**Advanced Working with Function (Part4)**

**Decorators and forwarding, call/apply**

JavaScript gives exceptional flexibility when dealing with functions. They can be passed around, used as objects, and now we’ll see how to forward calls between them and decorate them.

## **[Transparent caching](http://javascript.info/call-apply-decorators" \l "transparent-caching)**

Let’s say we have a function slow(x) which is CPU-heavy, but its results are stable. In other words, for the same x it always returns the same result.

If the function is called often, we may want to cache (remember) the results for different x to avoid spending extra-time on recalculations.

But instead of adding that functionality into slow() we’ll create a wrapper. As we’ll see, there are many benefits of doing so.

Here’s the code, and explanations follow:

function slow(x) {

alert(`Called with ${x}`);

return x;

}

function cachingDecorator(func) {

let cache = new Map();

return function(x) {

if (cache.has(x)) { // if the result is in the map

return cache.get(x); // return it

}

let result = func(x); // otherwise call func

cache.set(x, result); // and cache (remember) the result

return result;

};

}

slow = cachingDecorator(slow);

alert( slow(1) ); // slow(1) is cached

alert( "Again: " + slow(1) ); // the same

alert( slow(2) ); // slow(2) is cached

alert( "Again: " + slow(2) ); // the same as the previous line

In the code above cachingDecorator is a decorator: a special function that takes another function and alters its behavior.

The idea is that we can call cachingDecorator for any function, and it will return the caching wrapper. That’s great, because we can have many functions that could use such a feature, and all we need to do is to apply cachingDecorator to them.

## **[Using “func.call” for the context](http://javascript.info/call-apply-decorators" \l "using-func-call-for-the-context)**

The caching decorator mentioned above is not suited to work with object methods.

For instance, in the code below worker.slow() stops working after the decoration:

// we'll make worker.slow caching

let worker = {

someMethod() {

return 1;

},

slow(x) {

alert("Called with " + x);

return x \* this.someMethod(); // (\*)

}

};

// same code as before

function cachingDecorator(func) {

let cache = new Map();

return function(x) {

if (cache.has(x)) {

return cache.get(x);

}

let result = func(x); // (\*\*)

cache.set(x, result);

return result;

};

}

alert( worker.slow(1) ); // the original method works

worker.slow = cachingDecorator(worker.slow); // now make it caching

alert( worker.slow(2) ); // Whoops! Error: Cannot read property 'someMethod' of undefined

The error occurs in the line (\*) that tries to access this.someMethod and fails. Can you see why?

The reason is that the wrapper calls the original function as func(x)in the line (\*\*). And, when called like that, the function gets this = undefined.

In our case, we can use call in the wrapper to pass the context to the original function:

let result = func.call(this, x); // "this" is passed correctly now

Now everything is fine.

To make it all clear, let’s see more deeply how this is passed along:

1. After the decoration worker.slow is now the wrapper function (x) { ... }.

2. So when worker.slow(2) is executed, the wrapper gets 2 as an argument and this=worker (it’s the object before dot).

3. Inside the wrapper, assuming the result is not yet cached, func.call(this, x) passes the current this (=worker) and the current argument (=2) to the original method.

## **[Going multi-argument with “func.apply”](http://javascript.info/call-apply-decorators" \l "going-multi-argument-with-func-apply)**

Now let’s make cachingDecorator even more universal. Till now it was working only with single-argument functions.

For instance, here say is called with this=user and messageData as a list of arguments:

function say(time, phrase) {

alert(`[${time}] ${this.name}: ${phrase}`);

}

let user = { name: "John" };

let messageData = ['10:00', 'Hello']; // become time and phrase

// user becomes this, messageData is passed as a list of arguments (time, phrase)

say.apply(user, messageData); // [10:00] John: Hello (this=user)

The only syntax difference between call and apply is that callexpects a list of arguments, while apply takes an array-like object with them.

We already know the spread operator ... from the chapter [Rest parameters and spread operator](http://javascript.info/rest-parameters-spread-operator) that can pass an array (or any iterable) as a list of arguments. So if we use it with call, we can achieve almost the same as apply.

These two calls are almost equivalent:

let args = [1, 2, 3];

func.call(context, ...args); // pass an array as list with spread operator

func.apply(context, args); // is same as using apply

If we look more closely, there’s a minor difference between such uses of call and apply.

- The spread operator ... allows to pass iterable args as the list to call.

- The apply accepts only array-like args.

