedido de continuar a lidar com a entrada do usuário e realizar outras tarefas até que o aplicativo da web retorne o controle do processador.

Vejamos alguns exemplos que mostram o que queremos dizer com bloqueio.

Na nossa simple-sync.htmlexemplo (ver rodando ao vivo), adicionamos um ouvinte de evento de clique a um botão para que, quando clicado, ele execute uma operação demorada (calcula 10 milhões de datas e registra a última no console) e, em seguida, adiciona um parágrafo ao DOM:

```
const btn = document.querySelector('button');
btn.addEventListener('click', () => {
  let myDate;
  for(let i = 0; i < 10000000; i++) {
    let date = new Date();
    myDate = date;
  }


  console.log(myDate);


  let pElem = document.createElement('p');
  pElem.textContent = 'This is a newly-added paragraph.';
  document.body.appendChild(pElem);
});
```

Copiar para área de transferência

Ao executar o exemplo, abra o console JavaScript e clique no botão - você notará que o parágrafo não aparece até que as datas tenham sido calculadas e a mensagem do console tenha sido registrada. O código é executado na ordem em que aparece no código-fonte e a operação posterior não é executada até que a operação anterior seja concluída.

Nota:o exemplo anterior é muito irreal. Você nunca calcularia 10 milhões de datas em um aplicativo da web real! No entanto, serve para lhe dar uma ideia básica.

Em nosso segundo exemplo, simple-sync-ui-blocking.html(ver ao vivo), simulamos algo um pouco mais realista que você pode encontrar em uma página real. Bloqueamos a interatividade do usuário com a renderização da IU. Neste exemplo, temos dois botões:

- Um botão "Preencher tela" que, quando clicado, preenche o disponível <canvas>com 1 milhão de círculos azuis.
- Um botão "Clique em mim para obter um alerta" que, quando clicado, exibe uma mensagem de alerta.

```
function expensiveOperation() {
  for(let i = 0; i < 1000000; i++) {
    ctx.fillStyle = 'rgba(0,0,255, 0.2)';
    ctx.beginPath();
    ctx.arc(random(0, canvas.width), random(0, canvas.height), 10, degToRad(0), degToRad(360), false);
    ctx.fill();
  }
}


fillBtn.addEventListener('click', expensiveOperation);


alertBtn.addEventListener('click', () =>
  alert('You clicked me!');
);
```

Copiar para área de transferência
Se você clicar no primeiro botão e, em seguida, clicar rapidamente no segundo, verá que o alerta não aparecerá até que os círculos tenham terminado de ser renderizados. A primeira operação bloqueia a segunda até que ela termine a execução.

Nota:OK, no nosso caso, é feio e estamos fingindo o efeito de bloqueio, mas esse é um problema comum que desenvolvedores de aplicativos reais lutam para mitigar o tempo todo.

Por que é isso? A resposta é porque JavaScript, em geral, é **de thread único**. Neste ponto, precisamos apresentar o conceito de **threads**.

Tópicos

Um **thread**é basicamente um processo único que um programa pode usar para concluir tarefas. Cada thread só pode fazer uma única tarefa por vez:

```
Task A --> Task B --> Task C
```

Cada tarefa será executada sequencialmente; uma tarefa deve ser concluída antes que a próxima possa ser iniciada.

Como dissemos antes, muitos computadores agora têm vários núcleos, portanto, podem fazer várias coisas ao mesmo tempo. Linguagens de programação que podem suportar vários threads podem usar vários núcleos para completar várias tarefas simultaneamente:

```
Thread 1: Task A --> Task B
Thread 2: Task C --> Task D
```

JavaScript é de thread único

JavaScript é tradicionalmente de thread único. Mesmo com vários núcleos, você só poderia fazer com que ele executasse tarefas em um único thread, chamado de **thread principal**. Nosso exemplo acima é executado assim:

```
Main thread: Render circles to canvas --> Display alert()
```

Depois de algum tempo, o JavaScript ganhou algumas ferramentas para ajudar com esses problemas. Os Web workerspermitem que você envie parte do processamento do JavaScript para um thread separado, chamado de worker,

para que você possa executar vários blocos de JavaScript simultaneamente. Geralmente, você usaria um trabalhador para executar processos caros fora do thread principal para que a interação do usuário não fosse bloqueada.

```
  Main thread: Task A --> Task C
Worker thread: Expensive task B
```

