Dmitry Soshnikov in ECMAScript | 2017-12-05

JavaScript. Ядро: 2-ое издание

Read this article in: English, Chinese, German.

Данная статья является *вторым изданием* обзорной лекции JavaScript. Ядро, посвященной языку программирования ECMAScript, и ключевым компонентам его рантайм-системы.

Аудитория: опытные программисты, эксперты.

Первое издание статьи описывает аспекты JavaScript с точки зрения стандарта ES3 (на данный момент устаревшего), с небольшими отсылками к соответствующим изменениям в ES5 и ES6 (также известного как ES2015).

Начиная с ES2015, в спецификации поменялись описания и структуры некоторых ключевых компонентов, были введены новые модели, термины и т.д. И в данном издании мы фокусируемся на новых абстракциях, обновленной терминологии, однако сохраняя базовую структуру JS, не изменяющуюся на протяжении всех версий стандарта.

Данная статья описывает рантайм-систему ES2017+.

Обратите внимание: последнюю версию ECMAScript спецификации можно найти на сайте комитета TC-39.

Мы начинаем нашу дискуссию с рассмотрения концепции *объекта*, являющейся фундаментальной абстракцией в ECMAScript.

Объект

ECMAScript — это *объектно-ориентированный* язык программирования с *прототипной* организацией, имеющий концепцию *объекта* в качестве базовой абстракции.

> **Определение 1: *Объект*** — это *коллекция свойств*, имеющая также

> связанный с ней *объект-прототип*. Прототипом может быть также другой объект, или же значение `null`.

Рассмотрим простейшую схему объекта, с которой будем работать в последующих описаниях. На свой прототип объект ссылается посредством внутреннего свойства `[[Prototype]]`, которое доступно в пользовательском коде через свойство `__proto__`.

Для кода:

```
let point = {
  x: 10,
  y: 20,
};
```

мы имеем следующую структуру с двумя *явными собственными свойствами* и одним *неявным (внутренним)* свойством `__proto__`, которое является ссылкой на прототип объекта `point`:

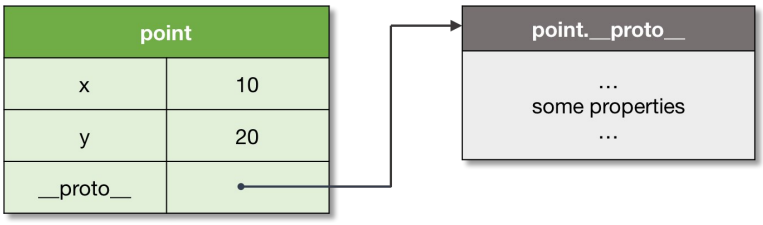

Схема 1. Простейший объект с прототипом.

Обратите внимание: объекты также могут хранить *символы*. Вы можете получить больше информации о символах в этой документации.

Прототипные объекты используются для реализации *наследования* при помощи механизма *динамической диспетчеризации (dynamic dispatch)*. Рассмотрим понятие *цепи прототипов*, чтобы увидеть этот механизм в действии.

Прототип

Каждый объект при создании получает свой *прототип*. Если прототип не задан *явно*, объекты получают *базовый прототип по-умолчанию* в качестве *объекта наследнования*.

> **Определение 2: *Прототип*** — это объект-делегат, используемый для реализации *прототипного наследования*.

При создании объекта, его прототип может быть установлен *явно* через свойство `__proto__`, или же с помощью метода `Object.create`:

```
// Базовый объект.
let point = {
  x: 10,
  y: 20,
};

// Наследуем от объекта `point`.
let point3D = {
  z: 30,
  __proto__: point,
```

```
};

console.log(
  point3D.x, // 10, унаследованное
  point3D.y, // 20, унаследованное
  point3D.z  // 30, собственное
);
```

Обратите внимание: по-умолчанию объекты получают `Object.prototype` в качестве наследуемого объекта.

Любой объект может быть использован в качестве прототипа другого объекта, и сам прототип может иметь свой собственный прототип. Если прототип имеет непустую ссылку на свой прототип, и т.д., такая связка называется *цепью прототипов (prototype chain)*.

> **Определение 3: *Цепь прототипов*** — это *конечная* цепь объектов, используемая для реализации *наследования* и *разделяемых свойств*.

Схема 2. Цепь прототипов.

Здесь правило очень простое: если свойство не найдено в самом объекте, осуществляется попытка *разрешить* (найти) это свойство в прототипе; в прототипе прототипа, и т.д. — до тех пор, пока вся цепь прототипов не будет рассмотрена.

Данный механизм известен как *динамическая диспетчеризация (dynamic dispatch)* или *делегация (delegation)*.

> **Определение 4: *Делегация*** — механизм, используемый для разрешения свойств в цепи наследования. Процесс осуществляется во время исполнения программы, поэтому также называется ***динамической диспетчеризацией***.

Обратите внимание: в отличие от *статической дипетчеризации*, когда ссылки разрешаются *во время компиляции*, *динамическая диспетчеризация* всегда разрешает ссылки *во время исполнения программы*.

Если свойство в итоге не найдено во всей цепи прототипов, возвращается значение `undefined`:

```
// "Пустой" объект.
let empty = {};

console.log(

  // функция, из прототипа по-умолчанию
  empty.toString,

  // undefined
  empty.x,

);
```

Как мы видим, обычный объект *никогда не является пустым* — он всегда наследует *что-то* из `Object.prototype`. Чтобы создать *беспрототипный*

словарь, необходимо явно установить его прототип в `null`:

```
// Не наследуем ни от кого.
let dict = Object.create(null);

console.log(dict.toString); // undefined
```

Механизм *динамической диспетчеризации* также позволяет *мутировать* цепь наследования и менять объект-делегат:

```
let protoA = {x: 10};
let protoB = {x: 20};

// То же, что и `let objectC = {__proto__: protoA};`:
let objectC = Object.create(protoA);
console.log(objectC.x); // 10

// Изменяем прототип:
Object.setPrototypeOf(objectC, protoB);
console.log(objectC.x); // 20
```

Обратите внимание: несмотря на то, что свойство `__proto__` на сегодняшний день стандартизовано, и проще для объяснения материала, на практике рекомендовано использование API методов для манипуляции с прототипами, таких как `Object.create`, `Object.getPrototypeOf`, `Object.setPrototypeOf`, и схожих в модуле `Reflect`.

