All About Prototypes In JS

Prototype And Prototypal Inheritance

The Mystery of Prototype

- Create a constructor function named `Person`. And make a Person object, now log this newly created object.

```
function Person (firstName, lastName){
    this.firstName = firstName;
    this.lastName = lastName;
}

let johnWick = new Person("John","Wick");

console.log(johnWick);
```

- We will see the following output

```
vPerson {firstName: 'John', lastName: 'Wick'}
firstName: "John"
lastName: "Wick"
>[[Prototype]]: Object
```

- Wait! we had only **two** properties in our constructor function, but when we log it, we get **three** properties. Where did this `[[Prototype]]` thing come from.
- Let explore this `[[Prototype]]` further.

Prototype :The superpower

- In JavaScript everything is an object, even a function is an object. It is a special type of object that can be called/invoked.
- Since the function is an object, it can have properties.
- Now **every function** (not just the constructor function) has a property named `prototype`
- JS engine behind the scenes creates an object, and this prototype property on the function, refers to this engine created object.
- Every functions has its own prototype which is different from the others.
- We can access this hidden object(the prototype) like this:

```
functionName.prototype
```

- Let's console log this object, and see what we find

```
function Person (firstName, lastName){
    this.firstName = firstName;
    this.lastName = lastName;
    //since it is a constructor function, we don't have any return here
}

console.log(Person.prototype);
```

- We will get the following result

```
v{constructor: ƒ}
constructor: ƒ Person(firstName, lastName)
>[[Prototype]]: Object
```

- We see that it is an **object** with few properties like **constructor** and **[[prototype]]**.
- For now let's not focus on these properties , we will get back to them later.
- The important point to note here is that it is an object, which we can access.
- Now let's create objects from this constructor function and analyze them under our microscope 🔬.

`__proto__` :The hidden link

- Let's get back to where we started, and understand that example better.
- Create an **object** from this `Person` constructor, and log it.

```
function Person (firstName, lastName){
    this.firstName = firstName;
    this.lastName = lastName;
}

let johnWick = new Person("John","Wick");

console.log(johnWick);
```

- We got this result

```
//logs on Chrome
vPerson {firstName: 'John', lastName: 'Wick'}
firstName: "John"
lastName: "Wick"
>[[Prototype]]: Object
```

- Notice the `[[Prototype]]` property.
- If you try to access the property by writing the following code , you will get an error.

```
johnWick.[[prototype]] //syntax error
```

- Why? Because `[[prototype]]` doesn't exist in JS, it is implemented by the Chrome internally
- Similarly, Mozilla uses `<prototype>` in place of `[[prototype]]`

```
//logs on Firefox
Object { firstName: "john", lastName: "wick" }

firstName: "john

lastName: "wick"

<prototype>: Object { … }
```

- In JS, `[[prototype]]` or `<prototype>` can be accessed by `__proto__`. Hence all these 3 are same.
- But what is `__proto__`?
- Constructor functions create objects, and every object that has been created by the constructor function can access, the prototype of its constructor function. (read this again, its important!)
- The object can access the prototype of its constructor function using the `__proto__` property.
- Hence, `__proto__` is nothing but a property on an object.
- Why so many underscores then? So that users don't access and modify it by mistake.
- Now if we log the `__proto__` of an object

```
function Person (firstName, lastName){
    this.firstName = firstName;
    this.lastName = lastName;
}


let johnWick = new Person("John","Wick");

console.log(johnWick.__proto__);
```

- We will get the following output

```
v{constructor: ƒ}
constructor: ƒ Person(firstName, lastName)
>[[Prototype]]: Object
```

- Do you find any similarity? We got the same output from `console.log(Person.prototype)`
- Why same? Because `Perons.prototype` and `johnWick.__proto__` are referring to the same object in memory
- We can test this🧪

```
console.log(johnWick.__proto__ === Person.prototype); //true
```

Modifying the prototype : A way towards inheritance

- We can modify the prototype of a constructor function.

```
Person.prototype.sayHi=function(){
    console.log(`Hi! my name is ${this.firstName}`);
}
```

- Lets see if we can call this `sayHi()` method from our Person objects.

```
function Person (firstName, lastName){
    this.firstName = firstName;
    this.lastName = lastName;
}

Person.prototype.sayHi = function(){
    console.log(`Hi! my name is ${this.firstName}`);
}

let johnWick = new Person("John","Wick");

let albertEinstein = new Person("Albert","Einstein");

console.log( johnWick.sayHi() ); //Hi! my name is John
console.log( albertEinstein.sayHi() ); //Hi! my name is Albert
```

- Yes we can , but how? The `sayHi()` method was not the part of the object.