Com isso em mente, dê uma olhada em simple-sync-worker.html(ver rodando ao vivo), novamente com o console JavaScript do seu navegador aberto. Esta é uma reescrita do nosso exemplo anterior que calcula os 10 milhões de datas, mas desta vez estamos usando um trabalhador para o cálculo. Você pode ver o código do trabalhador aqui:worker.js. Agora, quando você clica no botão, o navegador é capaz de exibir o parágrafo antes que as datas tenham terminado de calcular. Depois que o trabalhador termina de calcular, ele registra a data final no console. A primeira operação não bloqueia mais a segunda.

Código assíncrono

Os Web workers são muito úteis, mas têm suas limitações. A principal delas é que eles não podem acessar o DOM- você não pode fazer com que um trabalhador faça nada diretamente para atualizar a IU. Não poderíamos renderizar nosso 1 milhão de círculos azuis dentro de nosso trabalhador; ele pode basicamente fazer cálculos numéricos.

O segundo problema é que, embora o código executado em um trabalhador não esteja bloqueando, ele ainda é basicamente síncrono. Isso se torna um problema quando uma função depende dos resultados de vários processos anteriores para funcionar. Considere os seguintes diagramas de thread:

```
Main thread: Task A --> Task B
```

Nesse caso, digamos que a Tarefa A está fazendo algo como buscar uma imagem do servidor e a Tarefa B, em seguida, faz algo com a imagem, como aplicar um filtro a ela. Se você começar a executar a Tarefa A e tentar imediatamente executar a Tarefa B, receberá um erro, porque a imagem ainda não estará disponível.

```
  Main thread: Task A --> Task B --> |Task D|
Worker thread: Task C -----------> |      |
```

Nesse caso, digamos que a Tarefa D use os resultados da Tarefa B e da Tarefa C. Se pudermos garantir que esses resultados estarão disponíveis ao mesmo tempo, então podemos estar OK, mas isso é improvável. Se a Tarefa D tentar ser executada quando uma de suas entradas ainda não estiver disponível, ocorrerá um erro.

Para corrigir esses problemas, os navegadores nos permitem executar certas operações de forma assíncrona. Recursos como Promisespermitem que você defina uma operação em execução (por exemplo, a busca de uma imagem do servidor) e, em seguida, aguarde até que o resultado seja retornado antes de executar outra operação:

```
Main thread: Task A                  Task B
    Promise:      |__async operation__|
```

Como a operação está acontecendo em outro lugar, o thread principal não é bloqueado enquanto a operação assíncrona está sendo processada.

Começaremos a ver como podemos escrever código assíncrono no próximo artigo. Coisas emocionantes, hein? Continue lendo!

Conclusão

O design de software moderno gira cada vez mais em torno do uso de programação assíncrona, para permitir que os programas façam mais de uma coisa ao mesmo tempo. **Conforme você usa APIs mais novas e poderosas, você**

encontrará mais casos em que a única maneira de fazer as coisas é de forma assíncrona. Costumava ser difícil escrever código assíncrono. Ainda é preciso se acostumar, mas ficou muito mais fácil. No restante deste módulo, exploraremos mais por que o código assíncrono é importante e como projetar um código que evite alguns dos problemas descritos acima.

8.0 - ESTRUTURA DE DADOS JAVASCRIPT

HTML Content

Estrutura de dados do Javascript

Todas as linguagens de programação têm sua própria estrutura de dados embutida, mas essa estrutura frequentemente difere uma da outra. Este artigo busca listar os tipos de dados disponíveis na linguagem JavaScript e que propriedades eles possuem. Quando possível, comparações com outras linguagens serão apresentadas.

Tipagem Dinâmica

JavaScript é uma linguagem de tipagem *dinâmica*. Isso significa que você não necessita declarar o tipo de uma variável antes de sua atribuição. O tipo será automaticamente determinado quando o programa for processado. Isso também significa que você pode reatribuir uma mesma variável com um tipo diferente:

```
var foo = 42;    // foo é um Number agora
foo     = "bar"; // foo é um String agora
foo     = true;  // foo é um Boolean agora
```

Tipos de Dados

A última versão ECMAScript define sete tipos de dados:

- Sete tipos de dados são: primitives:
  - Boolean
  - Null
  - Undefined
  - Number
  - BigInt
  - String
  - Symbol
- e Object

Valores Primitivos

Todos os tipos, com a exceção de objetos, definem valores imutáveis (valores que são incapazes de mudar). Por exemplo e diferentemente da linguagem C, Strings são imutáveis. Nós nos referimos a valores desse tipo como "valores primitivos".