На примере `Object.prototype`, мы видим, что *один и тот же прототип* может наследоваться *многими объектами*. На этом принципе простроено *классовое наследование* в ECMAScript. Давайте рассмотрим пример и заглянем в детали реализации абстракции “класс” в JS.

Класс

Когда несколько объектов имеют один и тот же набор свойств и одинаковое поведение, они образуют *классификацию*.

> **Определение 5: *Класс*** — это формальное абстрактное множество, описывающее одинаково начальное состояние и поведение его объектов.

В случае, если нам нужно иметь *несколько объектов*, наследуемых от одного и того же прототипа, мы конечно могли бы создать этот объект-прототип, и затем явно унаследовать его из наших вновь созданных объектов:

```
// Общий прототип для всех букв.
let letter = {
  getNumber() {
    return this.number;
  }
};

let a = {number: 1, __proto__: letter};
let b = {number: 2, __proto__: letter};
// ...
let z = {number: 26, __proto__: letter};

console.log(
  a.getNumber(), // 1
  b.getNumber(), // 2
  z.getNumber(), // 26
);
```

Данные отношения объектов представлены на следующей схеме:

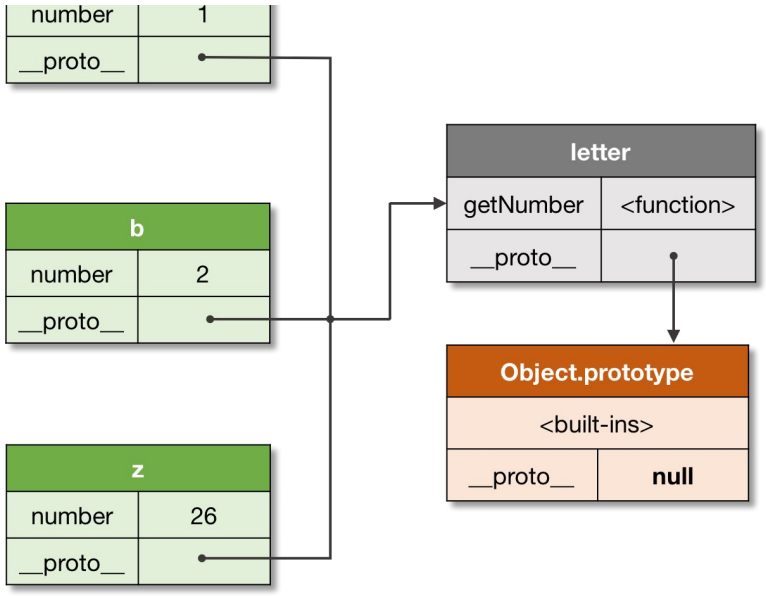

Схема 3. Разделяемый прототип.

Однако подобное описание объектов является *громоздким и неудобным*. И абстракция класса как раз и призвана разрешить эту проблему — будучи *синтаксическим сахаром* (т.е. конструкцией, которая *семантически выполняет те же функции*, но в более *удобной синтаксической форме*), она позволяет создавать множество объектов по удобному шаблону:

```
class Letter {
  constructor(number) {
    this.number = number;
  }

  getNumber() {
    return this.number;
  }
}

let a = new Letter(1);
let b = new Letter(2);
// ...
let z = new Letter(26);

console.log(
  a.getNumber(), // 1
  b.getNumber(), // 2
  z.getNumber(), // 26
);
```

Обратите внимание: *классовое наследование* в ECMAScript реализовано при помощи *прототипной делегации*.

Обратите внимание: понятие *“класса”* является лишь *теоретической абстракцией*. Технически она может быть реализована со *статической диспетчеризацией* как в Java или C++, или же *динамической диспетчеризацией (делегацией)* как в JavaScript, Python, Ruby, и т.д.

Технически “класс” представляет собой пару *“функция-конструктор + прототип”*. При этом, функция-конструктор *создает объекты*, а также *автоматически* устанавливает *прототип* для вновь созданных объектов. Этот прототип хранится в свойстве `<ФункцияКонструктор>.prototype`.

> **Определение 6: *Конструктор*** — это функция, которая используется для создания объектов и автоматической установки их прототипа.

Функцию-конструктор можно использовать и явно. Более того, до принятия стандартом абстракции класса, JavaScript программисты использовали именно этот подход, не имея лучшей альтернативы (мы все еще можем найти подобные устарелые конструкции в сети):

```
function Letter(number) {
  this.number = number;
}

Letter.prototype.getNumber = function() {
  return this.number;
};

let a = new Letter(1);
let b = new Letter(2);
// ...
let z = new Letter(26);

console.log(
  a.getNumber(), // 1
  b.getNumber(), // 2
  z.getNumber(), // 26
);
```

И если создание конструктора без наследования было достаточно простым, то создание наследуемой цепи классов становилось уже более проблематичной задачей. На сегодняшний день функции-конструкторы являются лишь *деталями реализации*, и это именно то, что скрывает в себе понятие “класса” в JavaScript.

Обратите внимание: *функции-конструкторы* являются лишь *деталями реализации* классового наследования.

