Яндекс

Прототипы

Александр Шоронов, разработчик интерфейсов

Как переиспользовать реализацию?

```
const lecturer = {
    name: 'Alex',
    getName() {
        return this.name;
    }
};
```

```
const student = {
    name: 'Ivan',
    getName() {
        return this.name;
    }
};
```

```
const person = {
    getName() {
        return this.name;
    }
};
```

```
const lecturer = {
    name: 'Alex',
    getName() {
        return this.name;
    }
};
```

```
const student = {
    name: 'Ivan',
    getName() {
        return this.name;
    }
};
```

```
const person = {
    getName() {
        return this.name;
    }
};
```

Задача — использовать общий вынесенный код person в student

Заимствование метода

```
const person = {
    getName() {
        return this.name;
    }
};

const student = {
    name: 'Ivan',
};


person.getName.call(student);              // student.getName();
```

Для создания такой связи между объектами есть специальное внутреннее поле…

[[Prototype]]

[[Prototype]]

```
const student = {
    name: 'Ivan',
    sleep() {}
    // [[Prototype]] =>
};
```

```
const person = {
    getName() {
        return this.name;
    }
};
```

```
student.getName(); // 'Ivan';
```

Объект, на который указывает ссылка в [[Prototype]], называется *прототипом*

Поиск метода интерпретатором

```
const person = {
    getName() {
        return this.name;
    }
};

const student = {
    name: 'Ivan',
    sleep() {},
    // [[Prototype]] => <Ссылка на person>
};

student.getName();
```

*Вызов метода **getName***

Поиск метода интерпретатором

```
const person = {
    getName() {
        return this.name;
    }
};

const student = {
    name: 'Ivan',
    sleep() {},
    // [[Prototype]] => <Ссылка на person>
};

student.getName();
```

Метода нет. Смотрим дальше на [[Prototype]]

Поиск метода интерпретатором

```
const person = {
    getName() {
        return this.name;
    }
};

const student = {
    name: 'Ivan',
    sleep() {},
    // [[Prototype]] => <Ссылка на person>
};

student.getName(); // 'Ivan';
```

Метода найден. Вызываем.
this** будет **student

Прототип объекта может также содержать в себе ссылку на другой прототип, образуя *цепочку прототипов*

Цепочка прототипов

```
const subject = {
    getName() { return this.name; }
};

const person = {
    // [[Prototype]]: <Ссылка на subject>
};

const student = {
    name: 'Ivan',
    // [[Prototype]]: <Ссылка на person>
};

student.getName();
```

Цепочка прототипов

```
Object.prototype = { /* [[Prototype]]: null */ };

const subject = { /* [[Prototype]]: <Object.prototype> */};
const person = { /* [[Prototype]]: <subject> */ };
const student = { /* [[Prototype]]: <person> */ };

student.getName(); // TypeError: student.getName is not a function
```

Интерпретатор будет идти по цепочке прототипов в поиске поля, пока не встретит null в поле [[Prototype]]

Базовый прототип

```
({}).toString();                    //?
```

Object.prototype – прототип для всех объектов по умолчанию. Содержит общие методы для всех объектов.