Tipo "Boolean"

Boolean representa uma entidade lógica e pode ter dois valores: verdadeiro `(true)` ou falso `(false)`.

Tipo "Null"

O tipo Null tem exatamente um valor: `null` (nulo). Veja `null` e Nullpara mais detalhes.

Tipo "Undefined"

Uma variável que não foi atribuída a um valor específico, assume o valor *`undefined`* (indefinido). Veja `undefined` e Undefinedpara mais detalhes.

Tipo "Number"

De acordo com os padrões ECMAScript, existe somente um tipo numérico. O double-precision 64-bit binary format IEEE 754 value(número entre $-(2^{53}-1)$ e $2^{53}-1$). **Não existe um tipo específico para inteiros**. Além de poderem representar números de ponto-flutuante, o tipo *number* possui três valores simbólicos: `+Infinity`, `-Infinity`, e `NaN` (não numérico).

Para verificar o maior ou o menor valor disponível dentro de +/- `Infinity`, você pode usar as constantes `Number.MAX_VALUE` ou `Number.MIN_VALUE`, e a partir do ECMAScript 6, você também consegue verificar se um número está dentro da região de um ponto flutuante do tipo `double-precision,` usando `Number.isSafeInteger()`, como também `Number.MAX_SAFE_INTEGER`, e `Number.MIN_SAFE_INTEGER`. Fora dessa região, números inteiros em JavaScript não são mais precisos e serão uma aproximação de um número de ponto flutuante do tipo `double-precision`.

O tipo *number*possui apenas um inteiro que tem duas representações: 0 é representado como -0 ou +0. ("0" é um pseudônimo para +0). Na prática, isso não gera grandes impactos. Por exemplo `+0 === -0` resulta em `true`. Entretanto, você pode notar a diferença quando realiza uma divisão por 0:

```
> 42 / +0
Infinity
> 42 / -0
-Infinity
```

Apesar de um número frequentemente representar somente seu valor, JavaScript fornece alguns operadores binários. Estes podem ser usados para representar muitos valores booleanos dentro de um único número usando *bit masking*. Entretanto, isto é usualmente considerado uma má prática, desde que JavaScript oferece outros meios para representar uma configuração de booleanos (como uma array de booleanos ou um objeto com valores booleanos assinalados em propriedades). Bit masking também tende a fazer o código mais difícil de ler, entender e de realizar manutenção. Isto pode ser necessário em um ambiente bastante limitado, como quando tentamos lidar com limitação de armazenamento ou armazenamento local ou em casos extremos quando cada bit na rede conta. Esta técnica somente deveria ser considerada quando é a última medida possível para otimizar o tamanho.

Tipo "String"

O tipo `String` em JavaScript é usado para representar dados textuais. Isto é um conjunto de "elementos" de valores de 16-bits *unsigned integer*. Cada elemento na *string*ocupa uma posição na *string*. O primeiro elemento está no índice 0, o próximo no índice 1, e assim por diante. O comprimento de uma *string*é o número de elementos nela.

Diferente de linguagens como C, strings em JavaScript são imutáveis. Isto significa que: uma vez criada a string, não é possível modificá-la. Entretanto, ainda é possível criar outra string baseada em um operador na string original. Por exemplo:

- Uma substring da original a partir de letras individuais ou usando `String.substr()`.
- Uma concatenação de duas strings usando o operador (`+`) ou `String.concat()`.

Cuidado com "stringly-typing" (digitação) no seu código!

Pode ser tentador utilizar strings para representar dados complexos. Fazer isso traz os seguintes benefícios de curto prazo:

- É facil construir strings complexas com concatenação.
- Strings são fáceis para debug (o que você vê escrito é o que está na string).
- Strings são o denominador comum entre várias APIs (input fields, local storagevalues, `XMLHttpRequest` responses ao usar `responseText`, etc.) e pode ser tentador trabalhar apenas com strings.

