

Раздел

[Прототипы, наследование](#)

Навигация по уроку

[[Prototype]]

Операция записи не использует прототип

Значение «this»


Цикл for...in

Итого

Задачи (4)

Комментарии

Поделиться

[Редактировать на GitHub](#)[🏠 → Язык программирования JavaScript](#)  
[→ Прототипы, наследование](#) 13-го марта 2020

## Прототипное наследование

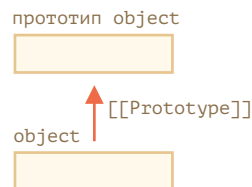
В программировании мы часто хотим взять что-то и расширить.

Например, у нас есть объект `user` со своими свойствами и методами, и мы хотим создать объекты `admin` и `guest` как его слегка изменённые варианты. Мы хотели бы повторно использовать то, что есть у объекта `user`, не копировать/переопределять его методы, а просто создать новый объект на его основе.

*Прототипное наследование* — это возможность языка, которая помогает в этом.

### [[Prototype]]

В JavaScript объекты имеют специальное скрытое свойство `[[Prototype]]` (так оно названо в спецификации), которое либо равно `null`, либо ссылается на другой объект. Этот объект называется «прототип»:



Прототип даёт нам немного «магии». Когда мы хотим прочитать свойство из `object`, а оно отсутствует, JavaScript автоматически берёт его из прототипа. В программировании такой механизм называется «прототипным наследованием». Многие интересные возможности языка и техники программирования основываются на нём.



Свойство `[[Prototype]]` является внутренним и скрытым, но есть много способов задать его.

Одним из них является использование `__proto__`, например так:

```
1 let animal = {
2   eats: true
3 };
4 let rabbit = {
5   jumps: true
6 };
7
8 rabbit.__proto__ = animal;
```



Раздел

[Прототипы, наследование](#)

Навигация по уроку

[[Prototype]]

Операция записи не использует прототип

Значение «this»

Цикл for...in

Итого

Задачи (4)

Комментарии

Поделиться



[Редактировать на GitHub](#)



### **❗ Свойство `__proto__` — исторически обусловленный геттер/сеттер для `[[Prototype]]`**

Обратите внимание, что `__proto__` — не то же самое, что `[[Prototype]]`. Это геттер/сеттер для него.

Он существует по историческим причинам, в современном языке его заменяют функции

`Object.getPrototypeOf/Object.setPrototypeOf`, которые также получают/устанавливают прототип. Мы рассмотрим причины этого и сами функции позже.

По спецификации `__proto__` должен поддерживаться только браузерами, но по факту все среды, включая серверную, поддерживают его. Далее мы будем в примерах использовать `__proto__`, так как это самый короткий и интуитивно понятный способ установки и чтения прототипа.

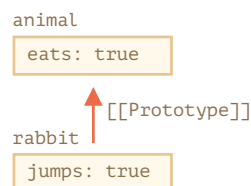
Если мы ищем свойство в `rabbit`, а оно отсутствует, JavaScript автоматически берёт его из `animal`.

Например:

```
1 let animal = {
2   eats: true
3 };
4 let rabbit = {
5   jumps: true
6 };
7
8 rabbit.__proto__ = animal; // (*)
9
10 // теперь мы можем найти оба свойства в rabbit:
11 alert( rabbit.eats ); // true (**)
12 alert( rabbit.jumps ); // true
```

Здесь строка (\*) устанавливает `animal` как прототип для `rabbit`.

Затем, когда `alert` пытается прочесть свойство `rabbit.eats` (\*\*), его нет в `rabbit`, поэтому JavaScript следует по ссылке `[[Prototype]]` и находит его в `animal` (смотрите снизу вверх):



Здесь мы можем сказать, что "animal является прототипом rabbit" или "rabbit прототипно наследует от animal".

Так что если у `animal` много полезных свойств и методов, то они автоматически становятся доступными у `rabbit`. Такие свойства называются «унаследованными».

