Материалы занятия

Курс: Разработка интерфейса на JavaScript

Дисциплина: Основы JavaScript

Тема занятия №28: Прототипы. Контекст. Виды функций. Замыкания

Встроенные прототипы

Свойство "prototype" широко используется внутри самого языка JavaScript. Все встроенные функции-конструкторы используют его.

Сначала мы рассмотрим детали, а затем используем "prototype" для добавления встроенным объектам новой функциональности.

Object.prototype

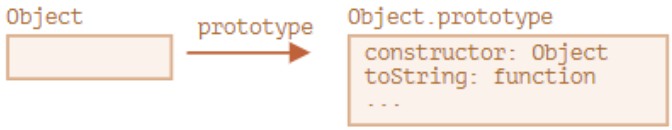
Давайте выведем пустой объект:

```
let obj = {};
alert( obj ); // "[object Object]" ?
```

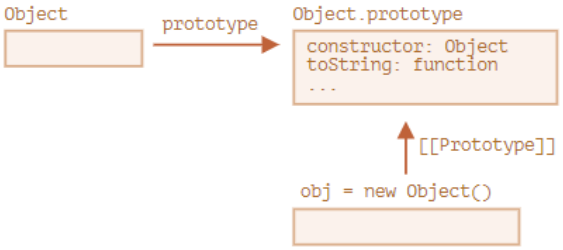
Где код, который генерирует строку "[object Object]"? Это встроенный метод toString, но где он? obj ведь пуст!

…Но краткая нотация obj = {} – это то же самое, что и obj = new Object(), где Object – встроенная функция-конструктор для объектов с собственным свойством prototype, которое ссылается на огромный объект с методом toString и другими.

Вот что происходит:



Когда вызывается new Object() (или создаётся объект с помощью литерала {...}), свойство [[Prototype]] этого объекта устанавливается на Object.prototype по правилам, которые мы обсуждали в предыдущей главе:



Таким образом, когда вызывается obj.toString(), метод берётся из Object.prototype.

Мы можем проверить это так:

```
let obj = {};

alert(obj.__proto__ === Object.prototype); // true
// obj.toString === obj.__proto__.toString === Object.prototype.toString
```

Обратите внимание, что по цепочке прототипов выше Object.prototype больше нет свойства [[Prototype]]:

```
alert(Object.prototype.__proto__); // null
```

