22nd October 2019

Prototype methods, objects without __proto__

In the first chapter of this section, we mentioned that there are modern methods to setup a prototype.

The `__proto__` is considered outdated and somewhat deprecated (in browser-only part of the JavaScript standard).

The modern methods are:

- Object.create(proto[, descriptors]) – creates an empty object with given `proto` as `[[Prototype]]` and optional property descriptors.
- Object.getPrototypeOf(obj) – returns the `[[Prototype]]` of `obj`.
- Object.setPrototypeOf(obj, proto) – sets the `[[Prototype]]` of `obj` to `proto`.

These should be used instead of `__proto__`.

For instance:

```
let animal = {
  eats: true
};

// create a new object with animal as a prototype
let rabbit = Object.create(animal);

alert(rabbit.eats); // true

alert(Object.getPrototypeOf(rabbit) === animal); // true

Object.setPrototypeOf(rabbit, {}); // change the prototype of rabbit to {}
```

`Object.create` has an optional second argument: property descriptors. We can provide additional properties to the new object there, like this:

```
let animal = {
  eats: true
};

let rabbit = Object.create(animal, {
  jumps: {
    value: true
  }
});
```

```
10
11  alert(rabbit.jumps); // true
```

The descriptors are in the same format as described in the chapter Property flags and descriptors.

We can use `Object.create` to perform an object cloning more powerful than copying properties in `for..in`:

```
1  // fully identical shallow clone of obj
2  let clone = Object.create(Object.getPrototypeOf(obj), Object.getOwnPropertyDe
```

This call makes a truly exact copy of `obj`, including all properties: enumerable and non-enumerable, data properties and setters/getters – everything, and with the right `[[Prototype]]`.

Brief history

If we count all the ways to manage `[[Prototype]]`, there are a lot! Many ways to do the same!

Why?

That's for historical reasons.

- The `"prototype"` property of a constructor function has worked since very ancient times.
- Later, in the year 2012, `Object.create` appeared in the standard. It gave the ability to create objects with a given prototype, but did not provide the ability to get/set it. So browsers implemented the non-standard `__proto__` accessor that allowed the user to get/set a prototype at any time.
- Later, in the year 2015, `Object.setPrototypeOf` and `Object.getPrototypeOf` were added to the standard, to perform the same functionality as `__proto__`. As `__proto__` was de-facto implemented everywhere, it was kind-of deprecated and made its way to the Annex B of the standard, that is: optional for non-browser environments.

As of now we have all these ways at our disposal.

Why was `__proto__` replaced by the functions `getPrototypeOf/setPrototypeOf`? That's an interesting question, requiring us to understand why `__proto__` is bad. Read on to get the answer.

Don't change `[[Prototype]]` on existing objects if speed matters

Technically, we can get/set `[[Prototype]]` at any time. But usually we only set it once at the object creation time and don't modify it anymore: `rabbit` inherits from `animal`, and that is not going to change.

And JavaScript engines are highly optimized for this. Changing a prototype "on-the-fly" with `Object.setPrototypeOf` or `obj.__proto__=` is a very slow operation as it breaks internal optimizations for object property access operations. So avoid it unless you know what you're doing, or JavaScript speed totally doesn't matter for you.

"Very plain" objects

As we know, objects can be used as associative arrays to store key/value pairs.

…But if we try to store *user-provided* keys in it (for instance, a user-entered dictionary), we can see an interesting glitch: all keys work fine except `"__proto__"`.

Check out the example:

```
1 let obj = {};
2
3 let key = prompt("What's the key?", "__proto__");
4 obj[key] = "some value";
5
6 alert(obj[key]); // [object Object], not "some value"!
```

Here, if the user types in `__proto__`, the assignment is ignored!

That shouldn't surprise us. The `__proto__` property is special: it must be either an object or `null`. A string can not become a prototype.

But we didn't *intend* to implement such behavior, right? We want to store key/value pairs, and the key named `"__proto__"` was not properly saved. So that's a bug!

Here the consequences are not terrible. But in other cases we may be assigning object values, and then the prototype may indeed be changed. As a result, the execution will go wrong in totally unexpected ways.

What's worse – usually developers do not think about such possibility at all. That makes such bugs hard to notice and even turn them into vulnerabilities, especially when JavaScript is used on server-side.

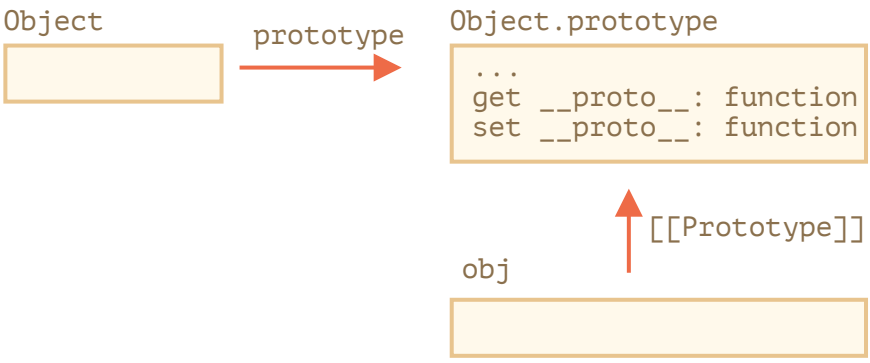
Unexpected things also may happen when assigning to `toString`, which is a function by default, and to other built-in methods.

