8th December 2019

Promisification

“Promisification” is a long word for a simple transformation. It’s the conversion of a function that accepts a callback into a function that returns a promise.

Such transformations are often required in real-life, as many functions and libraries are callback-based. But promises are more convenient, so it makes sense to promisify them.

For instance, we have `loadScript(src, callback)` from the chapter Introduction: callbacks.

```
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 ${src}`));

  document.head.append(script);
}

// usage:
// loadScript('path/script.js', (err, script) => {...})
```

Let’s promisify it. The new `loadScriptPromise(src)` function achieves the same result, but it accepts only `src` (no `callback`) and returns a promise.

```
let loadScriptPromise = function(src) {
  return new Promise((resolve, reject) => {
    loadScript(src, (err, script) => {
      if (err) reject(err)
      else resolve(script);
    });
  })
}

// usage:
// loadScriptPromise('path/script.js').then(...)
```

Now `loadScriptPromise` fits well in promise-based code.

As we can see, it delegates all the work to the original `loadScript`, providing its own callback that translates to promise `resolve/reject`.

In practice we'll probably need to promisify many functions, so it makes sense to use a helper. We'll call it `promisify(f)`: it accepts a to-promisify function `f` and returns a wrapper function.

That wrapper does the same as in the code above: returns a promise and passes the call to the original `f`, tracking the result in a custom callback:

```
function promisify(f) {
  return function (...args) { // return a wrapper-function
    return new Promise((resolve, reject) => {
      function callback(err, result) { // our custom callback for f
        if (err) {
          return reject(err);
        } else {
          resolve(result);
        }
      }

      args.push(callback); // append our custom callback to the end of f argu

      f.call(this, ...args); // call the original function
    });
  };
};

// usage:
let loadScriptPromise = promisify(loadScript);
loadScriptPromise(...).then(...);
```

Here we assume that the original function expects a callback with two arguments `(err, result)`. That's what we encounter most often. Then our custom callback is in exactly the right format, and `promisify` works great for such a case.

But what if the original `f` expects a callback with more arguments `callback(err, res1, res2, ...)`?

Here's a more advanced version of `promisify`: if called as `promisify(f, true)`, the promise result will be an array of callback results `[res1, res2, ...]`:

```
// promisify(f, true) to get array of results
function promisify(f, manyArgs = false) {
  return function (...args) {
    return new Promise((resolve, reject) => {
      function callback(err, ...results) { // our custom callback for f
        if (err) {
          return reject(err);
        } else {
          // resolve with all callback results if manyArgs is specified
          resolve(manyArgs ? results : results[0]);
        }
      }

      args.push(callback);

      f.call(this, ...args);
    });
  };
};
```

```
20
21  // usage:
22  f = promisify(f, true);
23  f(...).then(arrayOfResults => ..., err => ...)
```

For more exotic callback formats, like those without `err` at all: `callback(result)`, we can promisify such functions manually without using the helper.

There are also modules with a bit more flexible promisification functions, e.g. es6-promisify. In Node.js, there's a built-in `util.promisify` function for that.

Please note:

Promisification is a great approach, especially when you use `async/await` (see the next chapter), but not a total replacement for callbacks.

Remember, a promise may have only one result, but a callback may technically be called many times.

So promisification is only meant for functions that call the callback once. Further calls will be ignored.

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…)