June 18, 2022

Introduction: callbacks

We use browser methods in examples here

To demonstrate the use of callbacks, promises and other abstract concepts, we'll be using some browser methods: specifically, loading scripts and performing simple document manipulations.

If you're not familiar with these methods, and their usage in the examples is confusing, you may want to read a few chapters from the next part of the tutorial.

Although, we'll try to make things clear anyway. There won't be anything really complex browser-wise.

Many functions are provided by JavaScript host environments that allow you to schedule *asynchronous* actions. In other words, actions that we initiate now, but they finish later.

For instance, one such function is the `setTimeout` function.

There are other real-world examples of asynchronous actions, e.g. loading scripts and modules (we'll cover them in later chapters).

Take a look at the function `loadScript(src)`, that loads a script with the given `src`:

```
function loadScript(src) {
  // creates a <script> tag and append it to the page
  // this causes the script with given src to start loading and run wher
  let script = document.createElement('script');
  script.src = src;
  document.head.append(script);
}
```

It inserts into the document a new, dynamically created, tag `<script src="…">` with the given `src`. The browser automatically starts loading it and executes when complete.

We can use this function like this:

```
// load and execute the script at the given path
loadScript('/my/script.js');
```

The script is executed "asynchronously", as it starts loading now, but runs later, when the function has already finished.

If there's any code below `loadScript(…)`, it doesn't wait until the script loading finishes.

```
loadScript('/my/script.js');
// the code below loadScript
// doesn't wait for the script loading to finish
// ...
```

Let's say we need to use the new script as soon as it loads. It declares new functions, and we want to run them.

But if we do that immediately after the `loadScript(…)` call, that wouldn't work:

```
loadScript('/my/script.js'); // the script has "function newFunction()

newFunction(); // no such function!
```

Naturally, the browser probably didn't have time to load the script. As of now, the `loadScript` function doesn't provide a way to track the load completion. The script loads and eventually runs, that's all. But we'd like to know when it happens, to use new functions and variables from that script.

Let's add a `callback` function as a second argument to `loadScript` that should execute when the script loads:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(script);

  document.head.append(script);
}
```

The `onload` event is described in the article Resource loading: onload and onerror, it basically executes a function after the script is loaded and executed.

Now if we want to call new functions from the script, we should write that in the callback:

```
loadScript('/my/script.js', function() {
  // the callback runs after the script is loaded
  newFunction(); // so now it works
  ...
});
```

That's the idea: the second argument is a function (usually anonymous) that runs when the action is completed.

Here's a runnable example with a real script:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;
  script.onload = () => callback(script);
  document.head.append(script);
}

loadScript('https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodas
  alert(`Cool, the script ${script.src} is loaded`);
  alert( _ ); // _ is a function declared in the loaded script
});
```

That’s called a “callback-based” style of asynchronous programming. A function that does something asynchronously should provide a `callback` argument where we put the function to run after it’s complete.

Here we did it in `loadScript`, but of course it’s a general approach.

Callback in callback

How can we load two scripts sequentially: the first one, and then the second one after it?

The natural solution would be to put the second `loadScript` call inside the callback, like this:

```
loadScript('/my/script.js', function(script) {

  alert(`Cool, the ${script.src} is loaded, let's load one more`);

  loadScript('/my/script2.js', function(script) {
    alert(`Cool, the second script is loaded`);
  });

});
```

After the outer `loadScript` is complete, the callback initiates the inner one.

What if we want one more script…?

```
loadScript('/my/script.js', function(script) {

  loadScript('/my/script2.js', function(script) {

    loadScript('/my/script3.js', function(script) {
      // ...continue after all scripts are loaded
    });

  });

});
```

So, every new action is inside a callback. That’s fine for few actions, but not good for many, so we’ll see other variants soon.

Handling errors

In the above examples we didn’t consider errors. What if the script loading fails? Our callback should be able to react on that.

