27th December 2019

Function binding

When passing object methods as callbacks, for instance to `setTimeout` , there's a known problem: "losing `this` ".

In this chapter we'll see the ways to fix it.

Losing "this"

We've already seen examples of losing `this` . Once a method is passed somewhere separately from the object – `this` is lost.

Here's how it may happen with `setTimeout` :

```
let user = {
  firstName: "John",
  sayHi() {
    alert(`Hello, ${this.firstName}!`);
  }
};

setTimeout(user.sayHi, 1000); // Hello, undefined!
```

As we can see, the output shows not "John" as `this.firstName` , but `undefined` !

That's because `setTimeout` got the function `user.sayHi` , separately from the object. The last line can be rewritten as:

```
let f = user.sayHi;
setTimeout(f, 1000); // lost user context
```

The method `setTimeout` in-browser is a little special: it sets `this=window` for the function call (for Node.js, `this` becomes the timer object, but doesn't really matter here). So for `this.firstName` it tries to get `window.firstName` , which does not exist. In other similar cases, usually `this` just becomes `undefined` .

The task is quite typical – we want to pass an object method somewhere else (here – to the scheduler) where it will be called. How to make sure that it will be called in the right context?

Solution 1: a wrapper

The simplest solution is to use a wrapping function:

```
let user = {
  firstName: "John",
  sayHi() {
    alert(`Hello, ${this.firstName}!`);
  }
};

setTimeout(function() {
  user.sayHi(); // Hello, John!
}, 1000);
```

Now it works, because it receives `user` from the outer lexical environment, and then calls the method normally.

The same, but shorter:

```
setTimeout(() => user.sayHi(), 1000); // Hello, John!
```

Looks fine, but a slight vulnerability appears in our code structure.

What if before `setTimeout` triggers (there's one second delay!) `user` changes value? Then, suddenly, it will call the wrong object!

```
let user = {
  firstName: "John",
  sayHi() {
    alert(`Hello, ${this.firstName}!`);
  }
};

setTimeout(() => user.sayHi(), 1000);

// ...the value of user changes within 1 second
user = {
  sayHi() { alert("Another user in setTimeout!"); }
};

// Another user in setTimeout!
```

The next solution guarantees that such thing won't happen.

Solution 2: bind

Functions provide a built-in method bind that allows to fix `this` .

The basic syntax is:

```
// more complex syntax will come a little later
let boundFunc = func.bind(context);
```

The result of `func.bind(context)` is a special function-like "exotic object", that is callable as function and transparently passes the call to `func` setting `this=context` .

In other words, calling `boundFunc` is like `func` with fixed `this`.

For instance, here `funcUser` passes a call to `func` with `this=user`:

```
let user = {
  firstName: "John"
};

function func() {
  alert(this.firstName);
}

let funcUser = func.bind(user);
funcUser(); // John
```

Here `func.bind(user)` as a "bound variant" of `func`, with fixed `this=user`.

All arguments are passed to the original `func` "as is", for instance:

```
let user = {
  firstName: "John"
};

function func(phrase) {
  alert(phrase + ', ' + this.firstName);
}

// bind this to user
let funcUser = func.bind(user);

funcUser("Hello"); // Hello, John (argument "Hello" is passed, and this=user)
```

Now let's try with an object method:

```
let user = {
  firstName: "John",
  sayHi() {
    alert(`Hello, ${this.firstName}!`);
  }
};

let sayHi = user.sayHi.bind(user); // (*)

// can run it without an object
sayHi(); // Hello, John!

setTimeout(sayHi, 1000); // Hello, John!

// even if the value of user changes within 1 second
// sayHi uses the pre-bound value
user = {
  sayHi() { alert("Another user in setTimeout!"); }
};
```

In the line `(*)` we take the method `user.sayHi` and bind it to `user`. The `sayHi` is a “bound” function, that can be called alone or passed to `setTimeout` – doesn’t matter, the context will be right.

Here we can see that arguments are passed “as is”, only `this` is fixed by `bind`:

```
let user = {
  firstName: "John",
  say(phrase) {
    alert(`${phrase}, ${this.firstName}!`);
  }
};

let say = user.say.bind(user);

say("Hello"); // Hello, John ("Hello" argument is passed to say)
say("Bye"); // Bye, John ("Bye" is passed to say)
```

Convenience method: `bindAll`

If an object has many methods and we plan to actively pass it around, then we could bind them all in a loop:

```
for (let key in user) {
  if (typeof user[key] == 'function') {
    user[key] = user[key].bind(user);
  }
}
```

JavaScript libraries also provide functions for convenient mass binding , e.g. _.bindAll(obj) in lodash.

Partial functions

Until now we have only been talking about binding `this`. Let’s take it a step further.

We can bind not only `this`, but also arguments. That’s rarely done, but sometimes can be handy.

The full syntax of `bind`:

```
let bound = func.bind(context, [arg1], [arg2], ...);
```

It allows to bind context as `this` and starting arguments of the function.