Давайте посмотрим на связь объектов с их классом:

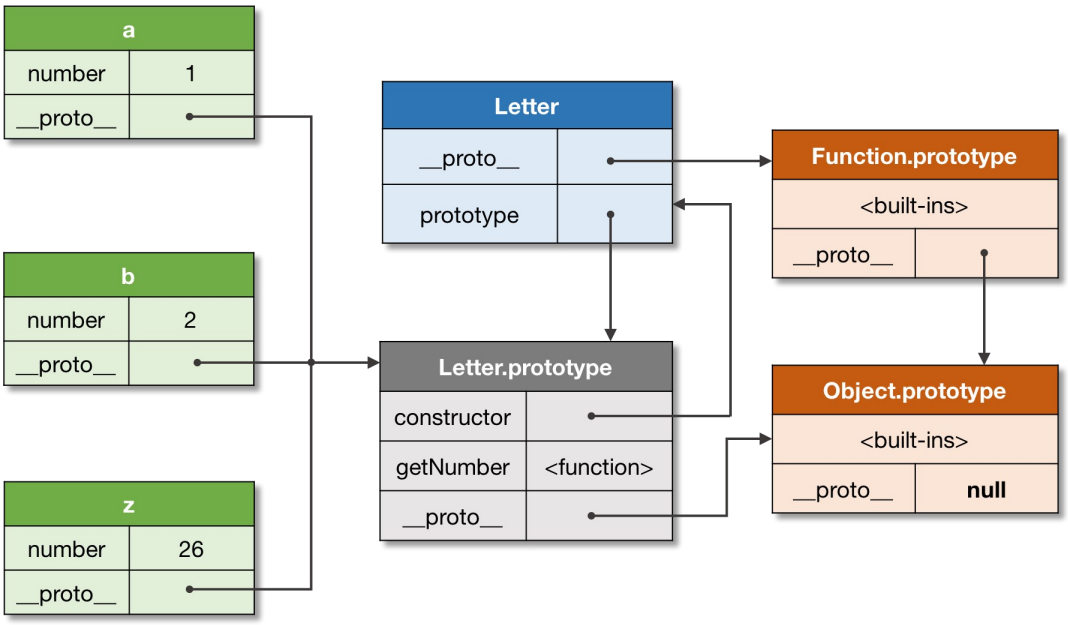

Схема 4. Связь объектов с конструктором.

Данная схема еще раз показывает, что *каждый объект* имеет ассоциированный с ним прототип. Даже сама функция-конструктор (класс) `Letter` также имеет свой прототип — `Function.prototype`. В свою очередь `Letter.prototype` является прототипом *объектов* класса `Letter`, т.е. `a`, `b` и `z`.

Обратите внимание: *непосредственным* прототипом *любого* объекта *всегда* является свойство `__proto__`. А явное свойство

`prototype` функции-конструктора — это всего лишь ссылка на прототип ее *объектов*; из объектов прототип по-прежнему доступен через свойство `__proto__`. См. детали в этой статье.

Вы можете найти подробные описания основных ООП концепций (включая различия между классовой и прототипной организациями) в статье ES3. 7.1 ООП: Общая теория.

Теперь, когда мы имеем представление об основных связях между ECMAScript объектами, давайте подробней рассмотрим *рантайм-систему* JS. Как мы увидим, практически все в ней также может быть представлено объектом.

Контекст исполнения

Для запуска и контроля выполнения JS-кода, ECMAScript спецификация определяет понятие *контекста исполнения (execution context)*. Логически контексты исполнения формируются в *стек* (*стек контекстов исполнения*, как мы увидим ниже), который соответствует общему понятию *стека вызовов (call-stack)*.

> **Определение 7: *Контекст исполнения*** — это абстрактное понятие, используемое спецификацией ECMAScript для типизации и разграничения исполняемого кода.

Существует несколько типов исполняемого кода ECMAScript: *глобальный код*, *код функции*, `eval` *код* и *код модуля*; каждый код запускается в своем контексте исполнения. Различные типы кода и их соотвествующие объекты могут влиять на структуру контекста исполнения: так, например, *функции-генераторы* сохраняют свой *объект-генератор* в качестве свойства контекста исполнения.

Рассмотрим рекурсивный вызов функции:

```
function recursive(flag) {

  // Условие выхода.
  if (flag === 2) {
    return;
  }

  // Рекурсивный вызов.
  recursive(++flag);
}

// Поехали!
recursive(0);
```

При вызове функции создается *новый контекст исполнения* и *добавляется (push)* в стек — в этот момент он становится *активным контекстом*. При возврате из функции контекст *удаляется (pop)* из стека.

Контекст, который вызывает другой контекст, называется *вызывающим (caller)*. Соответственно, контекст, который вызывали, называется *вызванным (callee)*. В нашем примере функция `recursive` играет обе роли: callee и caller — когда вызывает себя рекурсивно.

> **Определение 8: *Стек контекстов исполнения*** — это LIFO структура, используемая для контроля и очередности исполнения кода.

Для нашего примера выше мы имеем следующие *“push-pop”* модификации стека:

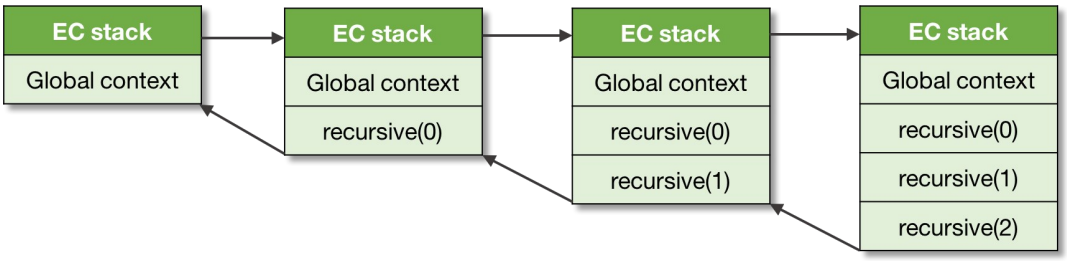

Схема 5. Стек контекстов исполнения.

Как можно видеть, *глобальный контекст* находится всегда первым элементом в стеке; он создается до выполнения любого другого кода и существует до конца выполнения программы.

Вы можете найти подробное описание контекстов исполнения в соответствующей главе.

В общем случае код контекста *выполняется до полного завершения*, однако, как мы отмечали выше, некоторые объекты — такие как *генераторы*, могут нарушать LIFO-порядок стека. Функция-генератор может остановить исполнение контекста где-нибудь посередине и *удалить* его из стека *до того, как исполнение закончится*. Когда этот генератор будет активирован вновь, его контекст будет *возобновлен* и вновь *положен* на стек:

```
function *gen() {
  yield 1;
  return 2;
}

let g = gen();

console.log(
  g.next().value, // 1
  g.next().value, // 2
);
```

Инструкция `yield` возвращает значение наружу и приостанавливает работу контекста, удаляя его со стека. В следующем вызове метода `next`, *тот же самый контекст* вновь добавлен в стек и *продолжен*. Подобные контексты могут *пережить* вызывающие контексты, которые их создают — отсюда и нарушение LIFO-структуры.