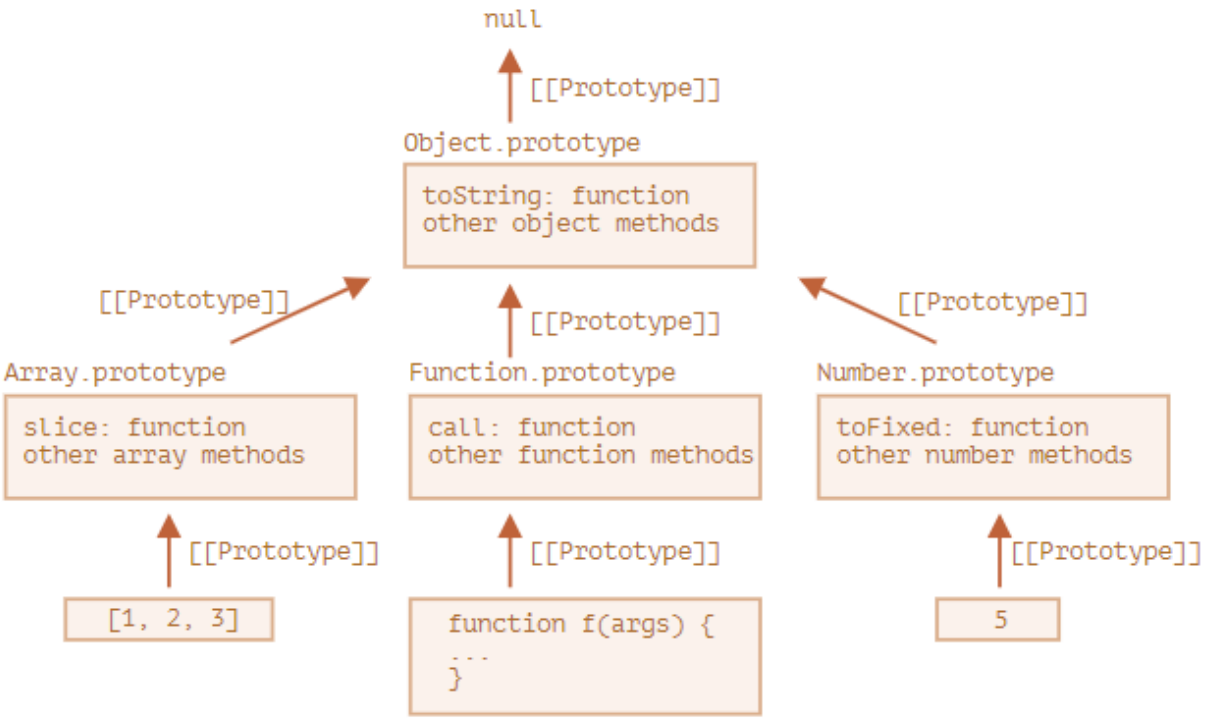
Другие встроенные прототипы

Другие встроенные объекты, такие как Array, Date, Function и другие, также хранят свои методы в прототипах.

Например, при создании массива [1, 2, 3] внутренне используется конструктор массива Array. Поэтому прототипом массива становится Array.prototype, предоставляя ему свои методы. Это позволяет эффективно использовать память.

Согласно спецификации, наверху иерархии встроенных прототипов находится Object.prototype. Поэтому иногда говорят, что «всё наследует от объектов».

Вот более полная картина (для трёх встроенных объектов):



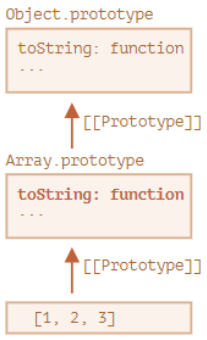
Давайте проверим прототипы:

```
let arr = [1, 2, 3];

// наследует ли от Array.prototype?
alert( arr.__proto__ === Array.prototype ); // true

// затем наследует ли от Object.prototype?
alert( arr.__proto__.__proto__ === Object.prototype ); // true

// и null на вершине иерархии
alert( arr.__proto__.__proto__.__proto__ ); // null
```

Некоторые методы в прототипах могут пересекаться, например, у Array.prototype есть свой метод toString, который выводит элементы массива через запятую:

```
let arr = [1, 2, 3]
alert(arr); // 1,2,3 <-- результат Array.prototype.toString
```

Как мы видели ранее, у Object.prototype есть свой метод toString, но так как Array.prototype ближе в цепочке прототипов, то берётся именно вариант для массивов:



В браузерных инструментах, таких как консоль разработчика, можно посмотреть цепочку наследования (возможно, потребуется использовать console.dir для встроенных объектов):

Другие встроенные объекты устроены аналогично. Даже функции – они объекты встроенного конструктора Function, и все их методы (call/apply и другие) берутся из Function.prototype. Также у функций есть свой метод toString.

```
function f() {}

alert(f.__proto__ == Function.prototype); // true
alert(f.__proto__.__proto__ == Object.prototype); // true, наследует от Object
```

Примитивы

Самое сложное происходит со строками, числами и булевыми значениями.

Как мы помним, они не объекты. Но если мы попытаемся получить доступ к их свойствам, то тогда будет создан временный объект-обёртка с использованием встроенных конструкторов String, Number и Boolean, который предоставит методы и после этого исчезнет.

Эти объекты создаются невидимо для нас, и большая часть движков оптимизирует этот процесс, но спецификация описывает это именно таким образом. Методы этих объектов также находятся в прототипах, доступных как String.prototype, Number.prototype и Boolean.prototype.

Значения null и undefined не имеют объектов-обёрток.

Специальные значения null и undefined стоят особняком. У них нет объектов-обёрток, так что методы и свойства им недоступны. Также у них нет соответствующих прототипов.

Изменение встроенных прототипов

Встроенные прототипы можно изменять. Например, если добавить метод к String.prototype, метод становится доступен для всех строк:

```
String.prototype.show = function() {
  alert(this);
};

"BOOM!".show(); // BOOM!
```

В течение процесса разработки у нас могут возникнуть идеи о новых встроенных методах, которые нам хотелось бы иметь, и искушение добавить их во встроенные прототипы. Это плохая идея.