So, these calls complement each other. Where we expect an iterable, call works, where we expect an array-like, apply works.

And if args is both iterable and array-like, like a real array, then we technically could use any of them, but apply will probably be faster, because it’s a single operation. Most JavaScript engines internally optimize is better than a pair call + spread.

When an external code calls such wrapper, it is indistinguishable from the call of the original function.

Now let’s bake it all into the more powerful cachingDecorator:

let worker = {

slow(min, max) {

alert(`Called with ${min},${max}`);

return min + max;

}

};

function cachingDecorator(func, hash) {

let cache = new Map();

return function() {

let key = hash(arguments); // (\*)

if (cache.has(key)) {

return cache.get(key);

}

let result = func.apply(this, arguments); // (\*\*)

cache.set(key, result);

return result;

};

}

function hash(args) {

return args[0] + ',' + args[1];

}

worker.slow = cachingDecorator(worker.slow, hash);

alert( worker.slow(3, 5) ); // works

alert( "Again " + worker.slow(3, 5) ); // same (cached)

Now the wrapper operates with any number of arguments.

There are two changes:

- In the line (\*) it calls hash to create a single key from arguments. Here we use a simple “joining” function that turns arguments (3, 5) into the key "3,5". More complex cases may require other hashing functions.

- Then (\*\*) uses func.apply to pass both the context and all arguments the wrapper got (no matter how many) to the original function.

## **[Summary](http://javascript.info/call-apply-decorators" \l "summary)**

Decorator is a wrapper around a function that alters its behavior. The main job is still carried out by the function.

It is generally safe to replace a function or a method with a decorated one, except for one little thing. If the original function had properties on it, like func.calledCount or whatever, then the decorated one will not provide them. Because that is a wrapper. So one needs to be careful if one uses them. Some decorators provide their own properties.

Decorators can be seen as “features” or “aspects” that can be added to a function. We can add one or add many. And all this without changing its code!

To implement cachingDecorator, we studied methods:

- [func.call(context, arg1, arg2…)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/call) – calls func with given context and arguments.

- [func.apply(context, args)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/apply) – calls func passing context as thisand array-like args into a list of arguments.

The generic call forwarding is usually done with apply:

let wrapper = function() {

return original.apply(this, arguments);

}

We also saw an example of method borrowing when we take a method from an object and call it in the context of another object. It is quite common to take array methods and apply them to arguments. The alternative is to use rest parameters object that is a real array.

There are many decorators there in the wild. Check how well you got them by solving the tasks of this chapter.

# Function binding

When using setTimeout with object methods or passing object methods along, there’s a known problem: "losing this".

Suddenly, this just stops working right. The situation is typical for novice developers, but happens with experienced ones as well.

## **[Losing “this”](http://javascript.info/bind" \l "losing-this)**

We already know that in JavaScript it’s easy to lose 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!

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 as we’ll see, 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](http://javascript.info/bind" \l "solution-1-a-wrapper)**

The simplest solution is to use an 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!

## 

## [**Solution 2: bind**](http://javascript.info/bind#solution-2-bind)

Functions provide a built-in method [bind](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/bind) that allows to fix 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

Now let’s try with an object method:

let user = {

firstName: "John",

sayHi() {

alert(`Hello, ${this.firstName}!`);

}

};

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

sayHi(); // Hello, John!

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

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)](http://lodash.com/docs#bindAll) in lodash.

## [**Summary**](http://javascript.info/bind#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 in an object method, so that we can pass it somewhere. For example, to setTimeout. There are more reasons to bind in the modern development, we’ll meet them later.

# Currying and partials

Till now we were only talking about binding this. Now let’s make 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:

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:

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](https://en.wikipedia.org/wiki/Partial_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:

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?

Here our benefit is that we created an independent function with a readable name (double, triple). We can use it and don’t write the first argument of every time, cause 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.

## **[Currying](http://javascript.info/currying-partials" \l "currying)**

Sometimes people mix up partial function application mentioned above with another thing named “currying”. That’s another interesting technique of working with functions that we just have to mention here.

[Currying](https://en.wikipedia.org/wiki/Currying) is translating a function from callable as f(a, b, c) into callable as f(a)(b)(c).

Let’s make curry function that performs currying for binary functions. In other words, it translates f(a, b) into f(a)(b):

function curry(func) {

return function(a) {

return function(b) {

return func(a, b);

};

};

}

// usage

function sum(a, b) {

return a + b;

}

let carriedSum = curry(sum);

alert( carriedSum(1)(2) ); // 3

As you can see, the implementation is a series of wrappers.