Базовый прототип

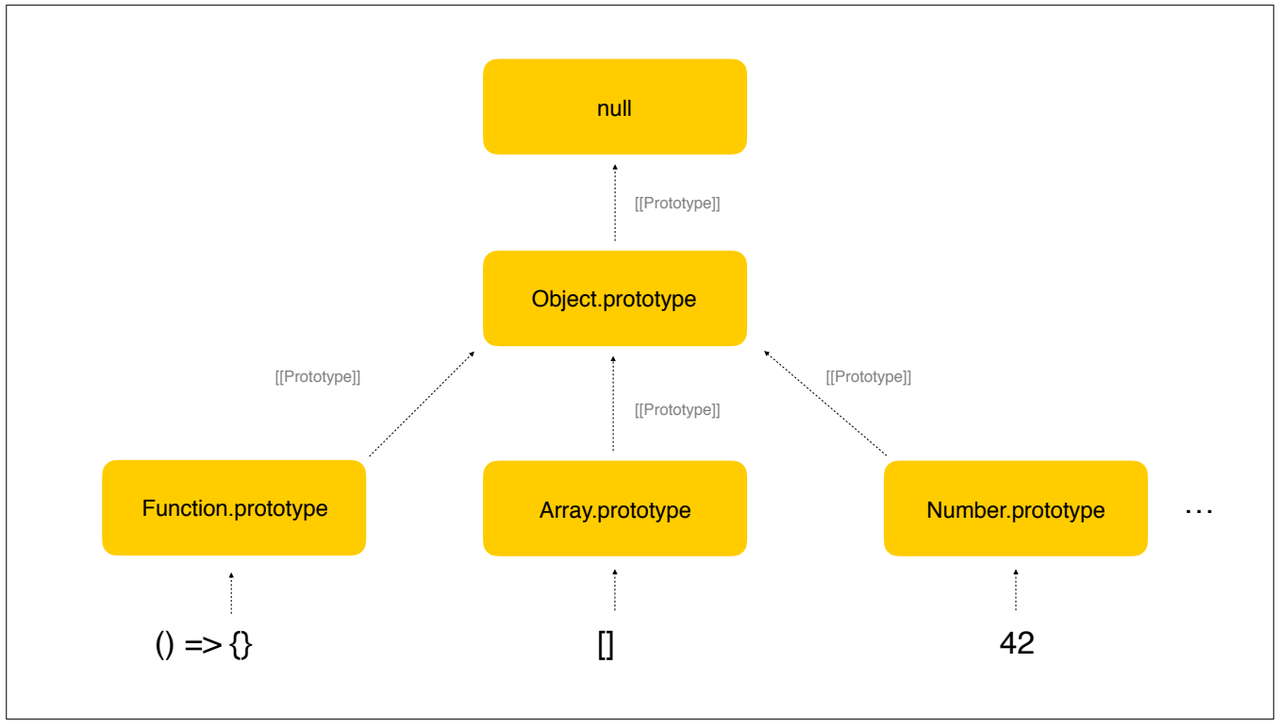
```
({
    // [[Prototype]] => <Object.prototype>
})
```

Функции-конструкторы

```
const arr = []; // new Array();
const obj = {}; // new Object();
const num = 42; // new Number(42);
```

Функции-конструкторы

```
Array.prototype = {
    concat() {},
    slice() {},
    splice() {},
    forEach() {},
    filter() {},
    map() {},
    // [[Prototype]]: <Object.prototype>
};
```


null
[[Prototype]]
Object.prototype
[[Prototype]]
[[Prototype]]
[[Prototype]]
Function.prototype
Array.prototype
Number.prototype
...
() => {}
[]
42

Как указать прототип у объекта?

```
Object.setPrototypeOf(obj, superObj);
```

Object.setPrototypeOf

```
const proto = { makeFun() {} };
const obj = {};

obj.makeFun(); // TypeError

Object.setPrototypeOf(obj, proto);

obj.makeFun(); // 👍
```

Object.setPrototypeOf

```
const proto = { makeFun() {} };
const obj = {};

Object.setPrototypeOf(obj, proto);
Object.setPrototypeOf(proto, obj);
/*
 * Thrown:
 * TypeError: Cyclic __proto__ value
 * at Function.setPrototypeOf (<anonymous>)
 */
```

Object.setPrototypeOf

```
const student = {};
const person = {};

Object.setPrototypeOf(student, person);
Object.setPrototypeOf(student, null);
Object.setPrototypeOf(student, 42); // Error


/*
 * proto может быть только Object или null
 */
```

Object.setPrototypeOf

```
const dict = Object.setPrototypeOf({}, null);
```

Object.getPrototypeOf

```
const student = {};
const person = {};

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

Warning: Changing the `[[Prototype]]` of an object is, by the nature of how modern JavaScript engines optimize property accesses, currently a very slow operation in every browser and JavaScript engine. In addition, the effects of altering inheritance are subtle and far-flung, and are not limited to simply the time spent in the `Object.setPrototypeOf(...)` statement, but may extend to ***any*** code that has access to any object whose `[[Prototype]]` has been altered.

Because this feature is a part of the language, it is still the burden on engine developers to implement that feature performantly (ideally). Until engine developers address this issue, if you are concerned about performance, you should avoid setting the `[[Prototype]]` of an object. Instead, create a new object with the desired `[[Prototype]]` using `Object.create()`.

Object.create

Object.create