Com convenções, é possível representar qualquer estrutura de dados com uma string. Ainda assim, isso não faz a prática ser uma boa ideia. Por exemplo, pode-se emular uma lista utilizando separadores (um JavaScript array seria mais adequado). Infelizmente, se o separador faz parte de um dos elementos da "lista", então, a lista é quebrada. Um caracter de escape pode ser utilizado,etc. Tudo isso requer convenções e cria uma carga de manuntenção desnecessária.

É aconselhável usar strings para dados textuais. Quando representar dados complexos, analise as strings e utilize abstrações apropriadas.

Symbol type

Symbols são novos no JavaScript ECMAScript edição 6. Um Symbol é um valor primitivo **único** e **imutável** e pode ser usado como chave de uma propriedade de Object (ver abaixo). em algumas linguagens de programação, Symbols são chamados de *atoms*(átomos). Você também pode compará-los à enumerações nomeadas (enum) em C. Para mais detalhes veja Symbole o `Symbol` *object wrapper*em JavaScript.

Objetos

Na ciência da computação, um objeto é um valor na memória que pode ser possivelmente referenciado por um identifier.

Propriedades

No JavaScript, obejtos podem ser vistos como uma coleção de propriedades. Com o object literal syntax, um conjunto limitado de propriedades podem ser inicializados; a partir daí propriedades podem ser adicionadas e removidas. Estas propriedades podem assumir valores de qualquer tipo, incluindo outros objetos, o que permite construir estruturas de dados mais complexas. Propriedades são identificadas usando valores chave. Um valor chave pode ser uma String ou um valor Symbol.

Existem dois tipos de propriedades de objetos que contém certos atributos: a propriedade de dados e a propriedade de acesso.

Propriedade de Dados

Associa uma chave com um valor e tem os seguintes atributos:

Attributes of a data property

Atributo	Tipo	Descrição	Valor default
[[Value]]	Qualquer tipo JavaScript	Valor retornado por uma chamada get da propriedade.	Indefinido
[[Writable]]	Booleano	Se `falso`, o[[Value]] da propriedade não pode ser alterado.	falso
[[Enumerable]]	Booleano	Se `true`, a propriedade será enumerda em for...inloops.	falso
[[Configurable]]	Booleano	Se `falso`, a propriedade não pode ser deletada e atributos além de [[Value]] e [[Writable]] não podem ser alterados.	falso

Propriedade de acesso

Associa uma chave com uma ou duas funções de acesso (get e set) para retornar ou armazenar um valor e tem os seguintes atributos:

Atributos de uma propriedade de acesso

Atributo	Tipo	Descrição	Valor default
[[Get]]	Função objeto ou indefinido	A função é chamada com uma lista de argumentos vazia e retorna o valor da propriedade sempre que é realizado um acesso get ao valor. Veja também: `get`.	Indefinido
[[Set]]	Função objeto ou indefinido	A função é chamada com um argumento o valor designado e é executada sempre que houver uma tentativa de alteração de uma propriedade específica. Veja também: `set`.	Indefinido
[[Enumerable]]	Booleano	Se `true`, a propriedade será enumerdada em for...inloops.	falso
[[Configurable]]	Boolean	Se `falso`, a propriedade não pode ser deletada e não pode ser alterada para uma propriedade de dados.	falso

Funções e objetos "normais"

Um objeto do JavaScript é um mapeamento entre chaves e valores. Chaves são Strings e valores podem ser de qualquer tipo. Isso faz com que objetos sejam perfeitos para hashmaps.

Funções são objetos comuns com a capacidade adicional de serem chamados.

Datas

Quando reprensentando datas, a melhor escolha é utilizar o `Date` utilityjá construído internamente no JavaScript.

Coleções indexadas: Arrays e Arrays tipados

Arrayssão objetos comuns nos quais existe uma relação particular entre propriedades de chaveamento inteiro e a propriedade 'length'. Além disso, arrays herdam de `Array.prototype` que nos dá vários metodos úteis para manipulação de arrays. Por exemplo, `indexOf` (procura um valor no array) ou `push` (adiciona um valor no array), etc. Isso faz com que os Arrays sejam candidatos perfeitos para representação de listas e conjuntos.