Обратите внимание: подробней о генераторах и итераторах можно прочитать в этой документации.

Теперь мы готовы к обсуждению важных компонентов контекстов исполнения; в частности мы увидим, как ECMAScript-рантайм организовывает *хранилище переменных* и *области видимости (scope)*, создаваемые вложенными блоками кода. Данная тема относится к общей теории *лексических окружений*, которые используются в JS для хранения переменных, а также для решения так называемой *“Фунарг проблемы”* — как мы увидим, при помощи механизма *замыканий*.

Лексическое окружение

Каждый контекст исполнения имеет ассоциированное с ним *лексическое окружение (lexical environment)*.

> **Определение 9: *Лексическое окружение*** — это структура, используемая для ассоциации *идентификаторов*, появляющихся в контексте, с их значениями. Каждое лексическое окружение также может иметь ссылку на *родительское окружение*.

Итак, лексическое окружение — это *хранилище* переменных, функций, и классов, объявленных в области видимости данного контекста.

Обратите внимание: вы можете найти пример *реализации* лексического окружения в соответствующей лекции из курса Основы Интерпретации.

Технически лексическое окружение представляет собой *пару*, состоящую из *записи окружения* (непосредственное хранилище-таблица, ассоциирующая идентификаторы с их значениями), а также ссылка на родительское окружение (которая может быть `null`).

Для кода:

```
let x = 10;
let y = 20;

function foo(z) {
  let x = 100;
  return x + y + z;
}

foo(30); // 150
```

Структуры окружений *глобального контекста* и контекста функции `foo` будут выглядеть следующим образом:

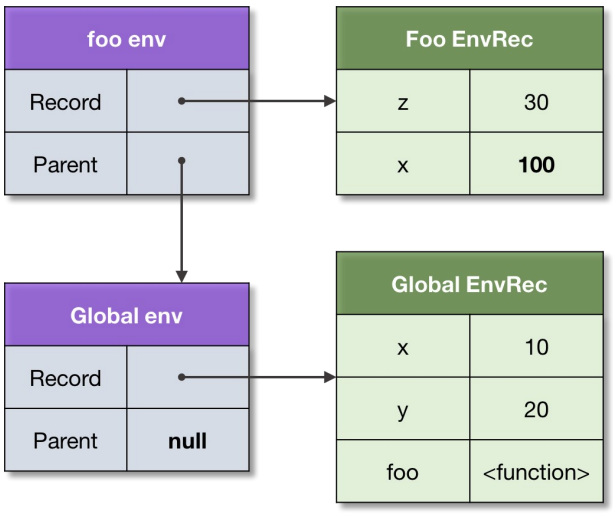

Схема 6. Цепь лексических окружений.

Логически это напоминает *цепь прототипов*, которую мы обсуждали выше. И правило для *разрешения идентификаторов (identifier resolution)* очень схоже: если переменная *не найдена* в *родном* окружении, осуществляется попытка найти ее в *родительском окружении*, в родителе родителя, и т.д. —

до тех, пока вся *цепь окружений* не будет рассмотрена.

> **Определение 10: *Разрешение идентификаторов*** — процесс поиска переменной *(идентификатора)* в цепи окружений. Неразрешенный идентификатор выбрасывает исключение `ReferenceError`.

Это объясняет, почему переменная `x` разрешена как `100`, а не `10` — она найдена в *родном* окружении функции `foo`; почему мы имеем доступ к параметру `z` — он тоже добавлен в *окружение активации*; а также почему мы имеем доступ к переменной `y` — она найдена в родительском окружении.

Аналогично прототипам одно и то же родительское окружение может разделяться несколькими дочерними окружениями: например, две глобальные функции разделяют одно и то же глобальное окружение.

Обратите внимание: вы можете найти подробное описание лексических окружений в этой статье.

Записи окружений различаются по *типам*. Так есть ***объектные*** записи окружений (object environment records) и ***декларативные*** записи окружений (declarative environment records). На базе декларативных записей также основаны ***функциональные*** записи окружений (function environment records) и ***модульные*** записи окружений (module environment records). Каждый тип записи имеет специфичные только для него свойства. Однако базовый механизм разрешения идентификаторов является общим для всех окружений, независимо от типа их записи.

Примером *объектной записи окружения* может служить запись *глобального окружения*. Такая запись имеет ассоциированный с ней *объект связей (binding object)*, который может хранить лишь некоторые свойства из записи, но не другие, и наоборот. Объект связей может быть также использован в качестве значения `this`.

```
// Устаревшее объявление переменной, используя `var`.
var x = 10;

// Современное объявление переменной, используя `let`.
let y = 20;

// Обе переменные добавлены в запись окружения:
console.log(
  x, // 10
  y, // 20
);

// Но только `x` добавлена в "объект связей".
// Объектом связей глобального окружения является
// глобальный объектом, и равен `this`:

console.log(
  this.x, // 10
  this.y, // undefined!
);

// Объект связей может хранить имя, которое не
// добавляется в запись окружения, поскольку не
// является валидным идентификатором:

this['not valid ID'] = 30;

console.log(
  this['not valid ID'], // 30
);
```

Это отображено на следующей схеме:

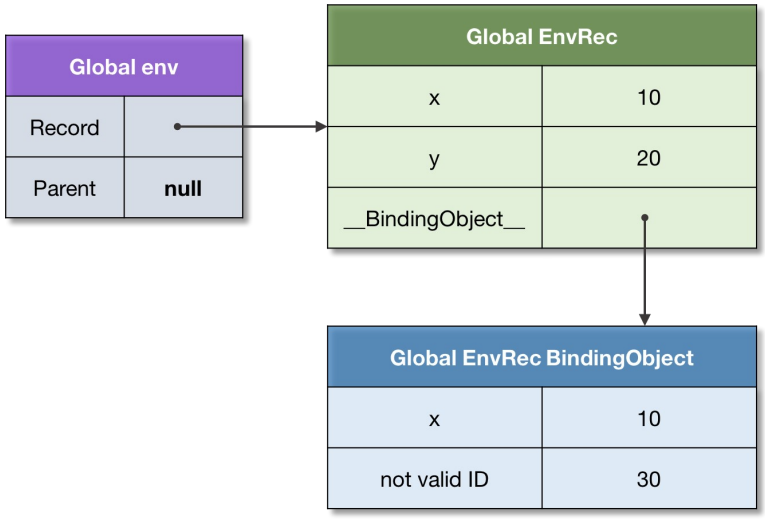

Схема 7. Объект связей.