Если у нас есть метод в `animal`, он может быть вызван на `rabbit`:

```
1 let animal = {
2   eats: true,
3   walk() {
4     alert("Animal walk");
5   }
6 };
7
8 let rabbit = {
9   jumps: true,
10  __proto__: animal
11 };
```

Раздел

[Прототипы, наследование](#)

Навигация по уроку

[[Prototype]]

Операция записи не использует прототип

Значение «this»

Цикл for...in

Итого

Задачи (4)

Комментарии

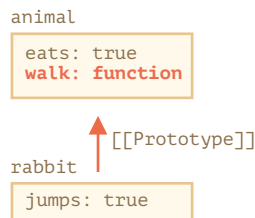
Поделиться



[Редактировать на GitHub](#)

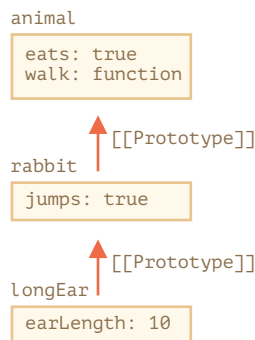
```
12
13 // walk взят из прототипа
14 rabbit.walk(); // Animal walk
```

Метод автоматически берётся из прототипа:



Цепочка прототипов может быть длиннее:

```
1 let animal = {
2   eats: true,
3   walk() {
4     alert("Animal walk");
5   }
6 };
7
8 let rabbit = {
9   jumps: true,
10  __proto__: animal
11 };
12
13 let longEar = {
14   earLength: 10,
15   __proto__: rabbit
16 };
17
18 // walk взят из цепочки прототипов
19 longEar.walk(); // Animal walk
20 alert(longEar.jumps); // true (из rabbit)
```



Есть только два ограничения:

1. Ссылки не могут идти по кругу. JavaScript выдаст ошибку, если мы попытаемся назначить `__proto__` по кругу.
2. Значение `__proto__` может быть объектом или `null`. Другие типы игнорируются.

Это вполне очевидно, но всё же: может быть только один `[[Prototype]]`. Объект не может наследоваться от двух других объектов.

## Операция записи не использует прототип

Прототип используется только для чтения свойств.

Операции записи/удаления работают напрямую с объектом.

В приведённом ниже примере мы присваиваем `rabbit` собственный метод `walk`:

Раздел

[Прототипы, наследование](#)

Навигация по уроку

[[Prototype]]

Операция записи не использует прототип

Значение «this»

Цикл for...in

Итого

Задачи (4)

Комментарии

Поделиться

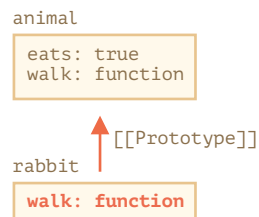


[Редактировать на GitHub](#)



```
1 let animal = {
2   eats: true,
3   walk() {
4     /* этот метод не будет использоваться в rabbit */
5   }
6 };
7
8 let rabbit = {
9   __proto__: animal
10 };
11
12 rabbit.walk = function() {
13   alert("Rabbit! Bounce-bounce!");
14 };
15
16 rabbit.walk(); // Rabbit! Bounce-bounce!
```

Теперь вызов `rabbit.walk()` находит метод непосредственно в объекте и выполняет его, не используя прототип:



Свойства-аксессоры – исключение, так как запись в него обрабатывается функцией-сеттером. То есть, это, фактически, вызов функции.

По этой причине `admin.fullName` работает корректно в приведённом ниже коде:



```
1 let user = {
2   name: "John",
3   surname: "Smith",
4
5   set fullName(value) {
6     [this.name, this.surname] = value.split(" ");
7   },
8
9   get fullName() {
10    return `${this.name} ${this.surname}`;
11  }
12 };
13
14 let admin = {
15   __proto__: user,
16   isAdmin: true
17 };
18
19 alert(admin.fullName); // John Smith (*)
20
21 // срабатывает сеттер!
22 admin.fullName = "Alice Cooper"; // (**)
23 alert(admin.name); // Alice
24 alert(admin.surname); // Cooper
```



Здесь в строке (\*) свойство `admin.fullName` имеет геттер в прототипе `user`, поэтому вызывается он. В строке (\*\*) свойство также имеет сеттер в прототипе, который и будет вызван.

## Значение «this»

В приведённом выше примере может возникнуть интересный вопрос: каково значение `this` внутри `set fullName(value)`? Куда записаны свойства `this.name` и `this.surname`: в `user` или в `admin`?

Раздел

[Прототипы, наследование](#)

Навигация по уроку

[[Prototype]]

Операция записи не использует прототип

Значение «this»

Цикл for...in

Итого

Задачи (4)

Комментарии

Поделиться



[Редактировать на GitHub](#)



Ответ прост: прототипы никак не влияют на `this`.

**Неважно, где находится метод: в объекте или его прототипе. При вызове метода `this` — всегда объект перед точкой.**

Таким образом, вызов сеттера `admin.fullName=` в качестве `this` использует `admin`, а не `user`.