Here’s an improved version of `loadScript` that tracks loading errors:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(null, script);
  script.onerror = () => callback(new Error(`Script load error for ${sr

  document.head.append(script);
}
```

It calls `callback(null, script)` for successful load and `callback(error)` otherwise.

The usage:

```
loadScript('/my/script.js', function(error, script) {
  if (error) {
    // handle error
  } else {
    // script loaded successfully
  }
});
```

Once again, the recipe that we used for `loadScript` is actually quite common. It’s called the “error-first callback” style.

The convention is:

1. The first argument of the `callback` is reserved for an error if it occurs. Then `callback(err)` is called.
2. The second argument (and the next ones if needed) are for the successful result. Then `callback(null, result1, result2…)` is called.

So the single `callback` function is used both for reporting errors and passing back results.

Pyramid of Doom

At first glance, it looks like a viable approach to asynchronous coding. And indeed it is. For one or maybe two nested calls it looks fine.

But for multiple asynchronous actions that follow one after another, we'll have code like this:

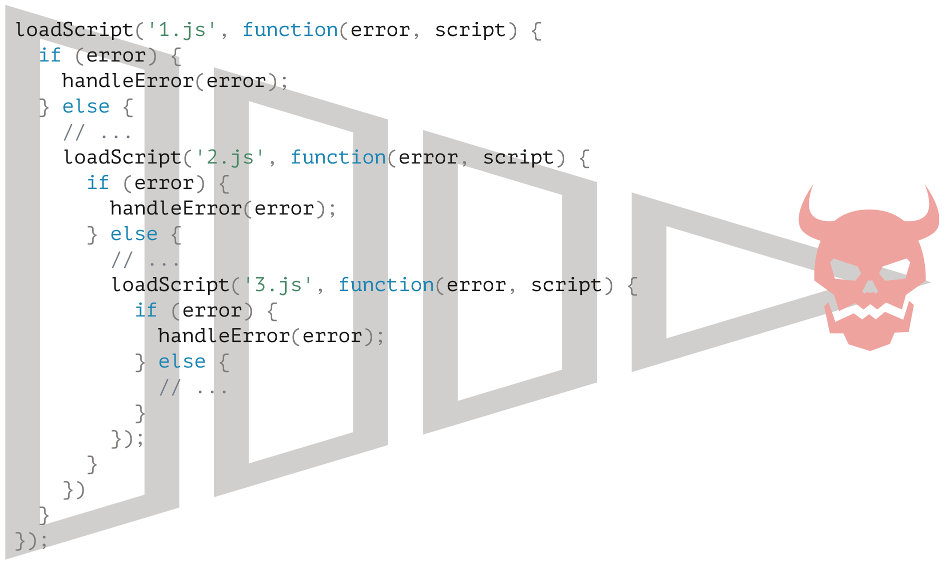
```
loadScript('1.js', function(error, script) {

  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('2.js', function(error, script) {
      if (error) {
        handleError(error);
      } else {
        // ...
        loadScript('3.js', function(error, script) {
          if (error) {
            handleError(error);
          } else {
            // ...continue after all scripts are loaded (*)
          }
        });

      }
    });
  }
});
```

In the code above:

1. We load `1.js`, then if there's no error…
2. We load `2.js`, then if there's no error…
3. We load `3.js`, then if there's no error – do something else `(*)`.

As calls become more nested, the code becomes deeper and increasingly more difficult to manage, especially if we have real code instead of `...` that may include more loops, conditional statements and so on.

That's sometimes called "callback hell" or "pyramid of doom."



```
loadScript('1.js', function(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('2.js', function(error, script) {
      if (error) {
        handleError(error);
      } else {
        // ...
        loadScript('3.js', function(error, script) {
          if (error) {
            handleError(error);
          } else {
            // ...
          }
        });
      }
    })
  }
});
```

The “pyramid” of nested calls grows to the right with every asynchronous action. Soon it spirals out of control.

So this way of coding isn’t very good.

We can try to alleviate the problem by making every action a standalone function, like this:

```
loadScript('1.js', step1);

function step1(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('2.js', step2);
  }
}

function step2(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('3.js', step3);
  }
}

function step3(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...continue after all scripts are loaded (*)
  }
}
```

See? It does the same thing, and there’s no deep nesting now because we made every action a separate top-level function.

It works, but the code looks like a torn apart spreadsheet. It's difficult to read, and you probably noticed that one needs to eye-jump between pieces while reading it. That's inconvenient, especially if the reader is not familiar with the code and doesn't know where to eye-jump.

Also, the functions named `step*` are all of single use, they are created only to avoid the "pyramid of doom." No one is going to reuse them outside of the action chain. So there's a bit of namespace cluttering here.

We'd like to have something better.

Luckily, there are other ways to avoid such pyramids. One of the best ways is to use "promises", described in the next chapter.

Previous lesson Next lesson

Share Tutorial map

Comments