Dziedziczenie i łańcuch prototypów

To tłumaczenie jest niepełne. Pomóż przetłumaczyć ten artykuł z języka angielskiego

JavaScript bywa mylący dla developerów doświadczonych w językach opartych na klasach (jak Java lub C++) przez to, że jest dynamiczny i nie zapewnia implementacji klasy per se (słowo kluczowe `class` zostało wprowadzone w ES2015, ale to tylko lukier składniowy, JavaScript pozostaje oparty na prototypach).

Jeżeli chodzi o dziedziczenie, JavaScript posiada tylko jedną konstrukcję: obiekty. Każdy obiekt posiada prywatną własność łączącą go z innym obiektem zwanym jego **prototypem**. Obiekt *prototype* posiada swój własny prototyp, i tak dalej aż obiekt osiągnie `null` jako swój prototyp. `null` nie ma prototypu i działa jak zakończenie **łańcucha prototypów**.

Podczas gdy jest to często uważane za słabość języka JavaScript, prototypowe podejście do dziedziczenia jest w rzeczywistości znacznie potężniejszym narzędziem niż model klasowy. Dla przykładu trywialnie proste jest zbudowanie klas w modelu prototypowym, podczas gdy odwrotna operacja jest znacznie bardziej skomplikowana.

Dziedziczenie z łańcucha prototypów

Dziedziczenie właściwości

Obiekty w JavaScript są dynamicznymi "workami" właściwości, nazywanych **własnymi właściwościami** (*own properties*). Obiekty JavaScript mają połączenie z obiektem prototypu. Podczas próby dostępu do właściwości obiektu, właściwość będzie szukana nie tylko w samym obiekcie, ale też w jego prototypie, prototypie jego prototypu i tak dalej, aż do odnalezienia właściwości o pasującej nazwie bądź końca łańcucha prototypów.

> W standardzie ECMAScript, notacja `jakisObiekt.[[Prototype]]` jest używana do oznaczenia prototypu `jakisObiekt.` Jest to odpowiednik właściwości `__proto__` (przestarzały).
>
> Nie powinno być to mylone z właściwością *`func`*`.prototype` funkcji, który zamiast tego oznacza `[[Prototype]]` który jest przypisany do wszystkich instancji stworzonych z danej funkcji kiedy jest użyta jako konstruktor . Od ECMAScript 6 do obiektu `[[Prototype]]` można uzyskać dostęp przez `Object.getPrototypeOf()` i `Object.setPrototypeOf()`.

Oto co się dzieje kiedy próbujemy uzyskać dostęp do właściwości:

```
// Przyjmijmy istnienie obiektu o, z własnymi właściwościami a i
// {a: 1, b: 2}
// o.[[Prototype]] ma właściwości b i c:
// {b: 3, c: 4}
// o.[[Prototype]].[[Prototype]] jest nullem.
// Oznacza to koniec łańcucha prototypów,
// zgodnie z definicją null nie ma [[Prototype]]
// A więc pełny łańcuch prototypów wygląda tak:
// {a:1, b:2} ---> {b:3, c:4} ---> null

console.log(o.a); // 1
// Czy istnieje własna właściwość 'a' obiektu o? Tak, jej wartoś

console.log(o.b); // 2
// Czy istnieje własna właściwość 'b' obiektu o? Tak, jej wartoś
// Prototyp także ma właściwość 'b', ale nie jest ona brana pod u
// Nazywa się to "zakrywaniem właściwości" (ang. property shadow

console.log(o.c); // 4
// Czy istnieje właściwość 'c' obiektu o? Nie, sprawdź prototyp.
// Czy istnieje właściwość 'c' obiektu o.[[Prototype]]? Tak, jej

console.log(o.d); // undefined
// Czy istnieje właściwość 'd' obiektu o? Nie, sprawdź prototyp.
// Czy istnieje właściwość 'd' obiektu o.[[Prototype]]? Nie, spr
// o.[[Prototype]].[[Prototype]] to null, koniec wyszukiwania.
// Nie znaleziono właściwości, zwróć undefined.
```

Ustawienie właściwości obiektu tworzy własną właściwość. Jedyny wyjątek od reguł pobierania i tworzenia właściwości stanowi przypadek gdy istnieje odziedziczona właściwość z getterem lub setterem.

Dziedziczenie "metody"

JavaScript nie ma "metod" w rozumieniu języków obiektowych. W JS każda funkcja może być dodana jako właściwość do obiektu. Odziedziczona funkcja zachowuje się jak każda inna właściwość, wliczając w to zakrywanie właściwości, tak jak pokazano wyżej (w tym wypadku forma *nadpisania metody*).

