Módulos

Introdução

À medida que nossa aplicação cresce, queremos dividi-lo em vários arquivos, os chamados “módulos”. Um módulo pode conter uma classe ou uma biblioteca de funções para um propósito específico.

Por muito tempo, o JavaScript existiu sem uma sintaxe de módulo em nível de linguagem. Isso não foi um problema, porque inicialmente os scripts eram pequenos e simples, então não havia necessidade.

Mas eventualmente os scripts se tornaram cada vez mais complexos, então a comunidade inventou uma variedade de maneiras de organizar o código em módulos, bibliotecas especiais para carregar módulos sob demanda.

O sistema de módulos em nível de linguagem apareceu no padrão em 2015, evoluiu gradualmente desde então e agora é suportado por todos os principais navegadores e no Node.js. Então vamos estudar os módulos JavaScript modernos a partir de agora.

O que é um módulo?

Um módulo é apenas um arquivo. Um script é um módulo.

Os módulos podem carregar uns aos outros e usar diretivas especiais `export e import` para trocar funcionalidades, chamar funções de um módulo de outro:

- `export` palavra-chave que rotula variáveis e funções que devem ser acessíveis de fora do módulo atual.
- `import` permite a importação de funcionalidades de outros módulos.

Por exemplo, se tivermos um arquivo `sayHi.js` exportando uma função:

sayHi.js

```
1   export function sayHi(user) {
2     alert(`Hello, ${user}!`);
3   }
```

Então outro arquivo pode importar e usá-lo:

main.js

```
import {sayHi} from './sayHi.js';

alert(sayHi); // function...
sayHi('John'); // Hello, John!
```

A diretiva `import` carrega o módulo por caminho relativo ao arquivo atual (`./sayHi.js`) e atribui a função exportada `sayHi` à variável correspondente.

Vamos executar o exemplo no navegador.

Como os módulos suportam palavras-chave e recursos especiais, devemos informar ao navegador que um script deve ser tratado como um módulo, usando o atributo `<script type="module">`.

Assim:

index.html

```
<!doctype html>
<script type="module">
  import {sayHi} from './say.js';

  document.body.innerHTML = sayHi('John');
</script>
```

say.js

```
export function sayHi(user) {
  return `Hello, ${user}!`;
}
```

O navegador busca e avalia automaticamente o módulo importado (e suas importações, se necessário) e, em seguida, executa o script.

Recursos principais dos módulos

O que há de diferente em módulos, em comparação com scripts “regulares”?

Sempre usam “use strict”

Os módulos sempre funcionam em modo estrito. Por exemplo, atribuir a uma variável não declarada dará um erro.

error

```
<script type="module">
  a = 5; // error
</script>
```

Escopo em nível de módulo

Cada módulo tem seu próprio escopo de nível superior. Em outras palavras, variáveis e funções de nível superior de um módulo não são vistas em outros scripts.

No exemplo abaixo, dois scripts são importados e `hello.js` tenta usar a variável `user` declarada em `user.js`. Ele falha, porque é um módulo separado (você verá o erro no console):

index.html

```
<!doctype html>
<script type="module" src="user.js"></script>
<script type="module" src="hello.js"></script>
```

user.js

```
let user = "John";
```

hello.js

```
alert(user); // no such variable (each module has independent variables)
```

- `user.js` deve exportar a variável `user`.
- `hello.js` deve importar do módulo `user.js`.

Em outras palavras, com módulos, usamos importação/exportação em vez de depender de variáveis globais.

Devendo ser feito desta maneira:

user.js

```
export let user = "John";
```

hello.js

```
import {user} from './user.js';
alert(user); // no such variable (each module has independent variables)
```

Aqui estão dois scripts na mesma página, ambos type="module". Eles não veem as variáveis de nível superior um do outro:

index.html

```
<script type="module">
  // The variable is only visible in this module script
  let user = "John";
</script>

<script type="module">
  alert(user); // Error: user is not defined
</script>
```

Um código de módulo é avaliado apenas na primeira vez quando importado

Se o mesmo módulo for importado para vários outros módulos, seu código será executado apenas uma vez, na primeira importação.

A avaliação única tem consequências importantes, das quais devemos estar cientes.

Vamos ver alguns exemplos.

Primeiro, se a execução de um código de módulo traz efeitos colaterais, como mostrar uma mensagem, importá-lo várias vezes o acionará apenas uma vez – a primeira vez:

alert.js

```
alert("Module is evaluated!");
```

1.js

```
import `./alert.js`; // Module is evaluated!
```

2.js

```
import `./alert.js`; // (shows nothing)
```

Agora, vamos considerar um exemplo mais profundo.

