JavaScript Reflection

Summary: in this tutorial, you will learn about the JavaScript reflection and Reflect API in ES6.

What is reflection

In computer programming, reflection is the ability of a program to manipulate variables, properties, and methods of objects at runtime.

Prior to ES6, JavaScript already had reflection features even though they were not officially called that by the community or the specification. For example, methods like `Object.keys()`, `Object.getOwnPropertyDescriptor()`, and `Array.isArray()` are the classic reflection features.

ES6 introduces a new global object called `Reflect` that allows you to call methods, construct objects, get and set properties, and manipulate and extend properties.

The `Reflect` API is important because it allows you to develop programs and frameworks that are able to handle dynamic code.

Reflect API

Unlike the most global objects, the `Reflect` is not a constructor. It means that you cannot use `Reflect` with the `new` operator or invoke the `Reflect` as a function. It is similar to the `Math` and `JSON` objects. All the methods of the `Reflect` object are static.

- `Reflect.apply()` – call a function with specified arguments.
- `Reflect.construct()` – act like the `new` operator, but as a function. It is equivalent to calling `new target(...args)`.
- `Reflect.defineProperty()` – is similar to `Object.defineProperty()`, but return a Boolean value indicating whether or not the property was successfully defined on the object.
- `Reflect.deleteProperty()` – behave like the `delete` operator, but as a function. It's equivalent to calling the `delete objectName[propertyName]`.

- `Reflect.get()` – return the value of a property.
- `Reflect.getOwnPropertyDescriptor()` – is similar to `Object.getOwnPropertyDescriptor()`. It returns a property descriptor of a property if the property exists on the object, or `undefined` otherwise.
- `Reflect.getPrototypeOf()` – is the same as `Object.getPrototypeOf()`.
- `Reflect.has()` – work like the `in` operator, but as a function. It returns a boolean indicating whether an property (either owned or inherited) exists.
  `Reflect.isExtensible()` – is the same as `Object.isExtensible()`.
- `Reflect.ownKeys()` – return an array of the owned property keys (not inherited) of an object.
- `Reflect.preventExtensions()` – is similar to `Object.preventExtensions()`. It returns a Boolean.
- `Reflect.set()` – assign a value to a property and return a Boolean value which is true if the property is set successfully.
- `Reflect.setPrototypeOf()` – set the prototype of an object.

Let's take some examples of using the Reflect API:

Creating objects: Reflect.construct()

The `Reflect.construct()` method behaves like the `new` operator, but as a function. It is equivalent to calling the `new target(...args)` with the possibility of specifying a different prototype:

```
Reflect.construct(target, args [, newTarget])
```

The `Reflect.construct()` returns the new instance of the `target`, or the `newTarget` if specified, initialized by the `target` as a constructor with the given array-like object args. See the following example:

```
class Person {
    constructor(firstName, lastName) {
        this.firstName = firstName;
```

```
        this.lastName = lastName;
    }
    get fullName() {
        return `${this.firstName} ${this.lastName}`;
    }
};

let args = ['John', 'Doe'];

let john = Reflect.construct(
    Person,
    args
);

console.log(john instanceof Person);
console.log(john.fullName); // John Doe
```

Output

```
true
John Doe
```

In this example:

- First, define a class called `Person`.
- Second, declare an `args` array that contains two strings.
- Third, create a new instance of the `Person` class using the `Reflect.construct()` method. The `john` object is an instance of the `Person` class so it has the `fullName` property.

Calling a function: Reflect.apply()

Prior to ES6, you call a function with a specified `this` value and `arguments` by using the `Function.prototype.apply()` method. For example:

```
let result = Function.prototype.apply.call(Math.max, Math, [10, 20, 30]);
console.log(result);
```

Output:

```
30
```

This syntax is quite verbose.

The `Reflect.apply()` provides the same features as the `Function.prototype.apply()` but less verbose and easier to understand:

```
let result = Reflect.apply(Math.max, Math, [10, 20, 30]);
console.log(result);
```

Here is the syntax of the `Reflect.apply()` method:

```
Reflect.apply(target, thisArg, args)
```

Defining a property: Reflect.defineProperty()

The `Reflect.defineProperty()` is like the `Object.defineProperty()`. However, it returns a Boolean indicating whether or not the property was defined successfully instead of throwing an exception:

```
Reflect.defineProperty(target, propertyName, propertyDescriptor)
```

See the following example:

```
let person = {
    name: 'John Doe'
};

if (Reflect.defineProperty(person, 'age', {
        writable: true,
        configurable: true,
        enumerable: false,
        value: 25,
    })) {
```

```
    console.log(person.age);
} else {
    console.log('Cannot define the age property on the person object.');

}
```

In this tutorial, you have learned about the JavaScript reflection and the Reflect API which contains a number of reflective methods.