Методы прототипов, объекты без свойства __proto__

В первой главе этого раздела мы упоминали, что существуют современные методы работы с прототипами.

Свойство `__proto__` считается устаревшим, и по стандарту оно должно поддерживаться только браузерами.

Современные же методы это:

- Object.create(proto, [descriptors]) – создаёт пустой объект со свойством `[[Prototype]]`, указанным как `proto`, и необязательными дескрипторами свойств `descriptors`.
- Object.getPrototypeOf(obj) – возвращает свойство `[[Prototype]]` объекта `obj`.
- Object.setPrototypeOf(obj, proto) – устанавливает свойство `[[Prototype]]` объекта `obj` как `proto`.

Эти методы нужно использовать вместо `__proto__`.

Например:

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

// создаём новый объект с прототипом animal
let rabbit = Object.create(animal);

alert(rabbit.eats); // true

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

Object.setPrototypeOf(rabbit, {}); // заменяем прототип
```

У `Object.create` есть необязательный второй аргумент: дескрипторы свойств. Мы можем добавить дополнительное свойство новому объекту таким образом:

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

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

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

Формат задания дескрипторов описан в главе Флаги и дескрипторы свойств.

Мы также можем использовать `Object.create` для «продвинутого» клонирования объекта, более мощного, чем копирование свойств в цикле `for..in`:

```
// клон obj с тем же прототипом (с поверхностным копиро
```

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


Такой вызов создаёт точную копию объекта `obj`, включая все свойства: перечисляемые и неперечисляемые, геттеры/сеттеры для свойств – и всё это с правильным свойством `[[Prototype]]`.

Краткая история

Если пересчитать все способы управления прототипом, то их будет много! И многие из них делают одно и то же!

Почему так?

В силу исторических причин.

- Свойство `"prototype"` функции-конструктора существует с совсем давних времён.
- Позднее, в 2012 году, в стандарте появился метод `Object.create`. Это давало возможность создавать объекты с указанным прототипом, но не позволяло устанавливать/получать его. Тогда браузеры реализовали нестандартный аксессор `__proto__`, который позволил устанавливать/получать прототип в любое время.
- Позднее, в 2015 году, в стандарт были добавлены `Object.setPrototypeOf` и `Object.getPrototypeOf`, заменяющие собой аксессор `__proto__`, который упоминается в Приложении Б стандарта, которое не обязательно к поддержке в небраузерных окружениях. При этом де-факто `__proto__` всё ещё поддерживается везде.

В итоге сейчас у нас есть все эти способы для работы с прототипом.

Почему же `__proto__` был заменён на функции `getPrototypeOf/setPrototypeOf`? Читайте далее, чтобы узнать ответ.

> ⚠ **Не меняйте `[[Prototype]]` существующих объектов, если важна скорость**
>
> Технически мы можем установить/получить `[[Prototype]]` в любое время. Но обычно мы устанавливаем прототип только раз во время создания объекта, а после не меняем: `rabbit` наследует от `animal`, и это не изменится.
>
> И JavaScript движки хорошо оптимизированы для этого. Изменение прототипа «на лету» с помощью `Object.setPrototypeOf` или `obj.__proto__=` – очень медленная операция, которая ломает внутренние оптимизации для операций доступа к свойствам объекта. Так что лучше избегайте этого кроме тех случаев, когда вы знаете, что делаете, или же когда скорость JavaScript для вас не имеет никакого значения.

"Простейший" объект

Как мы знаем, объекты можно использовать как ассоциативные массивы для хранения пар ключ/значение.

...Но если мы попробуем хранить *созданные пользователями* ключи (например, словари с пользовательским вводом), мы можем заметить интересный сбой: все ключи работают как ожидается, за исключением `"__proto__"`.

Посмотрите на пример:

```
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], не "some value"!
```

Если пользователь введёт `__proto__`, присвоение проигнорируется!

И это не должно удивлять нас. Свойство `__proto__` особенное: оно должно быть либо объектом, либо `null`, а строка не может стать прототипом.

Но мы не *намеревались* реализовывать такое поведение, не так ли? Мы хотим хранить пары ключ/значение, и ключ с именем `"__proto__"` не был сохранён надлежащим образом. Так что это ошибка!

Конкретно в этом примере последствия не так ужасны, но если мы присваиваем объектные значения, то прототип и в самом деле может быть изменён. В результате дальнейшее выполнение пойдёт совершенно непредсказуемым образом.

Что хуже всего – разработчики не задумываются о такой возможности совсем. Это делает такие ошибки сложным для отлавливания или даже превращает их в уязвимости, особенно когда JavaScript используется на сервере.

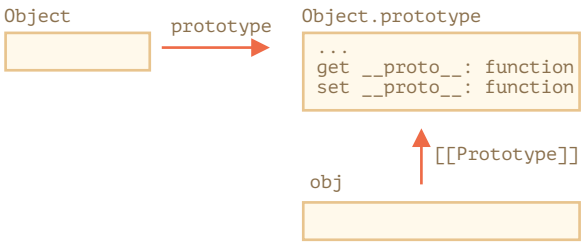
Неожиданные вещи могут случаться также при присвоении свойства `toString`, которое по умолчанию функция, и других свойств, которые тоже на самом деле являются встроенными методами.

Как же избежать проблемы?

Во-первых, мы можем переключиться на использование коллекции `Map`, и тогда всё будет в порядке.

Но и `Object` может также хорошо подойти, потому что создатели языка уже давно продумали решение проблемы.

Свойство `__proto__` – не обычное, а аксессор, заданный в `Object.prototype`:



Так что при чтении или установке `obj.__proto__` вызывается соответствующий геттер/сеттер из прототипа `obj`, и именно он устанавливает/получает свойство `[[Prototype]]`.

Как было сказано в начале этой секции учебника, `__proto__` – это способ доступа к свойству `[[Prototype]]`, это не само свойство `[[Prototype]]`.

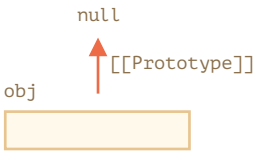
Теперь, если мы хотим использовать объект как ассоциативный массив, мы можем сделать это с помощью небольшого трюка:

```
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)` создаёт пустой объект без прототипа (`[[Prototype]]` будет `null`):



Таким образом не будет унаследованного геттера/сеттера для `__proto__`. Теперь это свойство обрабатывается как обычное свойство, и приведённый выше пример работает правильно.

Мы можем назвать такой объект «простейшим» или «чистым словарным объектом», потому что он ещё проще, чем обычные объекты `{...}`.

Недостаток в том, что у таких объектов не будет встроенных методов объекта, таких как `toString`:

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

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

...Но обычно это нормально для ассоциативных массивов.

Обратите внимание, что большая часть методов, связанных с объектами, имеют вид `Object.something(...)`. К примеру, `Object.keys(obj)`. Подобные методы не находятся в прототипе, так что они продолжат работать для таких объектов:

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

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

Итого

Современные способы установки и прямого доступа к прототипу это:

- Object.create(proto[, descriptors]) – создаёт пустой объект со свойством `[[Prototype]]`, указанным как `proto` (может быть `null`), и необязательными дескрипторами свойств.
- Object.getPrototypeOf(obj) – возвращает свойство `[[Prototype]]` объекта `obj` (то же самое, что и геттер `__proto__`).
- Object.setPrototypeOf(obj, proto) – устанавливает свойство `[[Prototype]]` объекта `obj` как `proto` (то же самое, что и сеттер `__proto__`).

Встроенный геттер/сеттер `__proto__` не безопасен, если мы хотим использовать *созданные пользователями* ключи в объекте. Как минимум потому, что пользователь может ввести `"__proto__"` как ключ, от чего может возникнуть ошибка. Если повезёт – последствия будут лёгкими, но, вообще говоря, они непредсказуемы.

Так что мы можем использовать либо `Object.create(null)` для создания «простейшего» объекта, либо использовать коллекцию `Map`.

Кроме этого, `Object.create` даёт нам лёгкий способ создать поверхностную копию объекта со всеми дескрипторами:

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

Мы также ясно увидели, что `__proto__` – это геттер/сеттер для свойства `[[Prototype]]`, и находится он в `Object.prototype`, как и другие методы.

Мы можем создавать объекты без прототипов с помощью `Object.create(null)`. Такие объекты можно использовать как «чистые словари», у них нет проблем с использованием строки `"__proto__"` в качестве ключа.

Ещё методы:

- Object.keys(obj) / Object.values(obj) / Object.entries(obj) – возвращают массив всех перечисляемых собственных строковых ключей/значений/пар ключ-значение.
- Object.getOwnPropertySymbols(obj) – возвращает массив всех собственных символьных ключей.
- Object.getOwnPropertyNames(obj) – возвращает массив всех собственных строковых ключей.
- Reflect.ownKeys(obj) – возвращает массив всех собственных ключей.

- obj.hasOwnProperty(key): возвращает `true`, если у `obj` есть собственное (не унаследованное) свойство с именем `key`.

Все методы, которые возвращают свойства объектов (такие как `Object.keys` и другие), возвращают «собственные» свойства. Если мы хотим получить и унаследованные, можно воспользоваться циклом `for..in`.

Задачи

Добавьте toString в словарь

важность: 5

Имеется объект `dictionary`, созданный с помощью `Object.create(null)` для хранения любых пар `ключ/значение`.

Добавьте ему метод `dictionary.toString()`, который должен возвращать список ключей, разделённых запятой. Ваш `toString` не должен выводиться при итерации объекта с помощью цикла `for..in`.

Вот так это должно работать:

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

// ваш код, который добавляет метод dictionary.toString

// добавляем немного данных
dictionary.apple = "Apple";
dictionary.__proto__ = "test"; // здесь __proto__ -- эт

// только apple и __proto__ выведены в цикле
for(let key in dictionary) {
  alert(key); // "apple", затем "__proto__"
}

// ваш метод toString в действии
alert(dictionary); // "apple,__proto__"
```

решение

Разница между вызовами

важность: 5

Давайте создадим новый объект `rabbit`:

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

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

Все эти вызовы делают одно и тоже или нет?

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

решение

Комментарии

перед тем как писать...