Стоит отметить, что объект связей специфицирован лишь для описания (устаревших) конструкций из *предыдущих версий стандарта*, таких как `var`-декларации и `with`-инструкции, которые также предоставляют свой объект в качестве объекта связей. Это в основном исторические причины ECMAScript, когда лексические окружения были представлены обычными объектами. Сегодняшняя модель окружений является более оптимизированной, однако, как результат, мы больше не можем обращаться к переменным как к свойствам.

Мы уже видели, как окружения связаны посредством родительской ссылки. Теперь давайте рассмотрим, как лексическое окружение может *пережить* создающий ее контекст. Это является базисом для механизма *замыканий*, которые мы и обсудим в следующем разделе.

Замыкание

Функции в ECMAScript являются *объектами первого класса (first-class objects)*. Эта концепция является фундаментальной для *функционального программирования*, аспекты которого поддерживаются в JavaScript.

> **Определение 11: *Функция первого класса*** — функция, которая может быть использованна в качестве обычных данных: т.е. сохранена в переменную, передана в качестве аргумента, или возвращена в качестве значения из другой функции.

С понятием функций первого класса связана так называемая “Фунарг проблема” (или *“Проблема функционального аргумента”*). Проблема возникает, когда функция использует *свободные переменные*.

> **Определение 12: *Свободная переменная*** — переменная, не являющаяся *ни параметром*, *ни локальной переменной* данной функции.

Давайте посмотрим на проблему Фунарга и увидим, как она решена в ECMAScript.

Рассмотрим следующий код:

```
let x = 10;

```

```
function foo() {
  console.log(x);
}

function bar(funArg) {
  let x = 20;
  funArg(); // 10, но не 20!
}

// Передаем `foo` в качестве аргумента в `bar`.
bar(foo);
```

Для функции `foo` переменная `x` является *свободной*. При вызове функции `foo` (посредством параметра `funArg`) — в каком окружении должна быть разрешена переменная `x`? Во *внешней* области видимости, где функция была *создана*, или в области видимости *вызывающего контекста*? Как мы видим, вызывающая сторона, т.е. функция `bar`, также определяет переменную `x` — со значением `20`.

Случай, описанный выше, известен как ***нисходящая фунарг-проблема***, т.е. *неоднозначность*, возникающая при определении *правильного лексического окружения* свободной переменной: должно ли это быть окружение *времени создания*, или же окружение *времени вызова*?

Данная проблема решена соглашением использования *статической области видимости (static scope)*, т.е. окружения *времени создания*.

> **Определение 13: *Статическая область видимости*** — язык программирования использует *статическую область видимости*, если только по анализу исходного кода, можно определить, в каком лексическом окружении будут разрешены свободные переменные.

Статическая область видимости иногда также называется *лексической областью видмости (lexical scope)*, отсюда и название *лексических окружений*.

Технически статическая область видимости реализована через *захват лексического окружения*, в котором функция *создается*.

Обратите внимание: вы можете прочитать подбробней про *статическую* и *динамическую* области видимости в этой статье.

В нашем примере окружение, захваченное функцией `foo`, — это *глобальное окружение*:

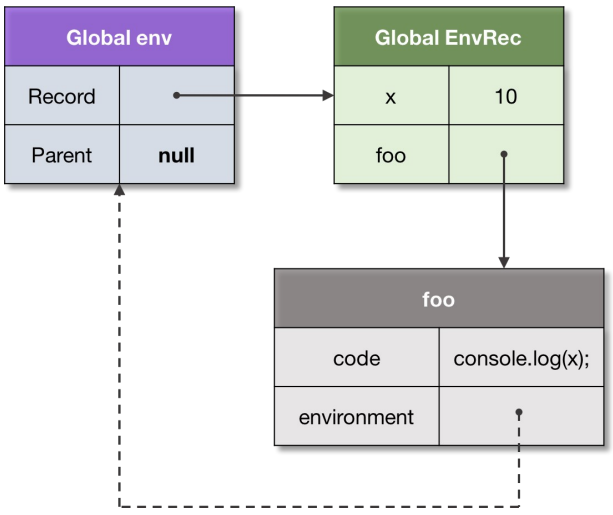

Схема 8. Замыкание.

Видим, что окружение ссылается на функцию, которая в свою очередь ссылается *обратно* на (захваченное) лексическое окружение.

> **Определение 14: *Замыкание*** — это функция, *захватывающая лексическое окружение* того контекста, где она *создана*. В дальнейшем это окружение используется для *разрешения идентификаторов*.

Обратите внимание: функция *вызывается* во *вновь созданном окружении активации*, которое содержит *локальные переменные* и *аргументы*. *Родительское окружение* устанавливается в *захваченное окружение* функции, реализуя семантику *лексической области видимости*.

Второй тип Фунарг-проблемы известен как ***восходящая фунарг-проблема***. Единственное отличие здесь в том, что захваченное окружение *переживает* порождающий ее контекст.

Рассмотрим следующий пример:

```
function foo() {
  let x = 10;

  // Замыкание, захватываем окружение `foo`.
  function bar() {
    return x;
  }

  // Восходящий фунарг.
  return bar;
}

let x = 20;

// Вызов `foo` возвращает замыкание `bar`.
let bar = foo();

bar(); // 10, но не 20!
```

Повторим, технически данный случай *ничем не отличается* от единого механизма захвата порождающего окружения. Просто в этом случае, если бы у нас не было замыкания, активационное окружение функции `foo` *было бы уничтожено*. Однако мы его *захватили*, и поэтому оно *не может быть удалено*, и сохраняется — для обеспечения семантики *статической области видимости*.

