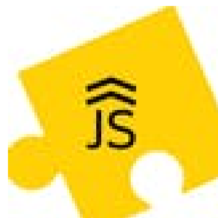
Прототипы

Курс: JavaScript, часть 2: прототипы и асинхронность

20 февраля 2018 г.



Оглавление

1 Прототипы **2**

1.1 Прототипы 2

1.2 Цепочки прототипов 4

1.3 Способы установки прототипов 7

1.3.1 `__proto__` 7

1.3.2 Метод `create` 7

1.3.3 `setPrototypeOf` 8

1.4 Эффект затенения 9

1.5 Поля только для чтения в прототипах 10

1.6 Сеттеры и геттеры в прототипах 12

1.7 Неперечисляемые поля в прототипах 13

Глава 1

Прототипы

1.1. Прототипы

Сформулируем задачу: пусть у нас есть некоторый объект, который олицетворяет студента. Он записан в литеральной нотации и описывает характеристики студента, а также полезные для него действия. Этот объект содержит поле name, которое хранит имя студента и метод `getName`, который возвращает имя нашего студента.

```
var student = {
    name: 'Billy',
    type: 'human',
    getName: function () {
        return this.name;
    },
    sleep: function () {
        console.info('zzzZZZ...');
    }
};

student.getName();
// Billy
```

Объект студента сложно рассматривать в отрыве от объекта преподавателя. У преподавателя также есть ряд полей и методов, например поле `name`, которое хранит имя. Если мы посмотрим на два этих объекта внимательнее, мы уви-

дим, что в них очень много похожего: у каждого из них есть метод `getName`, который выполняет одинаковую работу. Таким образом, мы дублируем реализацию одного и того же метода в двух разных объектах, и это проблема.
К счастью, решение очень простое. Мы можем выделить общие части в отдельную конструкцию.
Назовем объединяющий объект person/личность. В итоге мы получим три несвязанных объекта: студента, преподавателя и личность.

```
var person = {
    type: 'human',
    getName: function () {
        return this.name;
    }
};
```

Так как мы забрали у наших объектов студента и преподавателя полезный метод `getName`, нам необходимо после нашего рефакторинга решить следующую задачу: научить студента пользоваться общим кодом, который мы вынесли в другой объект. Для решения этой задачи мы можем воспользоваться методом заимствования. Для этого мы можем позаимствовать метод `getName` у объекта `person` и вызвать его при помощи метода `call`, передав первым аргументом объект студента.

```
var student = {
    name: 'Billy',
};
var person = {
    getName: function () {
        return this.name;
    }
};
person.getName.call(student);
```

Нам хотелось бы вызывать метод `getName`, как и раньше, от лица студента. Можем ли мы связать два наших объекта студента и `person` таким образом, чтобы это было возможным?
Необходимо лишь в специальное внутреннее поле `[[Prototype]]` одного объекта записать ссылку на другой. Так, мы можем записать в это поле у объекта

student ссылку на объект person и получить желаемое поведение. Обратиться напрямую ко внутреннему полю, конечно, нельзя, но существует ряд способов, которые позволяют записать в него новое значение. Один из них — геттер и сеттер _proto_.

```
var student = {
    name: 'Billy',
    sleep: function () {},
    [[Prototype]]: <link to person>,
};
student['[[Prototype]]'] = person; //так не работает!
```

> Объект, на который указывает ссылка во внутреннем поле [[Prototype]], называется **прототипом**.

1.2. Цепочки прототипов

Что происходит, когда мы пытаемся вызвать метод, которого нет у объекта, но он есть в прототипе? В этом случае интерпретатор переходит по ссылке, которая хранится во внутреннем поле Prototype и пробует найти этот метод в прототипе. В нашем случае мы вызываем метод getName у объекта student, но этого метода у этого объекта нет. Интерпретатор смотрит значения внутреннего поля Prototype, видит там ссылку на прототип — person, и переходит по этой ссылке, пробуя найти этот метод уже в прототипе. Там он этот метод находит и вызывает. Важно заметить, что this при исполнении этого метода будет ссылаться на объект student, так как мы этот метод вызываем от лица студента.

```
var student = {
    name: 'Billy',
    [[Prototype]]: <person>
};
var person = {
    type: 'human',
    getName: function () {
        return this.name;
    }
};
```

Но что произойдет, если мы попытаемся вызвать метод, которого нет не только у объекта, но и в прототипе? Можно заметить, что у прототипа также есть внутреннее поле `Prototype`.