Важно:

Прототипы глобальны, поэтому очень легко могут возникнуть конфликты. Если две библиотеки добавляют метод String.prototype.show, то одна из них перепишет метод другой.

Так что, в общем, изменение встроенных прототипов считается плохой идеей.

В современном программировании есть только один случай, в котором одобряется изменение встроенных прототипов. Это создание полифилов.

Полифил – это термин, который означает эмуляцию метода, который существует в спецификации JavaScript, но ещё не поддерживается текущим движком JavaScript.

Тогда мы можем реализовать его сами и добавить во встроенный прототип.

Например:

```
if (!String.prototype.repeat) { // Если такого метода нет
  // добавляем его в прототип

  String.prototype.repeat = function(n) {
    // повторить строку n раз

    // на самом деле код должен быть немного более сложным
    // (полный алгоритм можно найти в спецификации)
    // но даже неполный полифил зачастую достаточно хорош для использования
    return new Array(n + 1).join(this);
  };
}

alert( "La".repeat(3) ); // LaLaLa
```

Заимствование у прототипов

В главе Декораторы и переадресация вызова, call/apply мы говорили о заимствовании методов.

Это когда мы берём метод из одного объекта и копируем его в другой.

Некоторые методы встроенных прототипов часто одалживают.

Например, если мы создаём объект, похожий на массив (псевдомассив), мы можем скопировать некоторые методы из Array в этот объект.

Пример:

```
let obj = {
  0: "Hello",
  1: "world!",
  length: 2,
};

obj.join = Array.prototype.join;

alert( obj.join(',') ); // Hello,world!
```

Это работает, потому что для внутреннего алгоритма встроенного метода join важны только корректность индексов и свойство length, он не проверяет, является ли объект на самом деле массивом. И многие встроенные методы работают так же.

Альтернативная возможность – мы можем унаследовать от массива, установив obj.__proto__ как Array.prototype, таким образом все методы Array станут автоматически доступны в obj.

Но это будет невозможно, если obj уже наследует от другого объекта. Помните, мы можем наследовать только от одного объекта одновременно.

Заимствование методов – гибкий способ, позволяющий смешивать функциональность разных объектов по необходимости.

Итого

- Все встроенные объекты следуют одному шаблону:
  - Методы хранятся в прототипах (Array.prototype, Object.prototype, Date.prototype и т.д.).
  - Сами объекты хранят только данные (элементы массивов, свойства объектов, даты).
- Примитивы также хранят свои методы в прототипах объектов-обёрток: Number.prototype, String.prototype, Boolean.prototype. Только у значений undefined и null нет объектов-обёрток.
- Встроенные прототипы могут быть изменены или дополнены новыми методами. Но не рекомендуется менять их. Единственная допустимая причина – это добавление нового метода из стандарта, который ещё не поддерживается движком JavaScript.

Привязка контекста к функции

При передаче методов объекта в качестве колбэков, например для setTimeout, возникает известная проблема – потеря this.

В этой главе мы посмотрим, как её можно решить.

Потеря «this»

Мы уже видели примеры потери this. Как только метод передаётся отдельно от объекта – this теряется.

Вот как это может произойти в случае с setTimeout:

```
let user = {
  firstName: "Вася",
  sayHi() {
    alert(`Привет, ${this.firstName}!`);
  }
};

setTimeout(user.sayHi, 1000); // Привет, undefined!
```

При запуске этого кода мы видим, что вызов this.firstName возвращает не «Вася», а undefined!

Это произошло потому, что setTimeout получил функцию sayHi отдельно от объекта user (именно здесь функция и потеряла контекст). То есть последняя строка может быть переписана как:

```
let f = user.sayHi;
setTimeout(f, 1000); // контекст user потеряли
```

Метод setTimeout в браузере имеет особенность: он устанавливает this=window для вызова функции (в Node.js this становится объектом таймера, но здесь это не имеет значения). Таким образом, для this.firstName он пытается получить window.firstName, которого не существует. В других подобных случаях this обычно просто становится undefined.

Задача довольно типичная – мы хотим передать метод объекта куда-то ещё (в этом конкретном случае – в планировщик), где он будет вызван. Как бы сделать так, чтобы он вызывался в правильном контексте?