Digamos que um módulo exporte um objeto:

admins.js

```
export let admin = {
  name: "John"
};
```

Se este módulo for importado de vários arquivos, o módulo será avaliado apenas na primeira vez, o objeto `admin` será criado e, em seguida, passado para todos os importadores adicionais.

Todos os importadores obtêm exatamente o mesmo objeto `único`:

object

```
//1.js
import {admin} from './admin.js';
admin.name = "Pete";

// 2.js
import {admin} from './admin.js';
alert(admin.name); // Pete

// Ambos 1.js e 2.js referenciam ao mesmo objeto
// Mudanças feitas em 1.js são visiveis em 2.js
```

import.meta

O objeto `import.meta` contém as informações sobre o módulo atual.

Seu conteúdo depende do ambiente. No navegador, ele contém o URL do script ou um URL de página da Web atual se estiver dentro de HTML:

import.meta

```
<script type="module">
  alert(import.meta.url); // script URL
</script>
```

Em um módulo, “this” é indefinido

Em um módulo, o nível superior `isto` é indefinido.

this

```
<script>
  alert(this); // window
</script>

<script type="module">
  alert(this); // undefined
</script>
```

Os scripts do módulo são adiados

Scripts de módulo são sempre adiados, mesmo efeito que o atributo `defer`, tanto para scripts externos quanto inline.

Em outras palavras:

- baixar scripts de módulos externos `<script type="module" src="...">` não bloqueia o processamento de HTML, eles carregam em paralelo com outros recursos.
- os scripts de módulo esperam até que o documento HTML esteja totalmente pronto (mesmo que sejam pequenos e carreguem mais rápido que o HTML) e, em seguida, são executados.
- a ordem relativa dos scripts é mantida: os scripts que vão primeiro no documento são executados primeiro.

Como efeito colateral, os scripts de módulo sempre “vêem” a página HTML totalmente carregada, incluindo os elementos HTML abaixo deles.

defer

```
<script type="module">
  alert(typeof button); // object
</script>

//Compare to regular script below:

<script>
  alert(typeof button); // button is undefined
</script>

<button id="button">Button</button>
```

Scripts externos

Os scripts externos que possuem type="module"são diferentes em dois aspectos:

Scripts externos com a mesma `src` são executados apenas uma vez:

once

```
    <script type="module" src="my.js"></script>
    <script type="module" src="my.js"></script>
```

Scripts externos que são buscados de outra origem (por exemplo, outro site) requerem cabeçalhos CORS . Em outras palavras, se um script de módulo é buscado de outra origem, o servidor remoto deve fornecer um cabeçalho `Access-Control-Allow-Origin` que permita a busca.

Exemplo

No nosso primeiro exemplo (acesse basic-modules) nós temos uma estrutura de arquivos da seguinte maneira:

```
index.html
main.js
modules/
    canvas.js
    square.js
```

Os dois módulos do diretório modules são descritos abaixo:

- `canvas.js` — contém funções relacionadas à configuração da tela:
- `create()` — cria uma tela com uma largura e altura especificadas dentro de um invólucro `<div>` com um ID especificado, que é anexado dentro de um elemento pai especificado. Retorna um objeto que contém o contexto 2D da tela e o ID do wrapper.
- `createReportList()` — cria uma lista não ordenada anexada dentro de um elemento de wrapper especificado, que pode ser usado para gerar dados de relatório. Retorna o ID da lista.
- `square.js` — contém:
- `name` — uma constante contendo a string `'square'`.

- `draw()` — desenha um quadrado em uma tela especificada, com um tamanho, posição e cor especificados. Retorna um objeto que contém o tamanho, a posição e a cor do quadrado.
- `reportArea()` — grava a área de um quadrado em uma lista de relatórios específica, considerando seu tamanho.
- `reportPerimeter()` — grava o perímetro de um quadrado em uma lista de relatórios específica, considerando seu comprimento.

Neste exemplo, usamos extensões .js para nossos arquivos de módulo, mas em outros recursos você pode ver a extensão .mjs usada.

Exportando recursos do módulo

A primeira coisa que você faz para obter acesso aos recursos do módulo é exportá-los. Isso é feito usando a declaração `export`.

A maneira mais fácil de usá-lo é colocá-lo na frente de qualquer item que você queira exportar para fora do módulo, por exemplo:

export

```
export const name = 'square';

export function draw(ctx, length, x, y, color) {
  ctx.fillStyle = color;
  ctx.fillRect(x, y, length, length);

  return {
    length: length,
    x: x,
    y: y,
    color: color
  };
}
```

Você pode exportar `functions, var, let, const, e classes`. Eles precisam ser itens de nível superior; você não pode usar a exportação dentro de uma função, por exemplo.