- The result of curry(func) is a wrapper function(a).

- When it is called like sum(1), the argument is saved in the Lexical Environment, and a new wrapper is returned function(b).

- Then sum(1)(2) finally calls function(b) providing 2, and it passes the call to the original multi-argument sum.

## **[Currying? What for?](http://javascript.info/currying-partials" \l "currying-what-for)**

Advanced currying allows both to keep the function callable normally and to get partials easily. To understand the benefits we definitely need a worthy real-life example.

For instance, we have the logging function log(date, importance, message) that formats and outputs the information. In real projects such functions also have many other useful features like: sending it over the network or filtering:

function log(date, importance, message) {

alert(`[${date.getHours()}:${date.getMinutes()}] [${importance}] ${message}`);

}

Let’s curry it!

log = \_.curry(log);

After that log still works the normal way:

log(new Date(), "DEBUG", "some debug");

…But also can be called in the curried form:

log(new Date())("DEBUG")("some debug"); // log(a)(b)(c)

Let’s get a convenience function for today’s logs:

// todayLog will be the partial of log with fixed first argument

let todayLog = log(new Date());

// use it

todayLog("INFO", "message"); // [HH:mm] INFO message

And now a convenience function for today’s debug messages:

let todayDebug = todayLog("DEBUG");

todayDebug("message"); // [HH:mm] DEBUG message

# Arrow functions revisited

Let’s revisit arrow functions.

Arrow functions are not just a “shorthand” for writing small stuff.

JavaScript is full of situations where we need to write a small function, that’s executed somewhere else.

For instance:

- arr.forEach(func) – func is executed by forEach for every array item.

- setTimeout(func) – func is executed by the built-in scheduler.

- …there are more.

It’s in the very spirit of JavaScript to create a function and pass it somewhere.

And in such functions we usually don’t want to leave the current context.

## **[Arrow functions have no “this”](http://javascript.info/arrow-functions" \l "arrow-functions-have-no-this)**

As we remember from the chapter [Object methods, "this"](http://javascript.info/object-methods), arrow functions do not have this. If this is accessed, it is taken from the outside.

For instance, we can use it to iterate inside an object method:

let group = {

title: "Our Group",

students: ["John", "Pete", "Alice"],

showList() {

this.students.forEach(

student => alert(this.title + ': ' + student)

);

}

};

group.showList();

Here in forEach, the arrow function is used, so this.title in it is exactly the same as in the outer method showList. That is: group.title.

If we used a “regular” function, there would be an error:

let group = {

title: "Our Group",

students: ["John", "Pete", "Alice"],

showList() {

this.students.forEach(function(student) {

// Error: Cannot read property 'title' of undefined

alert(this.title + ': ' + student)

});

}

};

group.showList();

The error occurs because forEach runs functions with this=undefinedby default, so the attempt to access undefined.title is made.

That doesn’t affect arrow functions, because they just don’t have this.

Arrow functions can’t run with new

Not having this naturally means another limitation: arrow functions can’t be used as constructors. They can’t be called with new.

**Arrow functions VS bind**

There’s a subtle difference between an arrow function => and a regular function called with .bind(this):

- .bind(this) creates a “bound version” of the function.

- The arrow => doesn’t create any binding. The function simply doesn’t have this. The lookup of this is made exactly the same way as a regular variable search: in the outer lexical environment.

## **[Arrows have no “arguments”](http://javascript.info/arrow-functions" \l "arrows-have-no-arguments)**

Arrow functions also have no arguments variable.

That’s great for decorators, when we need to forward a call with the current this and arguments.

For instance, defer(f, ms) gets a function and returns a wrapper around it that delays the call by ms milliseconds:

function defer(f, ms) {

return function() {

setTimeout(() => f.apply(this, arguments), ms)

};

}

function sayHi(who) {

alert('Hello, ' + who);

}

let sayHiDeferred = defer(sayHi, 2000);

sayHiDeferred("John"); // Hello, John after 2 seconds

## [**Summary**](http://javascript.info/arrow-functions#summary)

Arrow functions:

- Do not have this.

- Do not have arguments.

- Can’t be called with new.

- (They also don’t have super, but we didn’t study it. Will be in the chapter [Class inheritance, super](http://javascript.info/class-inheritance)).

Big Thank For: javascript.info ([http://javascript.info](http://javascript.info/))