1

В JavaScript, по сути, существует два вида значений. Первый вид — примитивные значения, второй — объектные . Примитивные значения включают в себя простые типы данных наподобие чисел, логических значений, строк, значений null и undefined.

Примитивные значения иммутабельны. Их нельзя изменять. То есть, например 1 как значение – не меняется.

Конечно, в переменную, хранящую примитивное значение, можно записать что-то новое:

Но при этом не происходит изменения (мутации) примитивного числового значения 1.

-----------

2

Создаются новые символы с помощью функции Symbol():

Имена свойств объекта могут быть строками JavaScript, или тем, что может быть сконвертировано в строку, включая пустую строку. Как бы то ни было, доступ к любому имени свойства, которое содержит невалидный JavaScript идентификатор (например, имя свойства содержит в себе пробел и тире или начинается с цифры), может быть получен с использованием квадратных скобок

При создании символу можно дать описание (также называемое имя), в основном использующееся для отладки кода:

Теперь давайте посложнее.

Вот тут symbol становится немного странным, все другие примитивы могут стать объектами, если объявить их через new, а symbol не может. Почему так?

Простыми словами — от символа мы хотим получить уникальный идентификатор, то есть сам Символ, а new возвращает объект, объект нам не нужен.

---------

3

**Символы не преобразуются автоматически в строки !**

Большинство типов данных в JavaScript могут быть неявно преобразованы в строку. Например, функция alert принимает практически любое значение, автоматически преобразовывает его в строку, а затем выводит это значение, не сообщая об ошибке. Символы же особенные и не преобразуются автоматически.

Это – языковая «защита» от путаницы, ведь строки и символы – принципиально разные типы данных и не должны неконтролируемо преобразовываться друг в друга.

Если же мы действительно хотим вывести символ с помощью alert, то необходимо явно преобразовать его с помощью метода .toString():

Или мы можем обратиться к свойству symbol.description, чтобы вывести только описание:

4

Символы позволяют создавать «скрытые» свойства объектов, к которым нельзя нечаянно обратиться и перезаписать их из других частей программы.

Например такой код перезапишет свойство id

Символы, конечно, не добавляют в JavaScript возможностей по созданию приватных свойств объектов, но они являются ценным новшеством языка по другим причинам. А именно, они полезны в ситуациях, когда неким библиотекам нужно добавлять свойства в объекты, описанные за их пределами, и при этом не опасаться коллизии имён свойств объектов.

5

Если мы хотим использовать символ при литеральном объявлении объекта {…}, его необходимо заключить в квадратные скобки.

Это вызвано тем, что нам нужно использовать значение переменной id в качестве ключа, а не строку «id».

6

Свойства, чьи ключи – символы, не перебираются циклом for…in

Это – часть общего принципа «сокрытия символьных свойств». Если другая библиотека или скрипт будут работать с нашим объектом, то при переборе они не получат ненароком наше символьное свойство. Object.keys(user) также игнорирует символы.

А вот Object.asssign копирует и строковые, и символьные свойства. Идея заключается в том, что, когда мы клонируем или объединяем объекты, мы обычно хотим скопировать *все* свойства.

7

Иногда мы хотим, чтобы символы с одинаковыми именами были одной сущностью. Например, разные части нашего приложения хотят получить доступ к одному символу , подразумевая именно одно и то же свойство.

Для этого существует *глобальный реестр символов*.

Для чтения (или, при отсутствии, создания) символа из реестра используется вызов

Symbol.for(key)

Если мы ещё раз вызовем Symbol.for(“Kanye West”) он вернёт существующий символ, а не новый.

Для глобальных символов, кроме Symbol.for(key), который ищет символ по имени, существует обратный метод: Symbol.keyFor(sym), который, наоборот, принимает глобальный символ и возвращает его имя.

8

Представим, у вас есть некий объект в приложении, который используется много где, да и с ним работаете не только вы, он передаётся по разным функциям, над ним издеваются с помощью разных методов, и вдруг вам понадобилось запихнуть в этот объект свои данные, например логи.

Что вы будете делать — создадите новое поле объекта и присвоите ему свои логи. Но тут есть нюанс, если объект используется в большом количестве логики и другие разработчики как-то взаимодействуют с параметрами объекта, то есть шанс, что ваше новое поле где-то сможет сломать цикл, а где-то перезапишет собой другие поля.