Часто можно видеть неполное понимание замыканий — обычно программисты определяют замыкания лишь в рамках восходящей фунарг-проблемы (и на практике, действительно, это наиболее частый случай). Однако, как мы можем видеть, технический механизм *нисходящего* и *восходящего* фунарга *абсолютно идентичен*, и является *механизмом статической области видимости*.

Как было отмечено выше, подобно прототипам, одно и то же родительское окружение может *разделяться несколькими* замыканиями. Это обеспечивает доступ для чтения и записи разделяемых данных:

```
function createCounter() {
  let count = 0;

  return {
    increment() { count++; return count; },
    decrement() { count--; return count; },
```

```
  };
}

let counter = createCounter();

console.log(
  counter.increment(), // 1
  counter.decrement(), // 0
  counter.increment(), // 1
);
```

Поскольку оба замыкания, `increment` и `decrement`, созданы в области видимости, содержащей переменную `count`, они *разделяют* данное *родительское окружение*. Т.е., захват происходит всегда *“по ссылке”* — имея в виде *ссылку* на *всё родительское окружение* целиком.

Мы можем видеть это на следующей схеме:

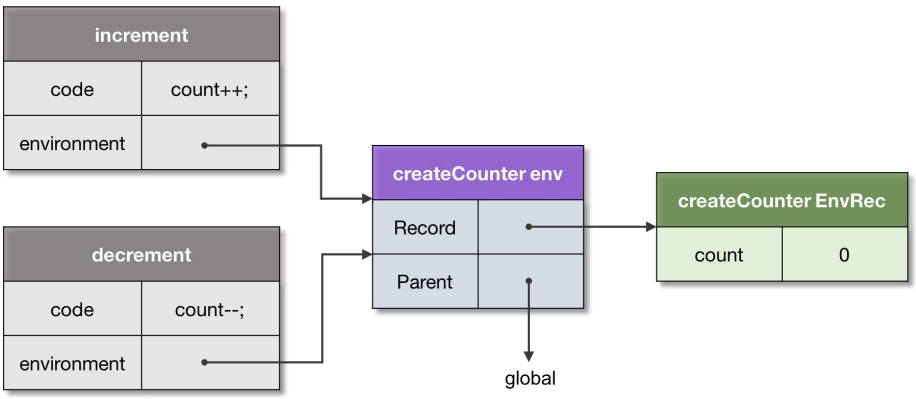

Схема 9. Разделяемое окружение.

Некоторые языки программирования могут захватывать свободные переменные *по значению*, создавая их копии, и не позволяя их изменение в родительской области видимости. Однако в JS, повторим, это всегда *ссылка* на родительское окружение.

Обратите внимание: реализации могут оптимизировать этот шаг и не захватывать всё окружение целиком. Захватывая *только используемые* свободные переменные, они однако по-прежнему обеспечивают семантику мутации данных в родительских областях видимости.

Вы можете найти подробное обсуждение замыканий и Фунарг-проблемы в соответствующей главе.

Итак, область видимости всех идентификаторов является статической. Существует однако *одно* значение, чья область видимости является *динамической* в ECMAScript. Это значение `this`.

This

Значением `this` является специальный объект, который *динамически* и *неявно* передается в код контекста исполения. Мы можем рассматривать его как *неявный дополнительный параметр*, к которому мы имеем доступ, но который не можем изменять.

Основным назначением `this` является исполнение одного и того же кода, но

в контексте разных объектов.

> **Определение 15: *This*** — неявный *объект контекста*, доступный из кода, для возможности применения данного кода для разных объектов.

Главным прецедентом в данном случае является классовое ООП. Метод объекта (объявленный в прототипе) существует в *единственном экземпляре*, однако же *разделяется* между *всеми объектами* этого класса.

```
class Point {
  constructor(x, y) {
    this._x = x;
    this._y = y;
  }

  getX() {
    return this._x;
  }

  getY() {
    return this._y;
  }
}

let p1 = new Point(1, 2);
let p2 = new Point(3, 4);

// Имеем доступ к `getX` и `getY` из
// обоих объектов (они переданы как `this`).

console.log(
  p1.getX(), // 1
  p2.getX(), // 3
);
```

Когда запускается метод `getX`, создается новое активационное окружение для хранения локальных переменных и параметров. В дополнение, *функциональная запись окружения* получает `[[ThisValue]]`, которое передается *динамически*, в зависимости от того, в какой форме функция *вызвана*. Когда она вызывается с объектом `p1`, значение `this` устанавливается именно в `p1`, а в следующем случае это уже `p2`.

Другим применением `this` являются *общие интерфейсные функции*, которые могут быть использованы в *примесях (mixins)* или *штрихах (traits)*.

В следующем примере, интерфейс `Movable` содержит общую функцию `move`, ожидающую от пользователей реализации свойств `_x` и `_y`:

```
// Общий интерфейс (примесь) Movable.
let Movable = {

  /**
   * Данная функция является обобщенной, и работает
   * с любым объектом, реализующим свойства `_x` и `_y`,
   * не зависимо от класса данного объекта.
   */
  move(x, y) {
    this._x = x;
    this._y = y;
  },
};

let p1 = new Point(1, 2);

// Делаем `p1` подвижной точкой.
Object.assign(p1, Movable);

// Имеем доступ к методу `move`.
p1.move(100, 200);

console.log(p1.getX()); // 100
```

Часто примеси добавляют *на уровне прототипа*, вместо подмешивания их *каждому объекту*, как мы сделали для примера выше.