Uma maneira mais conveniente de exportar todos os itens que você deseja exportar é usar uma única instrução de exportação no final do arquivo do módulo, seguida por uma lista separada por vírgula dos recursos que você deseja exportar envoltos em chaves. Por exemplo:

```
export { name, draw, reportArea, reportPerimeter };
```

Importando recursos para o seu script

Depois de exportar alguns recursos do seu módulo, é necessário importá-los para o script para poder usá-los. A maneira mais simples de fazer isso é a seguinte:

```
import { name, draw, reportArea, reportPerimeter } from './modules/square.js';
```

Você usa o `import`, seguido por uma lista separada por vírgula dos recursos que você deseja importar agrupados em chaves, seguidos pela palavra-chave `de`, seguido pelo caminho para o arquivo do módulo - um caminho relativo à raiz do site, que para o exemplo `a` seria `/js-examples/modules/basic-modules`.

No entanto, escrevemos o caminho de maneira um pouco diferente - estamos usando a sintaxe de ponto (.) Para significar "o local atual", seguido pelo caminho além do arquivo que estamos tentando encontrar. Isso é muito melhor do que escrever todo o caminho relativo a cada vez, pois é mais curto e torna o URL portátil - o exemplo ainda funcionará se você o mover para um local diferente na hierarquia do site.

Então, por exemplo:

```
/js-examples/modules/basic-modules/modules/square.js
```

torna-se

```
./modules/square.js
```

Depois de importar os recursos para o seu script, você pode usá-los exatamente como eles foram definidos no mesmo arquivo.

Exportações padrão versus exportações nomeadas

A funcionalidade que exportamos até agora foi composta por named exports — cada item (seja uma função, const, etc.) foi referido por seu nome na exportação e esse nome também foi usado para se referir a ele na importação.

Há também um tipo de exportação chamado default export — isso foi projetado para facilitar a função padrão fornecida por um módulo e também ajuda os módulos JavaScript a interoperar com os sistemas de módulos existentes.

Vejamos um exemplo ao explicar como ele funciona. Nos nossos módulos básicos square.js você pode encontrar uma função chamada randomSquare() que cria um quadrado com cor, tamanho e posição aleatórios. Queremos exportar isso como padrão, portanto, na parte inferior do arquivo, escrevemos isso:

```
export default randomSquare;
```

Note a falta dos colchetes. No nosso arquivo main.js., importamos a função padrão usando esta linha:

```
import randomSquare from './modules/square.js';
```

Isso ocorre porque há apenas uma exportação padrão permitida por módulo e sabemos que randomSquare é isso.

Evitando conflitos de nomenclatura

Renomeando importações e exportações

Dentro dos colchetes da instrução de importação e exportação, você pode usar a palavra-chave junto com um novo nome de recurso, para alterar o nome de identificação que será usado para um recurso dentro do módulo de nível superior.

Por exemplo, os dois itens a seguir executariam o mesmo trabalho, embora de uma maneira ligeiramente diferente:

rename1

```
1  //  module.js
2  export {
3    function1 as newFunctionName,
4    function2 as anotherNewFunctionName
5  };
6
7  //  main.js
8  import { newFunctionName, anotherNewFunctionName } from
'./modules/module.js';
```

rename2

```
1  //  module.js
2  export { function1, function2 };
3
4  //  main.js
5  import { function1 as newFunctionName,
6           function2 as anotherNewFunctionName } from
'./modules/module.js';
```

O método acima funciona bem, mas é um pouco confuso e longo. Uma solução ainda melhor é importar os recursos de cada módulo dentro de um objeto de módulo. O seguinte formulário de sintaxe faz isso:

```
import * as Module from './modules/module.js';
```

Isso captura todas as exportações disponíveis no module.js e as torna disponíveis como membros de um objeto Module, efetivamente dando o seu próprio namespace. Então, por exemplo:

module

```
    Module.function1()
    Module.function2()
    etc.
```

Módulos e classes

Como sugerimos anteriormente, você também pode exportar e importar classes; essa é outra opção para evitar conflitos no seu código e é especialmente útil se você já tiver o código do módulo gravado em um estilo orientado a objetos. Observe o diretório classes

square.js

```
    class Square {
      constructor(ctx, listId, length, x, y, color) {
        ...
      }

      draw() {
        ...
      }

      ...
    }
```

que exportamos então:

```
export { Square };
```

No main.js, nós o importamos assim:

```
import { Square } from './modules/square.js';
```

E então use a classe para desenhar nosso quadrado:

squase

```
    let square1 = new Square(myCanvas.ctx, myCanvas.listId, 50, 50, 100,
'blue');
    square1.draw();
    square1.reportArea();
    square1.reportPerimeter();
```

Exercício

- Modifique o código criado no tópico 09 para utilizar a estrutura de módulos