Для решение этой проблемы, можно использовать symbol.

Как видно, поле, ключом которого является

symbol , нельзя обнаружить с помощью

методов и конструкций вроде for..in. Поэтому

новое свойство объекта не сломает приложе –

ние там, где используется логика похожая на

ту, что выше.

Ну окей. Представим, вы вдруг решили

добавить свойство \_\_my\_uniq\_logs\_\_, вместо

symbol, тогда смотрим что произойдёт дальше.

1. При вызове метода Object.keys(myObj) вернётся дополнительное поле — ваши логи (а Object.keys обычно используется для прохода по массиву ключей)

2. Новая не предвиденная переменная (ваши логи) в этом массиве ломает приложение

Так что, Symbol — ваш друг.

Что такое символы, а что нет.

Итак, у нас есть хороший обзор того, что такое символы и как они работают - но не менее важно знать, для чего символы подходят, а для чего нет, поскольку их легко можно принять за то, что они нет:

Символы никогда не будут конфликтовать с ключами строки объекта. Благодаря этому они отлично подходят для расширения объектов, которые вам предоставлены (например, в качестве параметра функции), не затрагивая объект заметным образом.

Символы нельзя прочитать с помощью существующих инструментов отражения. Вам нужен новый Object.getOwnPopertySymbols () для доступа к символам объекта, это делает символы отличными для хранения битов информации, которые вы не хотите, чтобы люди получали при нормальной работе. Использование Object.getOwnPropertySymbols () - довольно особенный вариант использования.

Символы не являются частными. Другое преимущество этого меча - все символы объекта можно получить с помощью Object.getOwnPropertySymbols () - не очень полезно для действительно частного значения. Не пытайтесь хранить в объекте информацию, которую вы хотите по-настоящему конфиденциально, с помощью символа - ее можно получить!

Перечислимые символы можно копировать в другие объекты с помощью новых методов, таких как Object.assign. Если вы попытаетесь вызвать Object.assign (newObject, objectWithSymbols), все (перечисляемые) символы во втором параметре (objectWithSymbols) будут скопированы в первый (newObject). Если вы не хотите, чтобы это произошло, сделайте их невычислимыми с помощью Object.defineProperty.

Символы не могут быть преобразованы в примитивы. Если вы попытаетесь привести символ к примитиву (+ Symbol (), '' + Symbol (), Symbol () + 'foo'), он выдаст ошибку. Это предотвращает случайное преобразование их в строку при установке в качестве имен свойств.

Символы не всегда уникальны. Как упоминалось выше, Symbol.for () возвращает неуникальный символ. Не думайте, что символ, который у вас есть, уникален, если только вы не сделали его сами.

10

По сути, это набор статических свойств класса Symbol, которые реализованы в других собственных объектах, таких как массивы, строки и во внутреннем устройстве движка JavaScript. Именно здесь происходит настоящая часть «Reflection within Implementation», поскольку эти хорошо известные символы изменяют поведение (то, что раньше было) внутренними компонентами JavaScript.

##### **Symbol.hasInstance: instanceof**

Symbol.hasInstance - это символ, который управляет поведением instanceof. Когда совместимый с ES6 движок видит оператор instanceof в выражении, он вызывает Symbol.hasInstance. Например,

lho instanceof rho вызовет rho [Symbol.hasInstance] (lho) (где rho - это правый операнд, а lho - левый операнд). Затем метод должен определить, наследуется ли он от этого конкретного экземпляра, вы можете реализовать это следующим образом:

##### **Symbol.iterator**

Если вы что-нибудь слышали о символах, то наверняка слышали о Symbol.iterator. В ES6 появился новый паттерн - цикл for of, который вызывает Symbol.iterator в правом операнде для получения значений для перебора. Другими словами, эти два эквивалента:

Symbol.iterator позволит вам переопределить оператор of - это означает, что если вы создадите библиотеку, которая его использует, разработчики полюбят вас:

Используется в for…of, …(spred), деструктуризации

##### **Symbol.isConcatSpreadable**

Symbol.isConcatSpreadable - довольно специфический символ, управляющий поведением Array # concat. Видите ли, Array # concat может принимать несколько аргументов, которые - если массивы - сами будут сглажены (или расширены) как часть операции concat. Рассмотрим следующий код