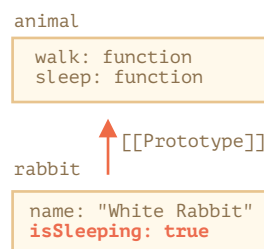
Это на самом деле очень важная деталь, потому что у нас может быть большой объект со множеством методов, от которого можно наследовать. Затем наследующие объекты могут вызывать его методы, но они будут изменять своё состояние, а не состояние объекта-родителя.

Например, здесь `animal` представляет собой «хранилище методов», и `rabbit` использует его.

Вызов `rabbit.sleep()` устанавливает `this.isSleeping` для объекта `rabbit`:

```
1 // методы animal
2 let animal = {
3   walk() {
4     if (!this.isSleeping) {
5       alert('I walk');
6     }
7   },
8   sleep() {
9     this.isSleeping = true;
10  }
11 };
12
13 let rabbit = {
14   name: "White Rabbit",
15   __proto__: animal
16 };
17
18 // модифицирует rabbit.isSleeping
19 rabbit.sleep();
20
21 alert(rabbit.isSleeping); // true
22 alert(animal.isSleeping); // undefined (нет такого свой
```

Картинка с результатом:



Если бы у нас были другие объекты, такие как `bird`, `snake` и т.д., унаследованные от `animal`, они также получили бы доступ к методам `animal`. Но `this` при вызове каждого метода будет соответствовать объекту (перед точкой), на котором происходит вызов, а не `animal`. Поэтому, когда мы записываем данные в `this`, они сохраняются в этих объектах.

В результате методы являются общими, а состояние объекта — нет.

## Цикл for...in

Цикл `for...in` проходит не только по собственным, но и по унаследованным свойствам объекта.

Например:

```
1 let animal = {
2   eats: true
```

Раздел

[Прототипы, наследование](#)

Навигация по уроку

[[Prototype]]

Операция записи не использует прототип

Значение «this»

Цикл for...in

Итого

Задачи (4)

Комментарии

Поделиться



[Редактировать на GitHub](#)



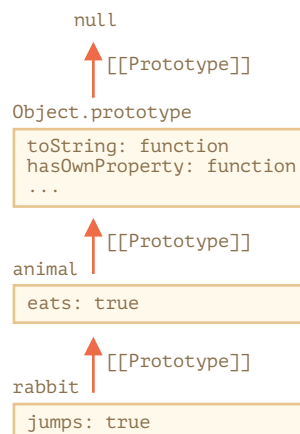
```
3 };
4
5 let rabbit = {
6   jumps: true,
7   __proto__: animal
8 };
9
10 // Object.keys возвращает только собственные ключи
11 alert(Object.keys(rabbit)); // jumps
12
13 // for..in проходит и по своим, и по унаследованным ключам
14 for(let prop in rabbit) alert(prop); // jumps, затем ea
```

Если унаследованные свойства нам не нужны, то мы можем отфильтровать их при помощи встроенного метода `obj.hasOwnProperty(key)`: он возвращает `true`, если у `obj` есть собственное, не унаследованное, свойство с именем `key`.

Пример такой фильтрации:

```
1 let animal = {
2   eats: true
3 };
4
5 let rabbit = {
6   jumps: true,
7   __proto__: animal
8 };
9
10 for(let prop in rabbit) {
11   let isOwn = rabbit.hasOwnProperty(prop);
12
13   if (isOwn) {
14     alert(`Our: ${prop}`); // Our: jumps
15   } else {
16     alert(`Inherited: ${prop}`); // Inherited: eats
17   }
18 }
```

В этом примере цепочка наследования выглядит так: `rabbit` наследует от `animal`, который наследует от `Object.prototype` (так как `animal` – литеральный объект `{...}`, то это по умолчанию), а затем `null` на самом верху:



Заметим ещё одну деталь. Откуда взялся метод `rabbit.hasOwnProperty`? Мы его явно не определяли. Если посмотреть на цепочку прототипов, то видно, что он берётся из `Object.prototype.hasOwnProperty`. То есть, он унаследован.

...Но почему `hasOwnProperty` не появляется в цикле `for...in` в отличие от `eats` и `jumps`? Он ведь перечисляет все унаследованные свойства.

Ответ простой: оно не перечислимо. То есть, у него внутренний флаг `enumerable` стоит `false`, как и у других свойств `Object.prototype`.

Поэтому оно и не появляется в цикле.

### Почти все остальные методы получения ключей/значений игнорируют унаследованные свойства

Почти все остальные методы, получающие ключи/значения, такие как `Object.keys`, `Object.values` и другие – игнорируют унаследованные свойства.

Они учитывают только свойства самого объекта, не его прототипа.

