JavaScript bind() Method

Summary: in this tutorial, you will learn about the JavaScript `bind()` method and know how to use it effectively.

Introduction to JavaScript bind() method

The `bind()` method returns a new function, when invoked, has its `this` sets to a specific value.

The following illustrates the syntax of the `bind()` method:

```
fn.bind(thisArg[, arg1[, arg2[, ...]]])
```

In this syntax, the `bind()` method returns a copy of the function `fn` with the specific `this` value (`thisArg`) and arguments (`arg1` , `arg2` , ...).

> Unlike the `call()` and `apply()` methods, the `bind()` method doesn't immediately execute the function. It just returns a new version of the function whose `this` sets to `thisArg` argument.

Using JavaScript bind() for function binding

When you pass a method an object is to another function as a callback, the `this` is lost. For example:

```
let person = {
    name: 'John Doe',
    getName: function() {
        console.log(this.name);
    }
};
```

```
setTimeout(person.getName, 1000);
```

Output:

```
undefined
```

As you can see clearly from the output, the `person.getName()` returns `undefined` instead of `'John Doe'`.

This is because `setTimeout()` received the function `person.getName` separately from the `person` object.

The statement:

```
setTimeout(person.getName, 1000);
```

can be rewritten as:

```
let f = person.getName;
setTimeout(f, 1000); // lost person context
```

The `this` inside the `setTimeout()` function is set to the global object in non-strict mode and `undefined` in the strict mode.

Therefore, when the callback `person.getName` is invoked, the `name` does not exist in the global object, it is set to `undefined`.

To fix the issue, you can wrap the call to the `person.getName` method in an anonymous function, like this:

```
setTimeout(function () {
    person.getName();
}, 1000);
```

This works because it gets the `person` from the outer scope and then calls the method `getName()`.

Or you can use the `bind()` method:

```
let f = person.getName.bind(person);
setTimeout(f, 1000);
```

In this code:

- First, bind the `person.getName` method to the `person` object.
- Second, pass the bound function `f` with `this` value set to the `person` object to the `setTimeout()` function.

Using bind() to borrow methods from a different object

Suppose you have a `runner` object that has the `run` () method:

```
let runner = {
    name: 'Runner',
    run: function(speed) {
        console.log(this.name + ' runs at ' + speed + ' mph.');
    }
};
```

And the `flyer` object that has the `fly()` method:

```
let flyer = {
    name: 'Flyer',
    fly: function(speed) {
        console.log(this.name + ' flies at ' + speed + ' mph.');
    }
};
```

If you want the `flyer` object to be able to run, you can use the `bind()` method to create the `run()` function with the `this` sets to the `flyer` object:

```
let run = runner.run.bind(flyer, 20);
run();
```

In this statement:

- Call the `bind()` method of the `runner.run()` method and pass in the flyer object as the first argument and 20 as the second argument.
- Invoke the `run()` function.

Output:

```
Flyer runs at 20 mph.
```

The ability to borrow a method of an object without making a copy of that method and maintain it in two separate places is very powerful in JavaScript.

Summary

- The `bind()` method creates a new function that, when invoked, has the `this` sets to a provided value.
- The `bind()` method allows an object to borrow a method from another object without making a copy of that method. This is known as function borrowing in JavaScript.