Шлыкова Ирина, ДКИ-302

*Перевела часть книги Understanding ESMAScript5, Nikolas С. Zakas.*

*Не все, конечно, лаконично, но постаралась донести основную мысль.*

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

В ECMAScript 5, строки, наконец, получили trim(), и ECMAScript 6 продолжает расширение функциональности строк.

Разработчики использовали IndexOf () как способ выявления строк внутри других строк.

ECMAScript 6 добавляет три новых методов, целью которых является выявление строки внутри других строк:

• contains()- возвращает истину, если данный текст находится в любом месте в строке или ложь, если нет.

• StartsWith () - возвращает истину, если данный текст найден в начале строки или ложь, если нет.

• EndsWith () - возвращает истину, если данный текст находится в конце строки или ложь, если нет.

Каждый из этих методов принимает два аргумента: текст для поиска и откуда начинать поиск. Когда второй аргумент опущен, содержит () и StartsWith () начинается поиск с начала строки, при EndsWith () начинается с конца.

Примеры:

**var** msg = "Hello world!";

console.log(msg.startsWith("Hello")); *// true*

console.log(msg.endsWith("!")); *// true*

console.log(msg.contains("o")); *// true*

console.log(msg.startsWith("o")); *// false*

console.log(msg.endsWith("world!")); *// true*

console.log(msg.contains("x")); *// false*

console.log(msg.startsWith("o", 4)); *// true*

console.log(msg.endsWith("o", 8)); *// true*

console.log(msg.contains("o", 8)); *// false*

Эти три метода позволяют намного легче индетифицировать подстроки, не выявляя их точное положение.

Все эти методы возвращают логическое значение. Если нужно найти позицию строки в другой, можно использовать IndexOf () или LastIndexOf ().

**repeat()**

ECMAScript 6 также появляется repeat () для строк. Этот метод принимает один аргумент, который передается несколько раз, чтобы повторить строку, и возвращает новую строку.

Например:

console.log("x".repeat(3)); *// "xxx"*

console.log("hello".repeat(2)); *// "hellohello"*

console.log("abc".repeat(4)); *// "abcabcabcabc"*

Эта функция особенно полезна при работе с текстом. Одним из примеров, где эта функция полезна. В примере нужно создать уровни отступов:

*// indent using a specified number of spaces*

**var** indent = " ".repeat(size),

indentLevel = 0;

*// whenever you increase the indent*

**var** newIndent = indent.repeat(++indentLevel);

**Object.is()**

Для сравнения, обычно, мы используем (==) или (===). Но === не всегда точен.

Например, 0 === -0 – равны, NaN === NaN возвращает ложь, для этого требуется использование isNaN() для верного обнаружения NaN.

ECMAScript 6 вводит Object.is (). Этот метод принимает два аргумента и возвращает истину, если значения эквивалентны. Два значения считаются эквивалентными, если они того же типа и имеют одинаковое значение. Во многих случаях, Object.is () работает так же, как ===. Единственное отличие заключается в том, что +0 и -0 считаются не эквивалентны и NaN приравнивается к NaN. Вот несколько примеров:

console.log(+0 == -0); *// true*

console.log(+0 === -0); *// true*

console.log(Object.is(+0, -0)); *// false*

console.log(**NaN** == **NaN**); *// false*

console.log(**NaN** === **NaN**); *// false*

console.log(Object.is(**NaN**, **NaN**)); *// true*

console.log(5 == 5); *// true*

console.log(5 == "5"); *// true*

console.log(5 === 5); *// true*

console.log(5 === "5"); *// false*

console.log(Object.is(5, 5)); *// true*

console.log(Object.is(5, "5")); *// false*

В основном мы все равно будем использовать для сравнения == и ===, поэтому byObject.is () полезен только для исключений.

**Выводы:**

Были добавлены дополнительные методы работы со строками, что позволяет более легко определить подстроки независимо от того, где они находятся. Метод Object.is () выполняет строгое равенства на любое значение, фактически став более безопасной версией === при работе со специальными значениями JavaScript.

**Числа.**

Для целочисленных (integer) и с плавающей точкой(float) используется один тип. Числа хранятся в формате IEEE 754 двойной точности с плавающей точкой, и в тот же формат используется для представления обоих типов. В качестве одного из основополагающих типов данных JavaScript, цифры имеют весьма важное значение для разработчиков. Учитывая новый акцент на играх и графики в JavaScript, ECMAScript 6 стремится сделать работу с числами легче и мощнее.