И для того, чтобы показать еще раз динамическую природу значения `this`,

рассмотрим следующий пример, который мы оставляем читателю в качестве упражнения и задачи для решения:

```
function foo() {
  return this;
}

let bar = {
  foo,

  baz() {
    return this;
  },
};

// `foo`
console.log(
  foo(),       // global или undefined

  bar.foo(),   // bar
  (bar.foo)(), // bar

  (bar.foo = bar.foo)(), // global
);

// `bar.baz`
console.log(bar.baz()); // bar

let savedBaz = bar.baz;
console.log(savedBaz()); // global
```

Поскольку лишь по анализу исходного кода функции `foo` мы *не можем определить*, какое значение `this` будет иметь *в определенном вызове*, мы говорим, что значение `this` имеет *динамическую область видимости*.

Обратите внимание: вы можете найти подробное объяснение, как определяется значение `this`, и почему код выше работает именно таким образом, в соотвествующей главе.

Стрелочные функции (arrow functions) являются исключением из правила в случае определния значения `this`: их `this` всегда *лексический (статический)*, а *не динамический*. Т.е. их функциональная запись окружения *не предоставляет значение* `this`, и оно наследуется из *родительского окружения*.

```
var x = 10;

let foo = {
  x: 20,

  // Динамический `this`.
  bar() {
    return this.x;
  },

  // Лексический `this`.
  baz: () => this.x,

  qux() {
    // Лексический в рамках данного вызова.
    let arrow = () => this.x;

    return arrow();
  },
};

console.log(
  foo.bar(), // 20, из `foo`
  foo.baz(), // 10, из global
  foo.qux(), // 20, из `foo` и стрелочной функции
);
```

Как было отмечено, в *глобальном контексте* значением `this` является *глобальный объект*. И в предыдущих версиях JS был только один глобальный объект. Текущая же версия стандарта определяет *множество глобальных объектов*, которые являются частью *областей (сфер) кода*. Рассмотрим эти структуры подробней.

Область кода (Сфера)

Перед исполнением ECMAScript-код должен быть ассоциирован с определенной *областью кода* или *сферой (realm)*. Технически сфера предоставляет собой инкапсулированное *глобальное окружение* для контекста исполнения.

Обратите внимание: русский перевод английского слова “Realm”. Наряду с “Областью” мы равнозначно используем перевод “Сфера”, чтобы исключить путаницу с “Областью видимости” (“Scope”).

> **Определение 16: *Область кода (Сфера)*** — это объект предоставляющий отдельное *глобальное окружение* контексту исполнения.

При *создании контекста исполнения* он *ассоциируется* с определенной *областью кода*. Данная ассоциация в дальнейшем *остается неизменной*.

Обратите внимание: прямым эквивалентом области кода (сферы) в браузере является элемент `iframe`, который именно и создает отдельное глобальное окружение. В Node.js это близко к песочнице (sandbox) vm-модуля.

Текущая версия спецификации не предоставляет возможности явного создания областей кода, однако они могут быть созданы неявно реализацией. Существует также предложение предоставить данный API в пользовательский код.

Логически же, каждый контекст из стека исполнения всегда ассоциирован со своей областью кода:

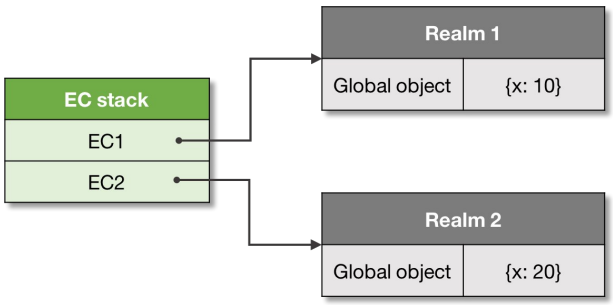

Схема 10. Ассоциация контекста и области кода.

Давайте посмотрим на пример раздельных областей кода, используя модуль `vm`:

```
const vm = require('vm');

// Первая область кода и её глобальный объект:
const realm1 = vm.createContext({x: 10, console});

// Вторая область кода и её глобальный объект:
const realm2 = vm.createContext({x: 20, console});

// Код для запуска:
const code = `console.log(x);`;

vm.runInContext(code, realm1); // 10
vm.runInContext(code, realm2); // 20
```

Итак, мы постепенно приближаемся к пониманию общей картины рантайм-системы ECMAScript. До этого, однако, мы должны понять и определить *точку входа* в программу, а также увидеть ее *процесс инициализации*. Данная область управляется механизмом *задач (job)* и *очередей задач (job queues)*.

Задача

Некоторые операции могут быть *отложенными* и запускаться, как только для них появится свободное место в стеке контекстов исполнения.

> **Определение 17: *Задача (работа)*** — это абстрактная операция, инициирующая ECMAScript вычисление, когда *нет никаких других* вычислений в данный момент.

Обратите внимание: русский перевод английского слова “Job”. Несмотря на прямой перевод слова как “Работа”, мы используем равнозначный перевод “Задача” (“Task”), как наиболее частый в русскоязычной литературе.

Задачи добавляются в ***очереди задач (job queues)***, и в текущей версии спецификации существует две очереди задач: ***ScriptJobs*** (задачи скриптов), and ***PromiseJobs*** (задачи “обещаний”).

И *начальная задача* в очереди *ScriptJobs* и является *главной точкой входа* в нашу программу — этот тот начальный скрипт, который загружается и запускается на исполнение: создается сфера кода, создается глобальный контекст и ассоциируется с этой сферой; он помещается на стек, и происходит запуск глобального кода.

Обратите внимание, что очередь *ScriptJobs* обрабатывает как *скрипты*, так и *модули*.

Дальше этот контекст может создавать и запускать *другие контексты* или добавлять в очередь *другие задачи*. Примером отложенной задачи может являться *обещание (promise)*.

Когда *нет запущенных* контекстов исполнения, и стек контекстов *пуст*, ECMAScript удаляет первую *ожидающую задачу* из очереди задач, создает для нее контекст исполнения и запускает ее код на исполнение.

Обратите внимание: очереди задач обычно обслуживаются абстракцией, известной как ***“Цикл событий” (“Event loop”)***. ECMAScript стандарт не описывает цикл событий, оставляя его детали реализациям, однако вы можете найти обучающий пример — здесь.