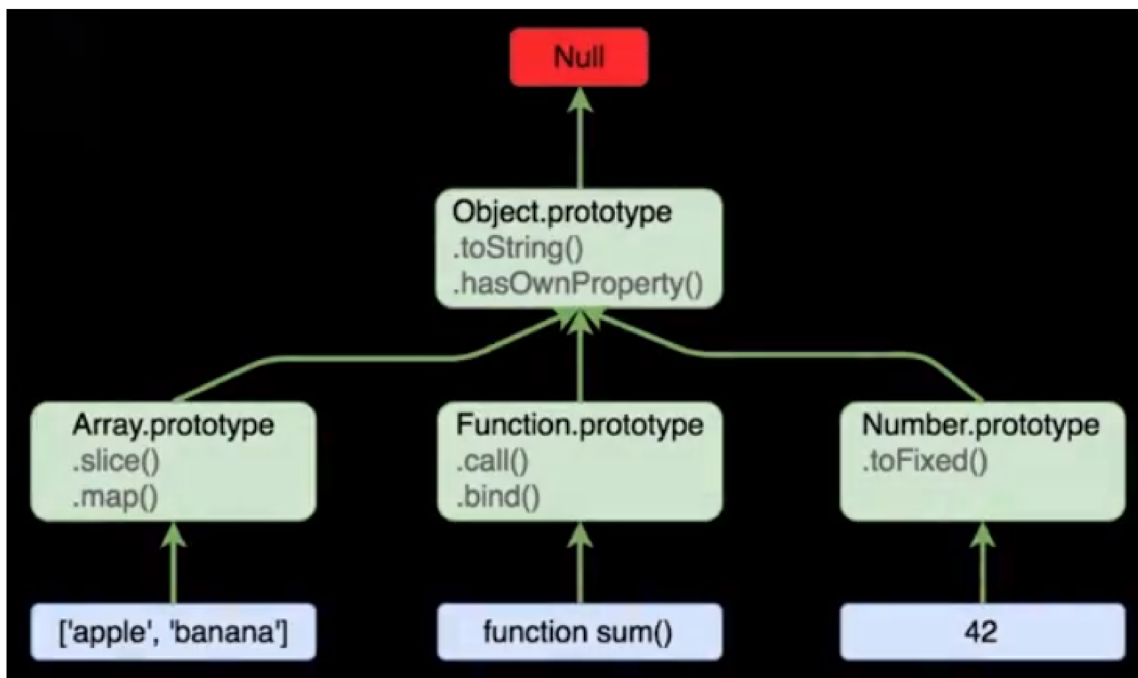
> Интерпретатор идет по выстроенной цепочке прототипов в поисках поля или метода до тех пор, пока не встретит значение `null` в специальном внутреннем поле `Prototype`. Если он прошел весь путь по цепочке, но так и не нашел искомого метода или поля, в этом случае он вернет `undefined`.

В нашем случае может показаться, что поиск остановится уже на объекте `person`, ведь мы специально не записывали никакую ссылку во внутреннее поле `Prototype`. Но любой объект уже имеет в качестве прототипа некоторый глобальный прототип — глобальный прототип для всех объектов. Он расположен в специальном поле `Prototype` функции `Object` и хранит в себе методы, полезные для всех объектов.

```
var student = {
    name: 'Billy',
    [[Prototype]]: <person>
};
var person = {
    type: 'human',
    [[Prototype]]: <Object.prototype>
};
```

Мы выяснили, что в нашем случае поиск метода не остановится на объекте `person` и мы проследуем по цепочке прототипов дальше в глобальный прототип