Prototype Chain🕵🏾

- We could call the `sayHi()` method on our objects even though it was not part of the object, because of something called **Prototype Chain.**

- When we call/invoke a method on an object like this:

```
johnWick.sayHi();
```

- Then the method is first searched inside the object itself, the object that called the method, `johnWick` in this case,
- If the method is not present in the calling object, then it is searched in the object linked to the calling object using `__proto__` . Remember that `__proto__` points to some other object.
- If it doesn't find the method there, it goes even further, and tries to find it inside the `__proto__` of `__proto__`.
- This goes on until the method is found or we reach where `__proto__` points to null.
- We would get `undefined` if were looking for a property in the property and didn't find it.
- And would get `reference error` if we were looking for a method.
- Hence we can conclude that if we put some method on the prototype of the constructor, then all the objects created by the constructor, would have a link to that constructor prototype using `__proto__`. In other words, `__proto__` will point to the prototype of its constructor.

Method in Constructor vs Method in Prototype

- One might ask what is the point of all this complexity when we can easily put methods in the constructor function like this:

```
function Person (firstName, lastName){
    this.sayHi = function(){
        console.log(`Hi my name is ${this.firstName}`);
    }
    this.firstName = firstName;
    this.lastName = lastName;
}
```

- The problem is memory. Every time you create a new object using the constructor function, a separate copy of the method is created for each object. This can increase the amount of memory used by our app.

- If we put our method on the prototype, all the objects can refer to this prototype to call this(`sayHi()`) method, we would have only one copy of the method and hence save the memory used by our app.

The mystery of `__proto__` inside an empty object.

- Let's create an empty object, and log it.

```
let emptyObject = {

}

console.log(emptyObject);
```

- This would be the log on Firefox console

```
vObject {  }

> <prototype>: Object { … }

/*Note : <prototype> is Firefox's way of implementing __proto__*/
```

- We didn't put any property inside our object literal , neither did we use any constructor function to create our object, then how come we have the `__proto__` on our `emptyObject` . Remember `<prototype>` is Firefox's way of implementing `__proto__` internally.
- This is because when use object literal , behind the scene we are using the `Object` constructor
- So a code like this

```
let emptyObject = {}
```

- Is treated as

```
let emptyObject = Object(null)
```

- And since `Object` is a constructor, it also has a prototype. We can see it by logging it.

```
console.log(Object.prototype)
```

- We get the following result

```
Object { … }

__defineGetter__: function __defineGetter__()

__defineSetter__: function __defineSetter__()

__lookupGetter__: function __lookupGetter__()

__lookupSetter__: function __lookupSetter__()

__proto__:

constructor: function Object()

hasOwnProperty: function hasOwnProperty()

isPrototypeOf: function isPrototypeOf()

propertyIsEnumerable: function propertyIsEnumerable()

toLocaleString: function toLocaleString()

toString: function toString()

valueOf: function valueOf()

<get __proto__()>: function __proto__()

<set __proto__()>: function __proto__()
```

- We can even check if Object is a constructor or not using the `typeof` operator.😃

```
console.log(typeof Object) //function
```

- The `__proto__` in our `emptyObject` was referring to this prototype(the `Object.prototype`) since it was built by the `Object` constructor.
- This is the reason we can use properties like `toString` on objects without defining them. With the help of prototype chaining, it goes to the final Object.prototype object, where this method resides.