```
const person = {
   getName() { return this.name; }
};

const lecturer = Object.create(person);
```

Object.create

```
const person = {
   getName() { return this.name; }
};

const lecturer = Object.create(person);

lecturer.name = 'Alex';
```

Object.create

```
const person = {
    getName() { return this.name; }
};

const lecturer = Object.create(person);

lecturer.name = 'Alex';

const student = Object.create(person, {
    name: 'Ivan'
});
```

Как обратиться к родительской реализацию?

Super

```
const person = {
    getName() { return this.name; }
};

const student = {
    name: 'Ivan',
    getName() { return 'Student ' + super.getName(); }
};

Object.setPrototypeOf(student, person);

student.getName(); // Student Ivan
```

При создании метода, его внутреннее поле [[HomeObject]] заполняется ссылкой на объект, в котором он определён

super ссылается на прототип объекта
из поля [[HomeObject]]

Псевдокод

```
student.getName.[[HomeObject]] == student;

super == Object.getPrototypeOf(student.getName.[[HomeObject]]);
```

У обычных полей-функций
[[HomeObject]] не заполняется

[[HomeObject]] функций-полей

```
const person = {
    getName() { return this.name; }
};

const student = {
    name: ‘Ivan',
    getName: function() {
        return 'Student ' + super.getName(); // Error
    }
};
```

SyntaxError: 'super' outside of function or class

Значение [[HomeObject]] нельзя изменить

[[HomeObject]] и this

```
const person = {
    getName() { return 'and person ' + this.name; }
};

const student = {
    name: 'Ivan',
    getName() { return 'Student ' + super.getName(); }
};

Object.setPrototypeOf(student, person);

const getName = student.getName; // this потеряли, но не super

getName(); // Student and person undefined
```

Атрибуты полей и прототипы

Установка поля объекта

```
const student = {
    name: 'Ivan'
};

student.age = 21;

student['age'] = 21;

Object.defineProperty(student, 'planet', {
    writeable: true,
    value: 'Earth'
});
```

getOwnPropertyDescriptor

```
const student = {};

Object.defineProperty(student, 'name', {
    value: 'Ivan'
});

Object.getOwnPropertyDescriptor(student, 'name');
/* {
 *      value: 'Ivan',
 *      writable: false,
 *      enumerable: false,
 *      configurable: false
 * }
 */
```

Эффект *затенения* полей

```
const person = {
    planet: 'Earth'
};

const student = Object.create(person);

student.planet = 'Mars';


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

console.info(person.planet); // Earth
```

Object.prototype.toString()

```
Object.prototype = {
   toString() {}
};

const student = {
   name: 'Ivan'
};

console.info('Hello, ' + student); // Hello, [object Object]
```

Object.prototype.toString()

```
Object.prototype = {
   toString() {}
};

const student = {
   name: 'Ivan'
};

student.toString = function {
   return this.name;
}

console.info('Hello, ' + student); // Hello, Ivan
```

Неперезаписываемые поля

```
const student = {};

Object.defineProperty(student, 'name', {
    writable: false, // Можно не указывать, по умолчанию false
    value: 'Billy'
});
```

Неперезаписываемые поля

```
const student = {};

Object.defineProperty(student, 'name', {
    writable: false, // Можно не указывать, по умолчанию false
    value: 'Billy'
});

student.name = 'Willy';

console.info(student.name); // Billy
```

Неявное поведение!

Неперезаписываемые поля и “use strict”

```
'use strict';

const student = {};

Object.defineProperty(student, 'name', {
   value: 'Billy'
});

student.name = 'Willy';

console.info(student.name);
```

TypeError: Cannot assign to read only property 'name' of object

Неперезаписываемые поля и в прототипах

```
const person = {};

