2nd December 2019

Property flags and descriptors

As we know, objects can store properties.

Until now, a property was a simple “key-value” pair to us. But an object property is actually a more flexible and powerful thing.

In this chapter we’ll study additional configuration options, and in the next we’ll see how to invisibly turn them into getter/setter functions.

Property flags

Object properties, besides a **`value`**, have three special attributes (so-called “flags”):

- **`writable`** – if `true`, the value can be changed, otherwise it’s read-only.
- **`enumerable`** – if `true`, then listed in loops, otherwise not listed.
- **`configurable`** – if `true`, the property can be deleted and these attributes can be modified, otherwise not.

We didn’t see them yet, because generally they do not show up. When we create a property “the usual way”, all of them are `true`. But we also can change them anytime.

First, let’s see how to get those flags.

The method Object.getOwnPropertyDescriptor allows to query the *full* information about a property.

The syntax is:

```
let descriptor = Object.getOwnPropertyDescriptor(obj, propertyName);
```

`obj`

The object to get information from.

`propertyName`

The name of the property.

The returned value is a so-called “property descriptor” object: it contains the value and all the flags.

For instance:

```
let user = {
  name: "John"
```

```
};

let descriptor = Object.getOwnPropertyDescriptor(user, 'name');

alert( JSON.stringify(descriptor, null, 2 ) );
/* property descriptor:
{
  "value": "John",
  "writable": true,
  "enumerable": true,
  "configurable": true
}
*/
```

To change the flags, we can use Object.defineProperty.

The syntax is:

```
Object.defineProperty(obj, propertyName, descriptor)
```

`obj`, `propertyName`

The object and its property to apply the descriptor.

`descriptor`

Property descriptor object to apply.

If the property exists, `defineProperty` updates its flags. Otherwise, it creates the property with the given value and flags; in that case, if a flag is not supplied, it is assumed `false`.

For instance, here a property `name` is created with all falsy flags:

```
let user = {};

Object.defineProperty(user, "name", {
  value: "John"
});

let descriptor = Object.getOwnPropertyDescriptor(user, 'name');

alert( JSON.stringify(descriptor, null, 2 ) );
/*
{
  "value": "John",
  "writable": false,
  "enumerable": false,
  "configurable": false
}
 */
```

Compare it with “normally created” `user.name` above: now all flags are falsy. If that’s not what we want then we’d better set them to `true` in `descriptor`.

Now let’s see effects of the flags by example.

Non-writable

Let's make `user.name` non-writable (can't be reassigned) by changing `writable` flag:

```
let user = {
  name: "John"
};

Object.defineProperty(user, "name", {
  writable: false
});

user.name = "Pete"; // Error: Cannot assign to read only property 'name'
```

Now no one can change the name of our user, unless they apply their own `defineProperty` to override ours.

> **Errors appear only in strict mode**
>
> In the non-strict mode, no errors occur when writing to non-writable properties and such. But the operation still won't succeed. Flag-violating actions are just silently ignored in non-strict.

Here's the same example, but the property is created from scratch:

```
let user = { };

Object.defineProperty(user, "name", {
  value: "John",
  // for new properties we need to explicitly list what's true
  enumerable: true,
  configurable: true
});

alert(user.name); // John
user.name = "Pete"; // Error
```

Non-enumerable

Now let's add a custom `toString` to `user`.

Normally, a built-in `toString` for objects is non-enumerable, it does not show up in `for..in`. But if we add a `toString` of our own, then by default it shows up in `for..in`, like this:

```
let user = {
  name: "John",
  toString() {
    return this.name;
  }
};

// By default, both our properties are listed:
for (let key in user) alert(key); // name, toString
```

If we don't like it, then we can set `enumerable:false`. Then it won't appear in a `for..in` loop, just like the built-in one:

```
let user = {
  name: "John",
  toString() {
    return this.name;
  }
};

Object.defineProperty(user, "toString", {
  enumerable: false
});

// Now our toString disappears:
for (let key in user) alert(key); // name
```

Non-enumerable properties are also excluded from `Object.keys`:

```
alert(Object.keys(user)); // name
```

Non-configurable

The non-configurable flag (`configurable:false`) is sometimes preset for built-in objects and properties.

