15. Metaprogramación

Los objetos `Proxy` y `Reflect`

- te permiten interceptar y definir un comportamiento personalizado para las operaciones fundamentales del lenguaje (por ejemplo, búsqueda de propiedades, asignación, enumeración, invocación de funciones, etc.).
- con la ayuda de estos dos objetos, puedes programar en el meta nivel de JavaScript.

Proxys

Los objetos `Proxy` permiten interceptar ciertas operaciones e implementar comportamientos personalizados.

- por ejemplo, obtener una propiedad en un objeto:

```
const gestor = {
  get(objetivo, nombre) {
    return nombre in objetivo ? objetivo[nombre] : 42;
  },
};

const p = new Proxy({}, gestor);
p.a = 1;
console.log(p.a, p.b);    // 1, 42
```

- el objeto `Proxy` define un **`objetivo`** (un objeto vacío aquí) y un objeto **`gestor`** (*handler*) en el que se implementa una trampa get (***get trap***).
- aquí, un objeto que se envía por proxy no devolverá **`undefined`** cuando obtenga propiedades indefinidas, sino que devolverá el número 42.

Terminología

Cuando se habla de la funcionalidad que ofrecen los proxies se utilizan los términos siguientes:

gestor/manipulador (***handler***)
: Objeto contenedor de trampas (*traps*).

trampa (***trap***)
: Los métodos que proporcionan acceso a la propiedad. (Esto es análogo al concepto de *trampas* en los sistemas operativos).

objetivo (***target***)

- es el objeto que el proxy virtualiza .
- a menudo se usa como almacenamiento backend para el proxy.
- los invariantes (semánticas que permanecen sin cambios) con respecto a la no extensibilidad del objeto o las propiedades no configurables se comprueban contra el objetivo.

invariantes

- la semántica que permanecen sin cambios cuando se implementan operaciones personalizadas se denominan **invariantes**.
- si violas un invariante de un gestor, se lanzará un `TypeError`.

`Proxy` revocable

El método `Proxy.revocable()` se utiliza para crear un objeto `Proxy` revocable.

- esto significa que el proxy se puede revocar a través de la función `revoke` y desactiva el proxy.
- posteriormente, cualquier operación en el proxy conduce a un `TypeError`.

```
const revocable = Proxy.revocable({}, {
  get(objetivo, nombre) {
    return `[[${nombre}]]`
  },
});

const proxy = revocable.proxy;
console.log(proxy.foo); // "[[foo]]"

revocable.revoke();
```

```
console.log(proxy.foo);
// TypeError: Cannot perform 'get' on a proxy that has been revoked proxy.foo = 1;
// TypeError: Cannot perform 'set' on a proxy that has been revoked delete proxy.foo;
// TypeError: Cannot perform 'deleteProperty' on a proxy that has been revoked
//    console.log(typeof proxy);
// "object", typeof doesn't trigger any trap
```

Reflection

`Reflect`

- es un objeto incorporado que proporciona métodos para operaciones de JavaScript interceptables.
- los métodos son los mismos que los de los gestores de proxy.
- no es un objeto de función.
- ayuda a reenviar las operaciones predeterminadas desde el gestor al objetivo.

Con `Reflect.has()` por ejemplo, obtienes el operador `in` como una función:

```
Reflect.has(Objeto, 'assign') // true
```

Una función `apply()` mejorada

Antes de `Reflect`, generalmente se usaba el método `Function.prototype.apply()` para llamar a una función con un valor dado y argumentos proporcionados como un array (o un objeto similar a un array).

```
Function.prototype.apply.call(Math.floor, undefined, [1.75])
```

Con `Reflect.apply` esto se vuelve menos complicado y más fácil de entender:

```
Reflect.apply(Math.floor, undefined, [1.75])
// 1

Reflect.apply(String.fromCharCode, undefined, [104, 101, 108, 108, 111])
// "Hola"

Reflect.apply(RegExp.prototype.exec, /ab/, ['confabulation']).index
// 4

Reflect.apply(''.charAt, 'ponis', [3])
// "i"
```

Comprobando si la definición de propiedad ha sido exitosa

La función `Object.defineProperty`

- devuelve un objeto si tiene éxito, o arroja un `TypeError` de lo contrario (usarías un bloque `try...catch` para detectar cualquier error que ocurriera al definir una propiedad).

El método `Reflect.defineProperty` devuelve un estado de éxito booleano, pudiendose usar entonces un bloque `if...else` así:

```
if (Reflect.defineProperty(objetivo, propiedad, atributos)) {
    // éxito
} else {
    // falla
}
```