For instance, we have a multiplication function `mul(a, b)`:

```
function mul(a, b) {
  return a * b;
}
```

Let’s use `bind` to create a function `double` on its base:

```
function mul(a, b) {
  return a * b;
}

let double = mul.bind(null, 2);

alert( double(3) ); // = mul(2, 3) = 6
alert( double(4) ); // = mul(2, 4) = 8
alert( double(5) ); // = mul(2, 5) = 10
```

The call to `mul.bind(null, 2)` creates a new function `double` that passes calls to `mul`, fixing `null` as the context and `2` as the first argument. Further arguments are passed "as is".

That's called partial function application – we create a new function by fixing some parameters of the existing one.

Please note that here we actually don't use `this` here. But `bind` requires it, so we must put in something like `null`.

The function `triple` in the code below triples the value:

```
function mul(a, b) {
  return a * b;
}

let triple = mul.bind(null, 3);

alert( triple(3) ); // = mul(3, 3) = 9
alert( triple(4) ); // = mul(3, 4) = 12
alert( triple(5) ); // = mul(3, 5) = 15
```

Why do we usually make a partial function?

The benefit is that we can create an independent function with a readable name (`double`, `triple`). We can use it and not provide the first argument every time as it's fixed with `bind`.

In other cases, partial application is useful when we have a very generic function and want a less universal variant of it for convenience.

For instance, we have a function `send(from, to, text)`. Then, inside a `user` object we may want to use a partial variant of it: `sendTo(to, text)` that sends from the current user.

Going partial without context

What if we'd like to fix some arguments, but not the context `this`? For example, for an object method.

The native `bind` does not allow that. We can't just omit the context and jump to arguments.

Fortunately, a function `partial` for binding only arguments can be easily implemented.

Like this:

```
function partial(func, ...argsBound) {
  return function(...args) { // (*)
```

```
    return func.call(this, ...argsBound, ...args);
  }
}

// Usage:
let user = {
  firstName: "John",
  say(time, phrase) {
    alert(`[${time}] ${this.firstName}: ${phrase}!`);
  }
};

// add a partial method with fixed time
user.sayNow = partial(user.say, new Date().getHours() + ':' + new Date().getM

user.sayNow("Hello");
// Something like:
// [10:00] John: Hello!
```

The result of `partial(func[, arg1, arg2...])` call is a wrapper `(*)` that calls `func` with:

- Same `this` as it gets (for `user.sayNow` call it's `user`)
- Then gives it `...argsBound` – arguments from the `partial` call (`"10:00"`)
- Then gives it `...args` – arguments given to the wrapper (`"Hello"`)

So easy to do it with the spread syntax, right?

Also there's a ready _.partial implementation from lodash library.

Summary

Method `func.bind(context, ...args)` returns a "bound variant" of function `func` that fixes the context `this` and first arguments if given.

Usually we apply `bind` to fix `this` for an object method, so that we can pass it somewhere. For example, to `setTimeout`.

When we fix some arguments of an existing function, the resulting (less universal) function is called *partially applied* or *partial*.

Partials are convenient when we don't want to repeat the same argument over and over again. Like if we have a `send(from, to)` function, and `from` should always be the same for our task, we can get a partial and go on with it.

Tasks

Bound function as a method

importance: 5

What will be the output?

```
function f() {
  alert( this ); // ?
}

let user = {
  g: f.bind(null)
};

user.g();
```

solution

Second bind

importance: 5

Can we change `this` by additional binding?

What will be the output?

```
function f() {
  alert(this.name);
}

f = f.bind( {name: "John"} ).bind( {name: "Ann" } );

f();
```

solution

Function property after bind

importance: 5

There’s a value in the property of a function. Will it change after `bind`? Why, or why not?

```
function sayHi() {
  alert( this.name );
}
sayHi.test = 5;

let bound = sayHi.bind({
  name: "John"
});

alert( bound.test ); // what will be the output? why?
```

solution

Fix a function that loses "this"

importance: 5

The call to `askPassword()` in the code below should check the password and then call `user.loginOk/loginFail` depending on the answer.

But it leads to an error. Why?

Fix the highlighted line for everything to start working right (other lines are not to be changed).

```
function askPassword(ok, fail) {
  let password = prompt("Password?", '');
  if (password == "rockstar") ok();
  else fail();
}

let user = {
  name: 'John',

  loginOk() {
    alert(`${this.name} logged in`);
  },

  loginFail() {
    alert(`${this.name} failed to log in`);
  },

};

askPassword(user.loginOk, user.loginFail);
```

solution

Partial application for login

importance: 5

The task is a little more complex variant of Fix a function that loses "this".

The `user` object was modified. Now instead of two functions `loginOk/loginFail`, it has a single function `user.login(true/false)`.

What should we pass `askPassword` in the code below, so that it calls `user.login(true)` as `ok` and `user.login(false)` as `fail`?

```
function askPassword(ok, fail) {
  let password = prompt("Password?", '');
  if (password == "rockstar") ok();
  else fail();
}

let user = {
  name: 'John',

  login(result) {
    alert( this.name + (result ? ' logged in' : ' failed to log in') );
  }
};

askPassword(?, ?); // ?
```

Your changes should only modify the highlighted fragment.

solution

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…)