How can we avoid this problem?

First, we can just switch to using `Map`, then everything's fine.

But `Object` can also serve us well here, because language creators gave thought to that problem long ago.

`__proto__` is not a property of an object, but an accessor property of `Object.prototype`:



So, if `obj.__proto__` is read or set, the corresponding getter/setter is called from its prototype, and it gets/sets `[[Prototype]]`.

As it was said in the beginning of this tutorial section: `__proto__` is a way to access `[[Prototype]]`, it is not `[[Prototype]]` itself.

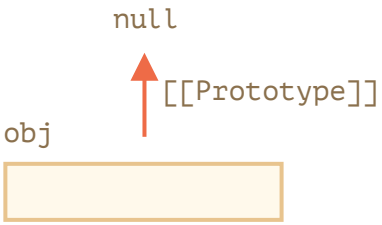
Now, if we want to use an object as an associative array, we can do it with a little trick:

```
let obj = Object.create(null);

let key = prompt("What's the key?", "__proto__");
obj[key] = "some value";

alert(obj[key]); // "some value"
```

`Object.create(null)` creates an empty object without a prototype (`[[Prototype]]` is `null`):



So, there is no inherited getter/setter for `__proto__`. Now it is processed as a regular data property, so the example above works right.

We can call such objects "very plain" or "pure dictionary" objects, because they are even simpler than the regular plain object `{...}`.

A downside is that such objects lack any built-in object methods, e.g. `toString`:

```
let obj = Object.create(null);

alert(obj); // Error (no toString)
```

…But that's usually fine for associative arrays.

Note that most object-related methods are `Object.something(...)`, like `Object.keys(obj)` – they are not in the prototype, so they will keep working on such objects:

```
let chineseDictionary = Object.create(null);
chineseDictionary.hello = "你好";
chineseDictionary.bye = "再见";

alert(Object.keys(chineseDictionary)); // hello,bye
```

Summary

Modern methods to set up and directly access the prototype are:

- Object.create(proto[, descriptors]) – creates an empty object with a given `proto` as `[[Prototype]]` (can be `null`) and optional property descriptors.
- Object.getPrototypeOf(obj) – returns the `[[Prototype]]` of `obj` (same as `__proto__` getter).
- Object.setPrototypeOf(obj, proto) – sets the `[[Prototype]]` of `obj` to `proto` (same as `__proto__` setter).

The built-in `__proto__` getter/setter is unsafe if we'd want to put user-generated keys into an object. Just because a user may enter `"__proto__"` as the key, and there'll be an error, with hopefully light, but generally unpredictable consequences.

So we can either use `Object.create(null)` to create a "very plain" object without `__proto__`, or stick to `Map` objects for that.

Also, `Object.create` provides an easy way to shallow-copy an object with all descriptors:

```
let clone = Object.create(Object.getPrototypeOf(obj), Object.getOwnPropertyDe
```

We also made it clear that `__proto__` is a getter/setter for `[[Prototype]]` and resides in `Object.prototype`, just like other methods.

We can create an object without a prototype by `Object.create(null)`. Such objects are used as "pure dictionaries", they have no issues with `"__proto__"` as the key.

Other methods:

- Object.keys(obj) / Object.values(obj) / Object.entries(obj) – returns an array of enumerable own string property names/values/key-value pairs.
- Object.getOwnPropertySymbols(obj) – returns an array of all own symbolic keys.
- Object.getOwnPropertyNames(obj) – returns an array of all own string keys.
- Reflect.ownKeys(obj) – returns an array of all own keys.
- obj.hasOwnProperty(key): returns `true` if `obj` has its own (not inherited) key named `key`.

All methods that return object properties (like `Object.keys` and others) – return "own" properties. If we want inherited ones, we can use `for..in`.

Tasks

Add toString to the dictionary

importance: 5

There's an object `dictionary`, created as `Object.create(null)`, to store any `key/value` pairs.

Add method `dictionary.toString()` into it, that should return a comma-delimited list of keys. Your `toString` should not show up in `for..in` over the object.

Here's how it should work:

```
let dictionary = Object.create(null);

// your code to add dictionary.toString method

// add some data
dictionary.apple = "Apple";
dictionary.__proto__ = "test"; // __proto__ is a regular property key here

// only apple and __proto__ are in the loop
```

```
for(let key in dictionary) {
  alert(key); // "apple", then "__proto__"
}

// your toString in action
alert(dictionary); // "apple,__proto__"
```

solution

The difference between calls

importance: 5

Let's create a new `rabbit` object:

```
function Rabbit(name) {
  this.name = name;
}
Rabbit.prototype.sayHi = function() {
  alert(this.name);
};

let rabbit = new Rabbit("Rabbit");
```

These calls do the same thing or not?

```
rabbit.sayHi();
Rabbit.prototype.sayHi();
Object.getPrototypeOf(rabbit).sayHi();
rabbit.__proto__.sayHi();
```

solution

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…)