31st July 2019

The "new Function" syntax

There's one more way to create a function. It's rarely used, but sometimes there's no alternative.

Syntax

The syntax for creating a function:

```
let func = new Function ([arg1, arg2, ...argN], functionBody);
```

The function is created with the arguments `arg1...argN` and the given `functionBody`.

It's easier to understand by looking at an example. Here's a function with two arguments:

```
let sum = new Function('a', 'b', 'return a + b');

alert( sum(1, 2) ); // 3
```

And here there's a function without arguments, with only the function body:

```
let sayHi = new Function('alert("Hello")');

sayHi(); // Hello
```

The major difference from other ways we've seen is that the function is created literally from a string, that is passed at run time.

All previous declarations required us, programmers, to write the function code in the script.

But `new Function` allows to turn any string into a function. For example, we can receive a new function from a server and then execute it:

```
let str = ... receive the code from a server dynamically ...

let func = new Function(str);
func();
```

It is used in very specific cases, like when we receive code from a server, or to dynamically compile a function from a template, in complex web-applications.

Closure

Usually, a function remembers where it was born in the special property `[[Environment]]`. It references the Lexical Environment from where it's created (we covered that in the chapter Variable scope).

But when a function is created using `new Function`, its `[[Environment]]` is set to reference not the current Lexical Environment, but the global one.

So, such function doesn't have access to outer variables, only to the global ones.

```
function getFunc() {
  let value = "test";

  let func = new Function('alert(value)');

  return func;
}

getFunc()(); // error: value is not defined
```

Compare it with the regular behavior:

```
function getFunc() {
  let value = "test";

  let func = function() { alert(value); };

  return func;
}

getFunc()(); // "test", from the Lexical Environment of getFunc
```

This special feature of `new Function` looks strange, but appears very useful in practice.

Imagine that we must create a function from a string. The code of that function is not known at the time of writing the script (that's why we don't use regular functions), but will be known in the process of execution. We may receive it from the server or from another source.

Our new function needs to interact with the main script.

What if it could access the outer variables?

The problem is that before JavaScript is published to production, it's compressed using a *minifier* – a special program that shrinks code by removing extra comments, spaces and – what's important, renames local variables into shorter ones.

For instance, if a function has `let userName`, minifier replaces it `let a` (or another letter if this one is occupied), and does it everywhere. That's usually a safe thing to do, because the variable is local, nothing outside the function can access it. And inside the function, minifier replaces every mention of it. Minifiers are smart, they analyze the code structure, so they don't break anything. They're not just a dumb find-and-replace.

So if `new Function` had access to outer variables, it would be unable to find renamed `userName`.

If `new Function` had access to outer variables, it would have problems with minifiers.

Besides, such code would be architecturally bad and prone to errors.

To pass something to a function, created as `new Function`, we should use its arguments.

Summary

The syntax:

```
let func = new Function ([arg1, arg2, ...argN], functionBody);
```

For historical reasons, arguments can also be given as a comma-separated list.

These three declarations mean the same:

```
new Function('a', 'b', 'return a + b'); // basic syntax
new Function('a,b', 'return a + b'); // comma-separated
new Function('a , b', 'return a + b'); // comma-separated with spaces
```

Functions created with `new Function`, have `[[Environment]]` referencing the global Lexical Environment, not the outer one. Hence, they cannot use outer variables. But that's actually good, because it insures us from errors. Passing parameters explicitly is a much better method architecturally and causes no problems with minifiers.

Previous lesson Next lesson

Share

Tutorial map

Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen…)