Kiedy jest wykonywana odziedziczona metoda, wartość `this` wskazuje na obiekt, który dziedziczy, nie na obiekt w którym ta metoda została zadeklarowana jako własna właściwość

```
var o = {
  a: 2,
  m: function(b){
    return this.a + 1;
  }
};

console.log(o.m()); // 3
// Podczas wywołania o.m, 'this' wskazuje na o

var p = Object.create(o);
// p jest obiektem dziedziczącym z o

p.a = 12; // tworzy własną właściwość 'a' w obiekcie p
console.log(p.m()); // 13
// Podczas wywołania p.m, 'this' wskazuje na p.
// p dziedziczy funkcję m z obiektu o.
// 'this.a' oznacza p.a, własną właściwość 'a' obiektu p.
```

Różne sposoby tworzenia obiektów i powiązane z nimi łańcuchy prototypów.

Obiekty stworzone za pomocą podstawowej składni

```
var o = {a: 1};
// Nowo stworzony obiekt używa Object.prototype jako swojego [[P
// o nie posiada właściwości o nazwie 'hasOwnProperty'
// hasOwnProperty jest własną właściwością Object.prototype.
// o dziedziczy hasOwnProperty z Object.prototype
// Object.prototype ma null jako swój prototyp.
// o ---> Object.prototype ---> null

var a = ["yo", "whadup", "?"];
// Tablice dziedziczą z Array.prototype
// (który zawiera metody takie jak indexOf, forEach, itd.)
// Łańcuch prototypów wygląda następująco:
// a ---> Array.prototype ---> Object.prototype ---> null

function f(){
  return 2;
}

// Funkcje dziedziczą z Function.prototype
// (który zawiera metody takie jak call, bind, itd.)
// f ---> Function.prototype ---> Object.prototype ---> null
```

Za pomocą konstruktora

"Konstruktor" w JavaScript jest "tylko" funkcją, której używa się w połączeniu z operatorem new.

```
function Graph() {
  this.vertices = [];
  this.edges = [];
}

Graph.prototype = {
  addVertex: function(v){
    this.vertices.push(v);
  }
};

```

```
var g = new Graph();
// g is an object with own properties 'vertices' and 'edges'.
// g.[[Prototype]] is the value of Graph.prototype when new Graph
```

Za pomocą `Object.create`

ECMAScript 5 wprowadził nową metodę: `Object.create()`. Wywołanie tej metody tworzy nowy obiekt. Jego prototypem staje się pierwszy argument tej metody:

```
var a = {a: 1};
// a ---> Object.prototype ---> null

var b = Object.create(a);
// b ---> a ---> Object.prototype ---> null
console.log(b.a); // 1 (inherited)

var c = Object.create(b);
// c ---> b ---> a ---> Object.prototype ---> null

var d = Object.create(null);
// d ---> null
console.log(d.hasOwnProperty);
// undefined, because d doesn't inherit from Object.prototype
```

Za pomocą słowa kluczowego `class`

ECMAScript 6 wprowadził zestaw nowych słów kluczowych do implementacji klas. Mimo, że konstrukcje te mogą wydawać się znajome programistom języków opartych na klasach, nie są one tym samym. JavaScript wciąż opiera się na prototypach. Nowe słowa kluczowe to `class`, `constructor`, `static`, `extends` oraz `super`.

```
"use strict";

class Polygon {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
}
```

```
 9
10  class Square extends Polygon {
11    constructor(sideLength) {
12      super(sideLength, sideLength);
13    }
14    get area() {
15      return this.height * this.width;
16    }
17    set sideLength(newLength) {
18      this.height = newLength;
19      this.width = newLength;
20    }
21  }
22
23  var square = new Square(2);
```

Wydajność

Czas dostępu do właściwości znajdujących się wysoko w łańcuchu prototypów może negatywnie wpływać na wydajność, co może mieć znaczenie w przypadku kodu, którego szybkość wykonania jest krytyczna. W dodatku próba dostępu do nieistniejącej właściwości zawsze powoduje przeszukanie pełnego łańcucha prototypów.

Kiedy iterujemy po właściwościach obiektu, sięgamy do **każdej** właściwości widocznej w łańcuchu prototypów.

Aby sprawdzić czy obiekt ma właściwość zdefiniowaną na nim samym, a nie gdzieś w łańcuchu prototypów, konieczne jest użycie metody `hasOwnProperty`, którą wszystkie obiekty dziedziczą z `Object.prototype`.

`hasOwnProperty` jest w JavaScript jedyną rzeczą, która działa na właściwościach obiektu **nie** przeszukując łańcucha prototypów.