Начиная с ES6, Array # concat будет определять, распространяется ли какой-либо из его аргументов, используя Symbol.isConcatSpreadable. Это больше используется, чтобы сказать, что созданный вами класс, расширяющий Array, не будет особенно хорош для Array # concat, а не наоборот:

Symbol.unscopables

У этого символа довольно интересная история. По сути, при разработке ES6 TC обнаружил старый код в популярных библиотеках JS, который делал такие вещи:

Это хорошо работает в старом коде ES5 и ниже, но ES6 теперь имеет ключи Array # - то есть, когда вы используете (Array.prototype), keys теперь являются ключами метода Array #, а не переменной, которую вы установили. Итак, было три решения:

Попытайтесь заставить все сайты, использующие этот код, изменить его / обновить библиотеки (невозможно).

Удалите клавиши Array # и надейтесь, что другой подобной ошибки не возникнет (на самом деле это не решает проблему).

Напишите хитрость вокруг всего этого, которая предотвращает включение некоторых свойств в операторы.

Что ж, TC пошел с опцией 3, и так родился Symbol.unscopables, который определяет набор «не копируемых» значений в объекте, которые не должны устанавливаться при использовании внутри оператора with. Вам, вероятно, никогда не понадобится это использовать - и вы не столкнетесь с этим в повседневном сценарии JavaScripting, но он демонстрирует некоторые полезности символов, а также здесь для полноты:

#### Symbol.match

#### Это еще один символ, специфичный для функции. Функция соответствия String # теперь будет использовать это, чтобы определить, можно ли использовать данное значение для сопоставления с ним. Таким образом, вы можете предоставить свою собственную подходящую реализацию вместо регулярных выражений:

#### Symbol.replace

#### Как и Symbol.match, Symbol.replace был добавлен, чтобы разрешить настраиваемые классы, в которых вы обычно используете регулярные выражения, для String # replace:

#### Symbol.search

#### Ага, как и Symbol.match и Symbol.replace, Symbol.search существует для поддержки поиска String #, что позволяет использовать пользовательские классы вместо регулярных выражений:

#### Symbol.split

Хорошо, последний из символов String - Symbol.split предназначен для разделения String #. Используйте так:

#### Symbol.toPrimitive

Этот символ - самое близкое к перегрузке оператора абстрактного равенства (для краткости ==). По сути, Symbol.toPrimitive используется, когда движку JavaScript необходимо преобразовать ваш объект в примитивное значение - например, если вы сделаете + object, тогда JS вызовет объект [Symbol.toPrimitive] ('number');, если вы это сделаете '' + object ', тогда JS вызовет объект [Symbol.toPrimitive] (' string '), и если вы сделаете что-то вроде if (object), тогда он вызовет объект [Symbol.toPrimitive] (' default '). До этого у нас были valueOf и toString, с которыми можно было жонглировать - оба были довольно корявыми, и вы никогда не могли добиться от них желаемого поведения. Symbol.toPrimitive реализуется так:

#### Symbol.toStringTag

Хорошо, это последний из хорошо известных символов. Да ладно, вы зашли так далеко, вы можете это сделать! Symbol.toStringTag на самом деле довольно крутой - если вы когда-либо пытались реализовать свою собственную замену для оператора typeof, вы, вероятно, встречали Object # toString () - и то, как он возвращает этот странный '[object Object]' или строка '[массив объектов]'. До ES6 это поведение было определено в щелях спецификации, однако сегодня в причудливой стране ES6 у нас есть символ для этого! Любой объект, переданный в Object # toString (), будет проверен на предмет наличия у него свойства [Symbol.toStringTag], которое должно быть строкой, и если оно есть, то оно будет использоваться в сгенерированной строке - например:

В качестве дополнения к этому - если вы используете Chai для тестирования, теперь он использует символы под капотом для определения типа, поэтому вы можете написать expect (x) .to.be.a ('Collection') в своих тестах (при условии, что x имеет свойство Symbol.toStringTag, как указано выше, о, и что вы запускаете код в браузере с помощью Symbol.toStringTag).