для всех объектов. И уже там наш поиск остановится, так как внутреннее поле `Prototype` глобального прототипа для всех объектов имеет значение `null`. Помимо общего, глобального прототипа для всех объектов, существуют в языке более частные глобальные прототипы: для массивов, функций, и т.д. Каждый из этих прототипов по умолчанию в качестве прототипа имеет глобальный прототип для всех объектов. И для того, чтобы поиск метода или поля по цепочке прототипа всегда заканчивался, в этом глобальном прототипе в поле `Prototype`, во внутреннем поле, хранится значение `null`.



Давайте попробуем обмануть интерпретатор и в качестве прототипа для преподавателя выбрать студента, а в качестве прототипа для студента выбрать преподавателя. И попробуем вызвать заведомо несуществующий метод или поле. В данном случае может показаться, что поиск будет идти бесконечно, интерпретатор будет вечно ходить по созданному нами циклу в цепочке прототипов. Но интерпретатор позаботился о таком поведении и выбросит ошибку уже на этапе попытки создания такой цепочки.

```
var lecturer = { name: 'Sergey'}
var student = { name: 'Billy' }

lecturer.__proto__ = student;
student.__proto__ = lecturer;

console.info(lecturer.abrakadabra);
```

Uncaught TypeError: Cyclic __proto__ value

Ещё на строчке «student.__proto__ = lecturer»

1.3. Способы установки прототипов

Есть три способа установки прототипа.

1.3.1. `__proto__`

Первый — это сеттер, геттер `__proto__`. Не идеальный метод:

- не является частью ECMAScript 5;
- он долгое время не являлся частью спецификации языка, и более того, поддерживался далеко не всеми платформами;
- появился он благодаря разработчикам браузеров, которые потихонечку внедряли эту возможность в свои продукты.

1.3.2. Метод `create`

Следующий способ установки прототипов — использование специального метода `create`, который в качестве параметра принимает в себя объект, который мы хотим видеть в качестве прототипа для нового объекта, который этот метод возвращает.

```
var student = Object.create(person)
```

Особенности способа:

- уже является частью ECMAScript 5;
- делает больше работы, чем простое присваивание ссылки;
- создаёт новые объекты и не может менять прототип существующих.

1.3.3. setPrototypeOf

Последний способ установки прототипа – специальный метод `setPrototypeOf`. Этот метод принимает уже два параметра: первый — исходный объект, а второй — объект, который мы хотим видеть в качестве прототипа для исходного.

```
var student = {
    name: 'Billy',
    sleep: function () {}
};

var person = {
    type: 'human',
    getName: function () {}
};

Object.setPrototypeOf(student, person);

student.getName();
// Billy
```

Особенности способа:

- появился только в ECMAScript 6
- близок к __proto__, но имеет особенность:
  - если мы попробуем присвоить через сеттер, геттер `__proto__` в качестве прототипа не объект, а число, то интерпретатор неявно проигнорирует это поведение; попробовав проделать тот же самый фокус с методом `setPrototypeOf`, интерпретатор поведет себя более явно и выбросит ошибку.

У метода `setPrototypeOf` есть парный метод `getPrototypeOf`. Этот метод возвращает ссылку на прототип. В отличие от `setPrototypeOf`, этот метод появился сравнительно давно в языке и позволяет нам проследовать по всей цепочке прототипов.

```
Object.getPrototypeOf(student) === person;
// true
Object.getPrototypeOf(person) === Object.prototype;
// true
Object.getPrototypeOf(Object.prototype) === null;
// true
```

1.4. Эффект затенения

Чтобы поменять значение какого-либо поля у объекта, нам достаточно выполнить простое присваивание. Но что, если мы попытаемся изменить поле, которого нет у объекта, но есть в его прототипе? Например, поле `type`.

```
var student = {
    name: 'Billy',
    [[Prototype]]: <person>
}

var person = {
    type: 'human',
    getName: function () {}
}

console.info(student.type); // human

student.type = 'robot';

console.info(student.type); // robot

console.info(person.type); // ???
```