**Восьмеричное и двоичные литералы**

ECMAScript 5 стремились упростить некоторые общие числовые ошибки, удалив предварительно включенное восьмеричное целое буквальное обозначение в двух местах: ParseInt () и строгий режим.

ECMAScript 5 сократить использование восьмеричных чисел. Во-первых, ParseInt () было изменено таким образом, что она игнорирует нули в первом аргументе, когда нет второй аргумент. Это означает, число которых не может быть случайно рассматривается как восьмеричное больше. Второе изменение заключается в устранении восьмеричное буквальное обозначение в строгом режиме. Попытка использовать восьмеричную буквальное в строгих результатов мод в синтаксической ошибки.

*// ECMAScript 5*

**var** number = 071; *// 57 in decimal*

**var** value1 = parseInt("71"); *// 71*

**var** value2 = parseInt("071"); *// 71*

**var** value3 = parseInt("071", 8); *// 57*

**function** getValue() {

"use strict";

**return** 071; *// syntax error*

}

Делая эти два изменения, ECMAScript 5 стремились устранить много путаницы и ошибок, связанных с восьмеричных литералов.

ECMAScript 6 взял восьмеричную систему исчисления, наряду с двоичной.

Используется: 0xor 0X.

Новый формат восьмеричного исчисления начинается с 0o или 0О, то время как новый бинарный начинается с 0b or0B. Каждому типу следует одна или несколько цифр, 0-7 для восьмеричной, 0-1 для двоичных. Вот пример:

*// ECMAScript 6*

**var** value1 = 0o71; *// 57 in decimal*

**var** value2 = 0b101; *// 5 in decimal*

Добавление этих двух типов позволяет разработчикам JavaScript, чтобы быстро и легко включить числовые значения в двоичной, восьмеричной, десятичной и форматов шестнадцатеричных, что очень важно в определенных типах математических операций.

Метод ParseInt () не обрабатывает строки, которые выглядят как восьмеричные или бинарные:

console.log(parseInt("0o71")); *// 0*

console.log(parseInt("0b101")); *// 0*

Тем не менее, Number() преобразует строку, содержащую восьмеричные или двоичные литералы правильно:

console.log(Number("0o71")); *// 57*

console.log(Number("0b101")); *// 5*

**isFinite () и IsNaN ()**

JavaScript давно было несколько глобальных методов идентификации определенных типов номеров:

• isFinite () определяет, является ли значение конечным числом (не бесконечность или минус бесконечность)

• IsNaN () определяет, является ли значение NaN (с NaN является единственным значением, которое не равно самому себе)

Хотя предназначен для работы с числами, эти методы способны выведение числового значение от и значение, которое передается дюйма, что оба метода могут возвращать неверные результаты при передаче значение, которое не является числом. Например:

console.log(isFinite(25)); *// true*

console.log(isFinite("25")); *// true*

console.log(isNaN(**NaN**)); *// true*

console.log(isNaN("NaN")); *// true*

Оба isFinite () и IsNaN () передают свои аргументы через номер (), чтобы получить числовое значение, а затем выполняют сравнение на этом числовом значение, а не оригинале. Когда типы значений не проверяются перед использованием с одним из этих методов, могут привести к ошибке.

ECMAScript 6 добавляет два новых метода, которые выполняют то же сравнение, но только для числовых значений: Number.isFinite () и Number.isNaN (). Эти методы всегда вернуться ложным, когда принял нечисловое значение и возвращают те же значения, что и их глобальные партнеры при передаче числового значения:

console.log(isFinite(25)); *// true*

console.log(isFinite("25")); *// true*

console.log(Number.isFinite(25)); *// true*

console.log(Number.isFinite("25")); *// false*

console.log(isNaN(**NaN**)); *// true*

console.log(isNaN("NaN")); *// true*

console.log(Number.isNaN(**NaN**)); *// true*

console.log(Number.isNaN("NaN")); *// false*

Эти два новых метода, направленные на устранение определенных типов ошибок, которые могут быть вызваны, когда не-числовых значений используются с isFinite () и IsNaN ().

**ParseInt () и parseFloat ()**

Глобальные функции ParseInt () и parseFloat () теперь также есть в Number.parseInt () and Number.parseFloat (). Эти функции ведут себя точно так же, как и глобальных функций одного и того же имени. Единственная цель в создании является классифицировать глобальные функции, которые явно относятся к определенному типу данных. Так как эти функции как создать номера из строк, они уже поступили в номер вместе с другими функциями, которые относятся к номерам. (Since these functions both create numbers from strings, they are now on Number along with the other functions that relate to numbers.)