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

const student = {};

Object.setPrototypeOf(student, person);

student.planet = 'Mars';
```

TypeError: Cannot assign to read only property

Перечисляемые поля

```
const student = {
    name: 'Billy',
    age: 21
};

for (let key in student) {
    console.info(key);
} // name, age
```

Перечисляемые поля и прототипы

```
const person = { planet: 'Earth' };

const student = {
    name: 'Billy',
    age: 20
};

Object.setPrototypeOf(student, person);

for (let key in student) {
    console.info(key);
} // name, age, planet
```

hasOwnProperty

```
const person = { planet: 'Earth' };

const student = {
    name: 'Billy',
    age: 21
};

Object.setPrototypeOf(student, person);

for (let key in student) {
    if (student.hasOwnProperty(key)) {
        console.info(key);
    }
} // name, age
```

Object.keys

```
const person = { planet: 'Earth' };

const student = {
    name: 'Billy',
    age: 21
};

Object.setPrototypeOf(student, person);

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

Object.entries

```
const person = { planet: 'Earth' };

const student = {
    name: 'Billy',
    age: 21
};

Object.setPrototypeOf(student, person);

for (let [key, value] of Object.entries(student)) {
    console.info(key);
} // name, age
```

Неперечисляемые поля

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

Object.defineProperty(student, 'age', {
   enumerable: false,
   value: 21
});

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

JSON.stringify(student); // '{"name":"Billy"}'

Object.assign({}, student); // { name: 'Billy' }
```

Неперечисляемые поля в прототипах

```
const person = {};

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

const student = { name: 'Billy' };

Object.setPrototypeOf(student, person);

for (let key in student) {
    console.info(key);
} // name
```

Неперечисляемые поля по умолчанию

```
Object.prototype = {
    toString() {}
};

const student = { name: 'Billy' };

for (let key in student) {
    console.info(key);
} // name
```

getOwnPropertyNames

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

Object.defineProperty(student, 'age', {
   value: 21
});

Object.getOwnPropertyNames(student);
// ['name', 'age']
```

set/get

```
let name = null; // Не будет доступна снаружи модуля

const student = {
    get name() { return 'Student ' + name; }

    set name(value) {
        name = value;
    }
};

student.name = 'Billy';

console.info(student.name); //Student Billy;
```

set/get в прототипах

```
let planet = null;

const person = {
    get planet() { return planet; },
    set planet(value) { planet = value; }
};

const student = {};

Object.setPrototypeOf(student, person);

student.planet = 'Mars';

student.hasOwnProperty('planet'); // false;
```

set/get

```
let name = null;

const student = {
    name: 'Willy',
    get name() { return 'Student ' + name; },
    set name(value) { name = value;}
};

student.name = 'Billy'

console.info(student.name); // Student Billy
```

set/get

```
let name = null;

const student = {
    get name() { return 'Student ' + name; },
    name: 'Willy', // get становится undefined
    set name(value) { name = value;}
};

student.name = 'Billy'

console.info(student.name); // undefined
```

set/get

```
let name = null;

const student = {
    name: 'Willy', // get становится undefined
    set name(value) { name = value;}
};

student.name = 'Billy'

console.info(student.name); // undefined
```

set/get

```
let name = null;

const student = {
    get name() { return 'Student ' + name; }
};

student.name = 'Billy'

console.info(student.name); // Student null
```

Поле одновременно может быть
либо *нормальным* либо *геттером/сеттером*

set/get

```
let name = null;

const student = {
   get name() { return 'Student ' + name; },
   set name(value) { name = value;}
};

Object.getOwnPropertyDescriptor(student, 'name');
// {
//     get: [Function: get],
//     set: [Function: set],
//     enumerable: true,
//     configurable: true
// }
```

Полезная литература

- Speaking JavaScript. Chapter 17. Objects and inheritance
- Exploring ES6. 14. New OOP features besides classes
- JavaScript engine fundamentals: optimizing prototypes
- MDN. Object
- Современный учебник JavaScript. Объекты и прототипы

Яндекс

Спасибо

Александр Шоронов
Разработчик интерфейсов