Например:

```
// Добавляем в очередь PromiseJobs новое "обещание".
new Promise(resolve => setTimeout(() => resolve(10), 0))
  .then(value => console.log(value));

// Этот log запускается раньше, поскольку он часть все еще
```

```
// запущенного контекста, а задачи не могут начать выполнение
// в этом случае.
console.log(20);

// Вывод: 20, 10
```

Обратите внимание: вы можете подробней прочитать об “обещаниях” в этой документации.

Асинхронные функции (async functions) могут *ожидать (await)* “обещания”, и поэтому так же добавляют задачу в очередь:

```
async function later() {
  return await Promise.resolve(10);
}

(async () => {
  let data = await later();
  console.log(data); // 10
})();

// Также выводится раньше, поскольку асинхронные
// функции добавляют задачу в очередь PromiseJobs.
console.log(20);

// Вывод: 20, 10
```

Обратите внимание: вы можете прочитать подробней об асинхронных функциях в данной документации.

Итак, мы подошли совсем близко к финальной картине текущей JS Вселенной. Осталось только рассмотреть *главных владельцев* всех этих компонентов, которые мы обсуждали выше. Данные владельцы известны как *Агенты (Agents)*.

Агент

Многозадачность и *параллелизм* реализованы в ECMAScript посредством *шаблона Агентов*. Шаблон Агентов очень близок к шаблону Акторов (Actor pattern) — *легковесный процесс* с коммуникаций через *посылку сообщений (message-passing)*.

> **Определение 18: *Агент*** — это абстракция, инкапсулирующая в себе стек контекстов исполнения, набор очередей задач, и областей кода.

В зависимости от реализации, агент может запускаться как в том же треде, так и в отдельном. `Worker` агент в браузерной среде является примером концепции *Агентов*.

Агенты *изолировны друг от друга* в состоянии и могут коммуницировать посредством *посылки сообщений*. Некоторые данные однако могут *разделяться* агентами, например объекты класса `SharedArrayBuffer`. Агенты также могут объединяться в *кластеры агентов (agent clusters)*.

В примере ниже, `index.html` вызывает агента `agent-smith.js`, передавая разделяемый участок памяти:

```
// Разделяемые данные между этим агентом и другим.
let sharedHeap = new SharedArrayBuffer(16);

// Наше представление общих данных.
let heapArray = new Int32Array(sharedHeap);

```

```
// Создаем нового агента (worker).
let agentSmith = new Worker('agent-smith.js');

agentSmith.onmessage = (message) => {
  // Агент посылает индекс измененных им данных.
  let modifiedIndex = message.data;

  // Проверяем, что данные изменены:
  console.log(heapArray[modifiedIndex]); // 100
};

// Передаем разделяемые данные агенту.
agentSmith.postMessage(sharedHeap);
```

И код агента следующий:

```
// agent-smith.js

/**
 * Получаем разделяемые данные в этом агенте.
 */
onmessage = (message) => {
  // Наше представление общих данных.
  let heapArray = new Int32Array(message.data);

  let indexToModify = 1;
  heapArray[indexToModify] = 100;

  // Посылаем индекс в сообщении назад.
  postMessage(indexToModify);
};
```

Вы можете найти полный код примера — здесь.

(Обратите внимание, при запуске данного примера на локальной машине, запускайте его в Firefox, т.к. Chrome, ввиду причин безопасности, не позволяет загружать web-worker'ов из локальных файлов)

Итак, ниже мы можем видеть полную картину рантайм-системы ECMAScript:

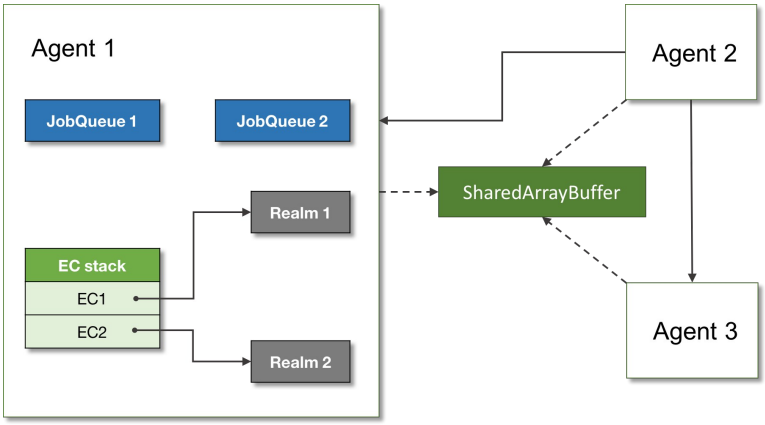

Схема 11. Рантайм-система ECMAScript.

Именно так устроен JavaScript внутри!

На этом мы завершаем наш разбор. Это тот объем материала о JS ядре, который удается поместить в рамки одной статьи. Как мы отмечали, JS код может быть сгруппирован в *модули*, свойства объектов могут быть отслежены `Proxy`-объектами, и т.д, т.д. — в новой версии JS существует множество пользовательских деталей, которые вы сможете найти в различной документации по JavaScript.

Здесь же мы попытались представить *логическую структуру* самой ECMAScript программы, и надеемся данная обзорная лекция прояснила эти детали. Если у вас возникнут какие-либо вопросы, предложения или отзывы,

— как всегда, я буду рад обсудить их в комментариях.

Я хочу поблагодарить представителей TC-39 и редакторов спецификации, которые помогли прояснить некоторые нюансы для этой статьи. Вы можете найти оригинальную дискуссию в этом Twitter треде.

Удачи в изучении ECMAScript!

Автор перевода: Дмитрий Сошников
Дата перевода: 5 декабря, 2017

Автор оригинала: Дмитрий Сошников
Дата оригинала: 14 ноября, 2017

Dmitry Soshnikov

Software engineer interested in learning and education. Sometimes blog on topics of programming languages theory, compilers, and ECMAScript.

Published
2017-12-05

Write a Comment

RELATED CONTENT BY TAG **CLOSURE** **CORE** **ECMASCRIPT** **FUNARG** **JAVASCRIPT** **LEXICAL ENVIRONMENT** **PROTOTYPE** **RUSSIAN** **THIS**