Important points to note before moving further

- The prototype of the Person constructor had two properties as we discussed earlier, **constructor** and `<prototype>`, constructor refer to the constructor function of this(the prototype) object, we can even call this constructor from our prototype

```
function foo(){
    console.log("Hi I was called using my prototype");
}

console.log(foo.prototype.constructor()); //Hi I was called using my prototype
```

- The other one, the `<prototype>` or `__proto__` refers to the `Object.prototype` since all objects by default have their `__proto__` refer to `Object.prototype`. Because all object are created by the `Object` constructor.
- Another important thing to note is that only the objects that are created using the constructor function have their `__proto__` refer to the constructor prototype. An object made by the factory function will have its `__proto__` refer to Object.prototype by default and not the prototype of the factory function.
- The `Object.prototype` has `__proto__` property as well, and it points to `null`, this is where most of the prototype chains end, at null.
- All the constructors like `Array()`, `Map()` , `String()`, `Number()` etc have a prototype, as all constructors have, and these prototype objects have their `__proto__` directly or indirectly refer to Object.prototype.

Object.create()

- It is a static method on Object prototype, which helps use create an object that has a certain prototype.

```
let newObject = Object.create(null)// here null means no __proto__
```

- `newObject.__proto__` will point to undefined

```
let oldObject = {
    sayHi:function(){
        console.log("Hi");
```

```
    }
}

let newObject = Object.create(oldObject);

console.log(newObject.sayHi());
```

- Here `newObject.__proto__` refers to `oldObject` so when `newObject.sayHi()` was called, it was first searched inside `newObject` and then inside `oldObject` where it was found, and Hi got printed.
- We can implement and use our own `Object.create()` and use it like this

```
function createObject(prototype){
    let newObject = {};
    newObject.__proto__ = prototype;
    return newObject;
}

let oldObject = {
    sayHi: function (){
        console.log("HI");
    }
}

let newObject = createObject(oldObject);
console.log(newObject.sayHi()); //Hi
```

Inheritance

- Inheritance is a way of not writing the same code again and again.
- If there are methods on a prototype of a constructor that already exists, and we need those same methods in another constructor, then there is no need to write these methods again.
- A better approach would to put the already existing prototype in our prototype chain.
- Here is an example

```
function Person(firstName,lastName){
    this.firstName = firstName;
    this.lastName = lastName;
}

Person.prototype.breath = function (){
    console.log(`${this.firstName}`)
}
```

```
function Programmer(firstName, lastName, language){
    this.firstName = firstName;
    this.lastName = lastName;
    this.language = language;
}

Programmer.prototype.code = function(){
    console.log(`${this.firstName} is coding in ${this.language}`);
}

Programmer.prototype.__proto__ = Person.prototype;

let rajeev = new Programmer("Rajeev","Pandey","JavaScript");

console.log(rajeev.code()); //Rajeev is coding in JavaScript
console.log(rajeev.breath()); //Rajeev is breathing
```

- In the above example
  - We created a constructor `Person` that creates a Person object with first name and last name
  - Then we added the **`breath`** function to the prototype of `Person` constructor.
  - Then we created a constructor `Programmer`
  - Added a function **`code`** to its prototype
  - Now the following code sets the `__proto__` of `Programmer.prototype` to `Person.prototype`, that means if a method is not found in `Programmer.prototype`, search it in `Programmer.prototype`

```
Programmer.prototype.__proto__ = Person.prototype;
```

  - We can also write it as

```
Programmer.prototype.__proto__ = Object.create(Person.prototype);
```

- In the above example we used prototypal inheritance
- It is called prototypal because prototypes are used to implement this kind of inheritance.
- It is different from classical inheritance in the sense that object are not created from some ‘blueprint’ rather they point to a prototype, if a method or property is not present in the current object it is looked up in the prototype.

- I hope you learnt something new after reading this. For any feedback or suggestion you can contact me here:

Rajeev Pandey - Freelance Web Developer - Freelance (Self employed) | LinkedIn

View Rajeev Pandey's profile on LinkedIn, the world's largest professional community. Rajeev has 1 job listed on their profile. See the complete profile on LinkedIn and discover Rajeev's connections and jobs at similar companies.

https://www.linkedin.com/in/rajeev-pandey-991878229/