JavaScript Function Type

Summary: in this tutorial, you'll learn about the JavaScript `Function` type, which is the type of all functions in JavaScript.

Introduction to the JavaScript Function type

In JavaScript, all functions are objects. They are the instances of the `Function` type. Because functions are objects, they have properties and methods like other objects.

Functions properties

Each function has two important properties: `length` and `prototype`.

- The `length` property determines the number of named arguments specified in the function declaration.
- The `prototype` property references the actual function object.

See the following example:

```
function add(x, y) {
    return x + y;
}

console.log(add.length); // 2
console.log(add.prototype); // Object{}
```

The `add()` function accepts two arguments `x` and `y`. Therefore, the `length` property returns two.

new.target

Typically, you call a function normally like this:

```
let result = add(10,20);
console.log(result); // 30
```

Also, you can call a function with `new` keyword as a constructor:

```
let obj = new add(10,20);
```

ES6 introduced the `new.target` pseudo-property that allows you to detect whether a function or constructor was called using the `new` operator.

If a function is called normally, the `new.target` is `undefined`. However, if the function is called using the `new` keyword as a constructor, the `new.target` return a reference to the constructor.

For example:

```
function add(x, y) {
  console.log(new.target);
  return x + y;
}

let result = add(10, 20);
let obj = new add(10, 20);
```

Output:

```
undefined
[Function: add]
```

By using the `new.target`, you can control how a function will be called.

For example, to prevent the `add()` function from being called with the `new` keyword as a constructor, you can throw an error by checking the `new.target` like this:

```
function add(x, y) {
  if (new.target) {
```

```
    throw 'The add function cannot be called as a constructor';
  }
  return x + y;
}

let obj = new add(10, 20);
console.log(obj);
```

Function methods: apply, call, and bind

A function object has three important methods: `apply()`, `call()` and `bind()`.

The apply() and call() methods

The `apply()` and `call()` methods call a function with a given `this` value and arguments.

The difference between the `apply()` and `call()` is that you need to pass the arguments to the `apply()` method as an array-like object, whereas you pass the arguments to the `call()` function individually. For example:

```
let cat = { type: 'Cat', sound: 'Meow' };
let dog = { type: 'Dog', sound: 'Woof' };

const say = function (message) {
  console.log(message);
  console.log(this.type + ' says ' + this.sound);
};

say.apply(cat, ['What does a cat say?']);
say.apply(dog, ['What does a dog say?']);
```

Output:

```
What does a cat sound?
Cat says Meow
What does a dog sound?
Dog says Woof
```

In this example:

First, declare two objects `cat` and `dog` with two properties:

```
let cat = { type: 'Cat', sound: 'Meow' };
let dog = { type: 'Dog', sound: 'Woof' };
```

Second, define the `say()` function that accepts one argument:

```
const say = function (message) {
  console.log(message);
  console.log(this.type + ' says ' + this.sound);
};
```

Third, call the `say()` function via the `apply()` method:

```
say.apply(cat, ['What does a cat say?']);
```

In this example, the first argument of the `apply()` method is the `cat` object. Therefore, the `this` object in the `say()` function references the `cat` object.

Fourth, call `say()` function and pass the `dog` object:

```
say.apply(dog, ['What does a dog say?']);
```

In this example, the `this` in the `say()` function reference the `dog` object.

The `call()` method like the `apply()` method except for the way you pass the arguments to the function:

```
say.call(cat, 'What does a cat say?');
say.call(dog, 'What does a dog say?');
```

The bind() method

The `bind()` method creates a new function instance whose `this` value is bound to the object that you provide. For example:

First, define an object named `car`:

```
let car = {
    speed: 5,
    start: function() {
        console.log('Start with ' + this.speed + ' km/h');
    }
};
```

Then, define another object named `aircraft`:

```
let aircraft = {
    speed: 10,
    fly: function() {
        console.log('Flying');
    }
};
```

The aircraft has no `start()` method. To start an aircraft, you can use the `bind()` method of the `start()` method of the `car` object:

```
let taxiing = car.start.bind(aircraft);
```

In this statement, we change the `this` value inside the `start()` method of the `car` object to the `aircraft` object. The `bind()` method returns a new function that is assigned to the `taxiing` variable.

Now, you can call the `start()` method via the `taxiing` variable:

```
taxiing();
```

It will show the following message:

```
Start with 10 km/h
```

The following uses the `call()` method to call the `start()` method on the `aircraft` object:

```
car.start.call(aircraft);
```

As you can see, the `bind()` method creates a new function that you can execute later while the `call()` method executes the function immediately. This is the main difference between the `bind()` and `call()` methods.

Technically, the aircraft object borrows the `start()` method of the `car` object via the `bind()`, `call()` or `apply()` method.

For this reason, the `bind()`, `call()`, and `apply()` methods are also known as borrowing functions.

Summary

- All functions are instances of the `Function` type, which are the objects that have properties and methods.
- A function has two important properties: `length` and `prototype`.
- A function also has three important methods: `call()`, `apply()`, and `bind()`.