Выполнив простое присваивание, мы добьемся желаемого. Может показаться,

что интерпретатор, не найдя это поле у студента, перейдет в прототип и поменяет значение уже там, но он оставит это поле в прототипе неприкосновенным. Вместо этого он создаст копию на стороне объекта, но уже с новым значением. Такой эффект называется **эффектом затенения свойства**.

```
console.info(person.type); // 'human'
```

Благодаря эффекту затенения мы можем переопределить методы, находящиеся в глобальном прототипе. Например, метод `toString`, который вызывается при приведении объекта к строке.

```
Object.prototype = {
    toString: function () {}
};
student.toString();
// [object Object]
console.info('Hello,' + student);
// Hello, [object Object]
```

1.5. Поля только для чтения в прототипах

Мы можем не просто установить поле, а задать ему некоторые характеристики, например, пометить это поле как изменяемое или неизменяемое.
Допустим, у нас есть объект студента с полем `name`, которое хранит его имя. Давайте добавим еще одно поле для этого объекта. Воспользуемся методом `defineProperty` и в качестве первого параметра передадим туда сам объект, в качестве второго параметра передадим название поля, а в качестве третьего — набор характеристик. Укажем значение поля, а также укажем, что это поле неизменяемое: зададим атрибуту `writable` значение `false`.

```
var student = { name: 'Billy' };
Object.defineProperty(student,'gender', {
    writable: false,
    value: 'male',
});
```

Если мы попытаемся перезаписать это начальное значение, то интерпретатор не даст нам этого сделать и сохранит исходное, причем сделает это неявно. Чтобы сделать поведение интерпретатора явным, нам необходимо переключиться в строгий режим интерпретации. Для этого понадобится добавить дополнительную директиву `use strict` в начало нашей программы. В этом случае при попытке перезаписать поле только для чтения интерпретатор бросит ошибку, в которой сообщит нам, что поле у объекта неизменяемое.

```
'use strict';
var student = { name: 'Billy' };
Object.defineProperty(student,
'gender'
, {
writable: false,
value: 'male'
});
```

Таким же образом работают неизменяемые поля в прототипах. Создадим новое поле в нашем прототипе person. Пусть это будет поле, которое хранит текущую планету, зададим ей начальное значение и атрибут `writable: false`. Мы увидим, что при попытке перезаписать это поле от лица исходного объекта, от лица студента, интерпретатор в строгом режиме также среагирует ошибкой, не даст нам этого сделать и скажет, что поле `planet` у объекта — неизменяемое.

```
Object.defineProperty(person,
'planet', {
    writable: false,
    value: 'Earth'
});

console.info(student.planet); // Earth

student.planet = 'Mars'; // TypeError: Cannot assign to read only
  ↪  property 'planet' of object
```

1.6. Сеттеры и геттеры в прототипах

Пусть у нас есть объект `student` с уже готовым полем, которое хранит имя студента. Мы хотим добавить для студента еще одно поле, которое будет хранить его возраст, такое, чтобы с ним было удобно работать.
Например, мы хотим передавать в это поле возраст в виде некоторой строки, но преобразовывать его внутри к числу. Для этого мы определяем сеттер и при помощи функции `parseInt` передаваемую строку, в которой содержится возраст, преобразуем к числу и сохраняем во внутреннее поле.

```
var student = {
    name: 'Billy',
    [[Prototype]]: <person>
};

Object.defineProperty(student,'age', {
    set: function(age) { this._age = parseInt(age); },
    get: function() { return this._age; }
});

student.age = '20 лет';

console.info(student.age); // 20;
```

Далее мы определяем геттер, который позволяет получать уже готовое значение из этого внутреннего поля. Имеет смысл добавить поле возраста не конкретно к студенту, а в его прототип, в объект `person`. Для этого мы воспользуемся тем же самым методом `defineProperty`, но в качестве первого параметра передадим уже не студента, а прототип. Далее попробуем указать возраст для студента в виде строки и увидим, что все работает как надо.

```
var student = {
    [[Prototype]]: <person>
};
var person = {
    type: 'human'
};
Object.defineProperty(person,'age', {
    set: function(age) { this._age = parseInt(age); },
    get: function() { return this._age; }
});

student.age = '20 лет';

console.info(student.age); // 20;

student.hasOwnProperty(age); // false;
```