Typed Arrays (arrays tipados) são novos no JavaScript com ECMAScript Edition 6 e apresenta uma visão básica similar a de um array para um data buffer binário. A tabela a seguir mostra o equivalente aos tipos de dados em C:

TypedArray objects

Type	Value Range	Size in bytes	Description	Web IDL type	Equiv
`Int8Array`	`-128` to `127`	1	8-bit two's complement signed integer	`byte`	`int8`
`Uint8Array`	`0` to `255`	1	8-bit unsigned integer	`octet`	`uint`
`Uint8ClampedArray`	`0` to `255`	1	8-bit unsigned integer (clamped)	`octet`	`uint`
`Int16Array`	`-32768` to `32767`	2	16-bit two's complement signed integer	`short`	`int1`
`Uint16Array`	`0` to `65535`	2	16-bit unsigned integer	`unsigned short`	`uint`
`Int32Array`	`-2147483648` to `2147483647`	4	32-bit two's complement signed integer	`long`	`int3`
`Uint32Array`	`0` to `4294967295`	4	32-bit unsigned integer	`unsigned long`	`uint`
`Float32Array`	`-3.4E38` to `3.4E38` and `1.2E-38` is the min positive number	4	32-bit IEEE floating point number (7 significant digits e.g., `1.234567`)	`unrestricted float`	`floa`
`Float64Array`	`-1.8E308` to `1.8E308` and `5E-324` is the min positive number	8	64-bit IEEE floating point number (16 significant digits e.g., `1.23456789012345`)	`unrestricted double`	`doub`

Type	Value Range	Size in bytes	Description	Web IDL type	Equiv
`BigInt64Array`	`-2^63` to `2^63 - 1`	8	64-bit two's complement signed integer	`bigint`	`int6`
`BigUint64Array`	`0` to `2^64 - 1`	8	64-bit unsigned integer	`bigint`	`uint`

Coleções chaveadas: Maps, Sets, WeakMaps, WeakSets

Estas estruturas de dados pegam referências de objetos como chaves e foram introduzidas no ECMAScript Edition 6. `Set` e `WeakSet` representa um conjunto de objetos, enquanto `Map` e `WeakMap` associa um valor a um objeto. A diferença entre Maps e WeakMaps é que no primeiro, as chaves dos objetos podem ser enumeradas. Isso permite otimização de garbage collection no segundo.

Pode-se implementar Maps e Sets no ECMAScript 5 puro. Porém, como objetos não podem ser comparados (no sentido de "menor que", por exemplo), o desempenho de pesquisa seria necessariamente linear. Implementações nativas deles (incluindo WeakMaps) podem ter uma performance de pesquisa logarítmica em relação ao tempo constante.

Geralmente, para vincular dados a um nó DOM, pode-se setar propriedades diretamente no objeto ou usar atributos `data-*`. Isso tem a desvantagem de os dados estarem disponíveis para qualquer script em execução no mesmo contexto. Maps e WeakMaps facilitam vincular dados privadamente a um objeto.

Dados estruturados: JSON

JSON (JavaScript Object Notation) is a lightweight data-interchange format, derived from JavaScript but used by many programming languages. JSON builds universal data structures. See JSONand `JSON` for more details.

Mais objetos na biblioteca padrão

Javascript possui uma biblioteca padrão com objetos pré-construidos.Por favor olhe a referência para descobrir mais sobre objetos.

Determinando tipos usando o operador `typeof`

O operador typeof pode lhe ajudar a encontrar o tipo de sua variavel. Por favor, leia a página de referência para mais detalhes e casos de uso.

Especificações

Specification	Status	Comment
ECMAScript 1st Edition.	Standard	Initial definition.

ECMAScript 5.1 (ECMA-262) The definition of 'Types' in that specification.	Padrão	
ECMAScript 2015 (6th Edition, ECMA-262) The definition of 'ECMAScript Data Types and Values' in that specification.	Padrão	
ECMAScript (ECMA-262) The definition of 'ECMAScript Data Types and Values' in that specification.	Padrão em tempo real	

CONTACT

Brunna Croches

Developer Full Stack

brunnacroches.dev

linkedin.com/brunnacroches

github.com/brunnacroches

@brunnacroches.dev

discord.com/brunnacroches

brunnacroches@gmail.com

let's share