A non-configurable property can not be deleted.

For instance, `Math.PI` is non-writable, non-enumerable and non-configurable:

```
let descriptor = Object.getOwnPropertyDescriptor(Math, 'PI');

alert( JSON.stringify(descriptor, null, 2 ) );
/*
{
  "value": 3.141592653589793,
  "writable": false,
  "enumerable": false,
  "configurable": false
}
*/
```

So, a programmer is unable to change the value of `Math.PI` or overwrite it.

```
Math.PI = 3; // Error

// delete Math.PI won't work either
```

Making a property non-configurable is a one-way road. We cannot change it back with `defineProperty`.

To be precise, non-configurability imposes several restrictions on `defineProperty`:

1. Can't change `configurable` flag.
2. Can't change `enumerable` flag.
3. Can't change `writable: false` to `true` (the other way round works).
4. Can't change `get/set` for an accessor property (but can assign them if absent).

Here we are making `user.name` a "forever sealed" constant:

```
let user = { };

Object.defineProperty(user, "name", {
  value: "John",
  writable: false,
  configurable: false
});

// won't be able to change user.name or its flags
// all this won't work:
//   user.name = "Pete"
//   delete user.name
//   defineProperty(user, "name", { value: "Pete" })
Object.defineProperty(user, "name", {writable: true}); // Error
```

"Non-configurable" doesn't mean "non-writable"

Notable exception: a value of non-configurable, but writable property can be changed.

The idea of `configurable: false` is to prevent changes to property flags and its deletion, not changes to its value.

Object.defineProperties

There's a method Object.defineProperties(obj, descriptors) that allows to define many properties at once.

The syntax is:

```
Object.defineProperties(obj, {
  prop1: descriptor1,
  prop2: descriptor2
  // ...
});
```

For instance:

```
Object.defineProperties(user, {
  name: { value: "John", writable: false },
  surname: { value: "Smith", writable: false },
  // ...
});
```

So, we can set many properties at once.

Object.getOwnPropertyDescriptors

To get all property descriptors at once, we can use the method Object.getOwnPropertyDescriptors(obj).

Together with `Object.defineProperties` it can be used as a "flags-aware" way of cloning an object:

```
let clone = Object.defineProperties({}, Object.getOwnPropertyDescriptors(obj))
```

Normally when we clone an object, we use an assignment to copy properties, like this:

```
for (let key in user) {
  clone[key] = user[key]
}
```

…But that does not copy flags. So if we want a "better" clone then `Object.defineProperties` is preferred.

Another difference is that `for..in` ignores symbolic properties, but `Object.getOwnPropertyDescriptors` returns *all* property descriptors including symbolic ones.

Sealing an object globally

Property descriptors work at the level of individual properties.

There are also methods that limit access to the *whole* object:

Object.preventExtensions(obj)

Forbids the addition of new properties to the object.

Object.seal(obj)

Forbids adding/removing of properties. Sets `configurable: false` for all existing properties.

Object.freeze(obj)

Forbids adding/removing/changing of properties. Sets `configurable: false, writable: false` for all existing properties.

And also there are tests for them:

Object.isExtensible(obj)

Returns `false` if adding properties is forbidden, otherwise `true` .

Object.isSealed(obj)

Returns `true` if adding/removing properties is forbidden, and all existing properties have `configurable: false` .

Object.isFrozen(obj)

Returns `true` if adding/removing/changing properties is forbidden, and all current properties are `configurable: false, writable: false`.

These methods are rarely used in practice.

Previous lesson Next lesson

Share

Tutorial map

Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen…)