Uwaga: sprawdzenie czy właściwość jest `undefined` **nie** wystarczy. Właściwość może istnieć, a jedynie mieć akurat wartość ustawioną na `undefined`.

Zła praktyka: Rozszerzanie natywnych prototypów

One mis-feature that is often used is to extend `Object.prototype` or one of the other built-in prototypes.

This technique is called monkey patching and breaks *encapsulation*. While used by popular frameworks such as Prototype.js, there is still no good reason for cluttering built-in types with additional *non-standard* functionality.

The **only** good reason for extending a built-in prototype is to backport the features of newer JavaScript engines; for example `Array.forEach`, etc.

Przykład

`B` shall inherit from `A`:

```
function A(a){
  this.varA = a;
}

// What is the purpose of including varA in the prototype when A.prot
// this.varA, given the definition of function A above?
A.prototype = {
  varA : null,  // Shouldn't we strike varA from the prototype as doi
      // perhaps intended as an optimization to allocate space in hid
      // https://developers.google.com/speed/articles/optimizing-java
      // would be valid if varA wasn't being initialized uniquely for
  doSomething : function(){
    // ...
  }
};

function B(a, b){
  A.call(this, a);
  this.varB = b;
}
B.prototype = Object.create(A.prototype, {
  varB : {
    value: null,
    enumerable: true,
    configurable: true,
    writable: true
  },
  doSomething : {
```

```
    value: function(){ // override
      A.prototype.doSomething.apply(this, arguments); // call super
      // ...
    },
    enumerable: true,
    configurable: true,
    writable: true
  }
});
B.prototype.constructor = B;

var b = new B();
b.doSomething();
```

The important parts are:

- Types are defined in `.prototype`
- You use `Object.create()` to inherit

`prototype` and `Object.getPrototypeOf`

JavaScript is a bit confusing for developers coming from Java or C++, as it's all dynamic, all runtime, and it has no classes at all. It's all just instances (objects). Even the "classes" we simulate are just a function object.

You probably already noticed that our `function A` has a special property called `prototype`. This special property works with the JavaScript `new` operator. The reference to the prototype object is copied to the internal `[[Prototype]]` property of the new instance. For example, when you do `var a1 = new A()`, JavaScript (after creating the object in memory and before running function `A()` with `this` defined to it) sets `a1.[[Prototype]] = A.prototype`. When you then access properties of the instance, JavaScript first checks whether they exist on that object directly, and if not, it looks in `[[Prototype]]`. This means that all the stuff you define in `prototype` is effectively shared by all instances, and you can even later change parts of `prototype` and have the changes appear in all existing instances, if you wanted to.

If, in the example above, you do `var a1 = new A(); var a2 = new A();` then `a1.doSomething` would actually refer to `Object.getPrototypeOf(a1).doSomething`, which is the same as the `A.prototype.doSomething` you defined, i.e.

`Object.getPrototypeOf(a1).doSomething == Object.getPrototypeOf(a2).doSomething == A.prototype.doSomething`.

In short, `prototype` is for types, while `Object.getPrototypeOf()` is the same for instances.

`[[Prototype]]` is looked at *recursively*, i.e. `a1.doSomething`, `Object.getPrototypeOf(a1).doSomething`, `Object.getPrototypeOf(Object.getPrototypeOf(a1)).doSomething` etc., until it's found or `Object.getPrototypeOf` returns null.

So, when you call

```
var o = new Foo();
```

JavaScript actually just does

```
var o = new Object();
o.[[Prototype]] = Foo.prototype;
Foo.call(o);
```

(or something like that) and when you later do

```
o.someProp;
```

it checks whether `o` has a property `someProp`. If not it checks `Object.getPrototypeOf(o).someProp` and if that doesn't exist it checks `Object.getPrototypeOf(Object.getPrototypeOf(o)).someProp` and so on.

Na zakończenie

It is **essential** to understand the prototypal inheritance model before writing complex code that makes use of it. Also, be aware of the length of the prototype chains in your code and break them up if necessary to avoid possible performance problems. Further, the native prototypes

should **never** be extended unless it is for the sake of compatibility with newer JavaScript features.

Ostatnia modyfikacja: 25 cze 2019, autorzy MDN

Powiązane tematy

JavaScript

Tutorials:

- Complete beginners
- JavaScript Guide
- Intermediate
- Advanced

  Inheritance and the prototype chain

  Strict mode

  JavaScript typed arrays

  Memory Management

  Concurrency model and Event Loop

References:

- Built-in objects
- Expressions & operators
- Statements & declarations
- Functions
- Classes
- Errors
- Misc