## Итого

- В JavaScript все объекты имеют скрытое свойство `[[Prototype]]`, которое является либо другим объектом, либо `null`.
- Мы можем использовать `obj.__proto__` для доступа к нему (исторически обусловленный геттер/сеттер, есть другие способы, которые скоро будут рассмотрены).
- Объект, на который ссылается `[[Prototype]]`, называется «прототипом».
- Если мы хотим прочитать свойство `obj` или вызвать метод, которого не существует у `obj`, тогда JavaScript попытается найти его в прототипе.
- Операции записи/удаления работают непосредственно с объектом, они не используют прототип (если это обычное свойство, а не сеттер).
- Если мы вызываем `obj.method()`, а метод при этом взят из прототипа, то `this` всё равно ссылается на `obj`. Таким образом, методы всегда работают с текущим объектом, даже если они наследуются.
- Цикл `for...in` перебирает как свои, так и унаследованные свойства. Остальные методы получения ключей/значений работают только с собственными свойствами объекта.

## Задачи

### Работа с прототипами

важность: 5

В приведённом ниже коде создаются и изменяются два объекта.

Какие значения показываются в процессе выполнения кода?

```
1 let animal = {
2   jumps: null
3 };
4 let rabbit = {
5   __proto__: animal,
6   jumps: true
7 };
8
9 alert( rabbit.jumps ); // ? (1)
10
11 delete rabbit.jumps;
12
13 alert( rabbit.jumps ); // ? (2)
14
15 delete animal.jumps;
16
17 alert( rabbit.jumps ); // ? (3)
```

Должно быть 3 ответа.

решение

### Алгоритм поиска

важность: 5

Раздел

[Прототипы, наследование](#)

Навигация по уроку

`[[Prototype]]`

Операция записи не использует прототип

Значение «this»

Цикл `for...in`

Итого

Задачи (4)

Комментарии

Поделиться



Редактировать на GitHub

Задача состоит из двух частей.

У нас есть объекты:

```
1 let head = {
2   glasses: 1
3 };
4
5 let table = {
6   pen: 3
7 };
8
9 let bed = {
10  sheet: 1,
11  pillow: 2
12 };
13
14 let pockets = {
15   money: 2000
16 };
```

1. С помощью свойства `__proto__` задайте прототипы так, чтобы поиск любого свойства выполнялся по следующему пути: `pockets` → `bed` → `table` → `head`. Например, `pockets.pen` должно возвращать значение 3 (найденное в `table`), а `bed.glasses` – значение 1 (найденное в `head`).

2. Ответьте на вопрос: как быстрее получить значение `glasses` – через `pockets.glasses` или через `head.glasses`? При необходимости составьте цепочки поиска и сравните их.

решение

## Куда будет произведена запись?

важность: 5

Объект `rabbit` наследует от объекта `animal`.

Какой объект получит свойство `full` при вызове `rabbit.eat()`: `animal` или `rabbit`?

```
1 let animal = {
2   eat() {
3     this.full = true;
4   }
5 };
6
7 let rabbit = {
8   __proto__: animal
9 };
10
11 rabbit.eat();
```

решение

## Почему наедаются оба хомяка?

важность: 5

У нас есть два хомяка: шустрый (`speedy`) и ленивый (`lazy`); оба наследуют от общего объекта `hamster`.

Когда мы кормим одного хомяка, второй тоже наедается. Почему? Как это исправить?

```
1 let hamster = {
2   stomach: [],
3 }
```



Раздел

[Прототипы, наследование](#)

Навигация по уроку

[\[\[Prototype\]\]](#)

Операция записи не использует прототип

Значение «this»

Цикл `for...in`

Итого

Задачи (4)

Комментарии

Поделиться



[Редактировать на GitHub](#)





Раздел

[Прототипы, наследование](#)

Навигация по уроку

[[Prototype]]

Операция записи не  
использует прототип

Значение «this»

Цикл for...in

Итого

Задачи (4)

Комментарии

Поделиться



[Редактировать на GitHub](#)



```
4   eat(food) {
5       this.stomach.push(food);
6   }
7 };
8
9   let speedy = {
10     __proto__: hamster
11 };
12
13   let lazy = {
14     __proto__: hamster
15 };
16
17   // Этот хомяк нашёл еду
18   speedy.eat("apple");
19   alert( speedy.stomach ); // apple
20
21   // У этого хомяка тоже есть еда. Почему? Исправьте
22   alert( lazy.stomach ); // apple
```

решение

Проводим [курсы по JavaScript и фреймворкам](#).



## Комментарии

перед тем как писать...