Здесь мы вспоминаем про эффект затенения: если мы попытаемся установить некоторое поле, которого нет у объекта, но есть в прототипе, поле в прототипе не будет изменено. Вместо этого интерпретатор создаст копию этого поля на объекте с новым значением. Но если поле в прототипе определено при помощи сеттера/геттера, данный эффект работать не будет, копия поля у объекта `student` не появится.

> Если поле в прототипе определено как геттер или сеттер, то эффект затенения **не** работает.

1.7. Неперечисляемые поля в прототипах

Мы можем пометить некоторые поля у объекта как перечисляемые (значение по умолчанию) или неперечисляемые. Если поля перечисляемые, то при помощи оператора `for...in` мы можем получить весь список полей объекта.
Более того, оператор `for...in` перечисляет не только поля самого объекта, но и поля связанного с ним прототипа. Допустим, если у нашего объекта есть прототип и в нём есть поля/методы, оператор `for...in` перечислит их наряду с полями объекта.
Это может быть нежелательным поведением, и, возможно, мы хотим перечислить именно собственные поля объекта, не затрагивая при этом поля в

прототипе. Для этого нам понадобится специальный метод `hasOwnProperty`, в качестве аргумента который принимает название поля. Данный метод просто отвечает на вопрос: принадлежит ли это поле объекту или нет. Добавив это условие в оператор `for...in`, мы можем вывести только собственные поля объекта.

```
var student = {
    name: 'Billy',
    age: 20,
    [[Prototype]]: <person>
};
var person = {
    type: 'human',
    getName: function () {}
};

for (var key in student)
    if (student.hasOwnProperty(key)) console.info(key);

// 'age', 'name'
```

Аналогично этой технике мы можем воспользоваться другим методом — методом `keys`, который хранится в функции `Object`. Для этого в этот метод мы передаём объект, а на выходе получаем массив из ключей полей объекта.

```
var student = {
    name: 'Billy',
    [[Prototype]]: <person>
}
var person = {
    type: 'human',
    getName: function () {}
}

var keys = Object.keys(student); // Получаем массив ключей

console.info(keys);

// ['name']
```

Чтобы добавить в объект неперечисляемое поле, воспользуемся методом `defineProperty`. Для этого передадим в него первым параметром сам объект, вторым параметром — название нового поля, а третьим параметром — характеристики. Укажем значение этого объекта. И с помощью специального атрибута укажем, что это поле неперечисляемое.

```
var student = { name: 'Billy' };

Object.defineProperty(student,'age', {
    enumerable: false,
    value: '20'
});

for (var key in student) console.info(key);
// 'name'

Object.keys(student);
// ['name']
```

В результате данное поле не будет участвовать в перечислениях, организуемых оператором `for...in` или методом `keys`.
Таким же образом мы можем задать неперечисляемое поле в прототипе, также воспользовавшись методом `defineProperty`. И данное поле также не будет участвовать в перечислениях.

```
var student = {
    name: 'Billy',
    [[Prototype]]: <person>
};

var person = {
    type: 'human'
};

Object.defineProperty(person,'age', {
    enumerable: false
});

for (var key in student) console.info(key);
// 'name', 'type'
```

Важно заметить, что у глобальных прототипов для объектов или для массивов поля, обозначенные там, неперечисляемые. Мы не увидим их в попытке перечислить все поля конкретного объекта, несмотря на то, что в его прототипе может лежать глобальный прототип.

```
Object.prototype = {
    toString: function () {},
    [[Prototype]]: null
};
var person = {
    type: 'human',
    [[Prototype]]: <Object.prototype>
};

for (var key in person) console.info(key);

// 'type'
```