**ОПРЕДЕЛЕНИЕ целых чисел.**

Первым дополнением является Number.isInteger (), которое позволяет определить, является ли значение целым в JavaScript. Цифры, которые выглядят как дробные могут на самом деле быть сохранены в виде целых чисел, и поэтому возвращают истинное от Number.isInteger (). Например:

console.log(Number.isInteger(25)); *// true*

console.log(Number.isInteger(25.0)); *// true*

console.log(Number.isInteger(25.1)); *// false*

В этом коде Number.isInteger () возвращает справедливо как для 25 и 25,0, хотя последние выглядит как дробное. Простое добавление десятичную точку в ряде автоматически не делает его дробным в JavaScript. 25,0 на самом деле просто 25, оно хранится в виде целого числа. 25,1 число, однако, хранится в виде дробного, потому что есть значение фракции.

Безопасные целые (не уверена в точности перевода).

Однако все не так просто с целыми числами. JavaScript может только точно представлять целые числа между -253 и 253, и за пределами этого "безопасного" диапазона, двоичные представления в конечном итоге повторяются для нескольких числовых значений. Например:

console.log(Math.pow(2, 53)); *// 9007199254740992*

console.log(Math.pow(2, 53) + 1); *// 9007199254740992*

Этот пример не содержит опечатка, два различных номера в конечном итоге представлены одним и тем же целым числом.

ECMAScript 6 вводит Number.isSafeInteger (), чтобы лучше определить целые числа, которые точно могут быть представлены на языке. Существует также Number.MAX\_SAFE\_INTEGER и Number.MIN\_SAFE\_INTEGER, что представляют собой верхние и нижние границы одного и того же диапазона, соответственно. Метод Number.isSafeInteger () гарантирует, что значение является целым числом и находится в пределах безопасного диапазона целочисленных значений:

**var** inside = Number.MAX\_SAFE\_INTEGER,

outside = inside + 1;

console.log(Number.isInteger(inside)); *// true*

console.log(Number.isSafeInteger(inside)); *// true*

console.log(Number.isInteger(outside)); *// true*

console.log(Number.isSafeInteger(outside)); *// false*

inside является крупнейшим безопасно целым, поэтому оно возвращает верное значение как для Number.isInteger (), так и для Number.isSafeInteger (). Outside больше не является безопасным, хотя оно по-прежнему - целое число.

Вывод: Number.isSafeInteger () можно использовать как часть проверки входных данных.

**Математические методы.**

Многие математические расчеты можно сделать более эффективными.

В результате ECMAScript 6 добавляет несколько новых методов объекта Math. Эти новые методы важны для повышения скорости общих математических вычислений, и поэтому, улучшая скорость работы приложений, которые должны выполнять много вычислений (например, графических программ).

Math.acosh (х) Возвращает гиперболический косинус х.

Math.asinh (х) Возвращает гиперболический синус х.

Math.atanh (х) Возвращает гиперболический тангенс х

Math.cbrt (х) Возвращает кубиками корень х.

Math.clz32 (х) Возвращает количество ведущих нулевых битов в 32-разрядное целое представление х.

Math.cosh (х) Возвращает гиперболический косинус х.

Math.expm1 (х) Возвращает результат вычитания 1 из экспоненциальной функцией х

Math.fround (х) Возвращает ближайший одинарной точностью и плавающей точкой х.

Math.hypot (... значения) Возвращает квадратный корень из суммы квадратов каждого аргумента.

Math.imul (х, у) Возвращает результат выполнения истинное 32-битного умножения двух аргументов.

Math.log1p (х) Возвращает натуральный логарифм 1 + х.

Math.log10 (х) Возвращает база 10 логарифм х.

Math.log2 (х) Возвращает логарифм по основанию 2 х.

Math.sign (х) Возвращает -1, если х отрицательно 0, если х равно +0 или -0, или 1, если х является положительным.

Math.sinh (х) Возвращает гиперболический синус х.

Math.tanh (х) Возвращает гиперболический тангенс х.

Math.trunc (х) Удаляет фракция цифры от поплавка и возвращает целое число.

**Выводы:**

ECMAScript 6 облегчает работу с числами за счет внедрения нового синтаксиса и новых методов. Формы бинарные и восьмеричные литералы позволяют встраивать числа непосредственно в исходном коде, сохраняя наиболее подходящий для представление вид. Есть Number.isFinite () и Number.isNaN (), это более безопасные версии глобальных методов с теми же именами в связи с их отсутствием типа принуждения. Можно более легко идентифицировать целые используя Number.isInteger () и Number.isSafeInteger (), а также выполнить намного больше математических операций благодаря новым методам по математике.