Решение 1: сделать функцию-обёртку

Самый простой вариант решения – это обернуть вызов в анонимную функцию, создав замыкание:

```
let user = {
  firstName: "Вася",
  sayHi() {
    alert(`Привет, ${this.firstName}!`);
  }
};

setTimeout(function() {
  user.sayHi(); // Привет, Вася!
}, 1000);
```

Теперь код работает корректно, так как объект user достаётся из замыкания, а затем вызывается его метод sayHi.

То же самое, только короче:

```
setTimeout(() => user.sayHi(), 1000); // Привет, Вася!
```

Выглядит хорошо, но теперь в нашем коде появилась небольшая уязвимость.

Что произойдёт, если до момента срабатывания setTimeout (ведь задержка составляет целую секунду!) в переменную user будет записано другое значение? Тогда вызов неожиданно будет совсем не тот!

```
let user = {
  firstName: "Вася",
  sayHi() {
    alert(`Привет, ${this.firstName}!`);
  }
};

setTimeout(() => user.sayHi(), 1000);

// ...в течение 1 секунды
user = { sayHi() { alert("Другой пользователь в 'setTimeout'!"); } };

// Другой пользователь в 'setTimeout'!
```

Следующее решение гарантирует, что такого не случится.

Решение 2: привязать контекст с помощью bind

В современном JavaScript у функций есть встроенный метод bind, который позволяет зафиксировать this.

Базовый синтаксис bind:

```
// полный синтаксис будет представлен немного позже
let boundFunc = func.bind(context);
```

Результатом вызова func.bind(context) является особый «экзотический объект» (термин взят из спецификации), который вызывается как функция и прозрачно передаёт вызов в func, при этом устанавливая this=context.

Другими словами, вызов boundFunc подобен вызову func с фиксированным this.

Например, здесь funcUser передаёт вызов в func, фиксируя this=user:

```
let user = {
  firstName: "Вася"
};

function func() {
  alert(this.firstName);
}

let funcUser = func.bind(user);
funcUser(); // Вася
```

Здесь func.bind(user) – это «связанный вариант» func, с фиксированным this=user.

Все аргументы передаются исходному методу func как есть

Теперь давайте попробуем с методом объекта:

```
let user = {
  firstName: "Вася",
  sayHi() {
    alert(`Привет, ${this.firstName}!`);
  }
};

let sayHi = user.sayHi.bind(user); // (*)

sayHi(); // Привет, Вася!

setTimeout(sayHi, 1000); // Привет, Вася!
```

В строке (*) мы берём метод user.sayHi и привязываем его к user. Теперь sayHi – это «связанная» функция, которая может быть вызвана отдельно или передана в setTimeout (контекст всегда будет правильным).

Здесь мы можем увидеть, что bind исправляет только this, а аргументы передаются как есть:

```
let user = {
  firstName: "Вася",
  say(phrase) {
    alert(`${phrase}, ${this.firstName}!`);
  }
};

let say = user.say.bind(user);

say("Привет"); // Привет, Вася (аргумент "Привет" передан в функцию "say")
say("Пока"); // Пока, Вася (аргумент "Пока" передан в функцию "say")
```

Удобный метод: bindAll

Если у объекта много методов и мы планируем их активно передавать, то можно привязать контекст для них всех в цикле:

```
for (let key in user) {
  if (typeof user[key] == 'function') {
    user[key] = user[key].bind(user);
  }
}
```

Некоторые JS-библиотеки предоставляют встроенные функции для удобной массовой привязки контекста, например _.bindAll(obj) в lodash.

Итого

Метод bind возвращает «привязанный вариант» функции func, фиксируя контекст this и первые аргументы arg1, arg2…, если они заданы.

Обычно bind применяется для фиксации this в методе объекта, чтобы передать его в качестве колбэка. Например, для setTimeout.

Когда мы привязываем аргументы, такая функция называется «частично применённой» или «частичной».

Частичное применение удобно, когда мы не хотим повторять один и тот же аргумент много раз.

Например, если у нас есть функция send(from, to) и from всё время будет одинаков для нашей задачи, то мы можем создать частично применённую функцию и дальше работать с ней.