2-го октября 2020

Катастрофический возврат

Некоторые регулярные выражения, простые с виду, могут выполняться ооооочень долго, и даже «подвешивать» интерпретатор JavaScript.

Рано или поздно с этим сталкивается любой разработчик, потому что нечаянно создать такое регулярное выражение –- проще простого.

Типичный симптом: регулярное выражение обычно работает нормально, но иногда, с некоторыми строками, «подвешивает» интерпретатор и потребляет 100% процессора.

Как правило, веб-браузер при этом предлагает «убить» скрипт и перезагрузить зависшую страницу. Явно плохая ситуация.

Ну а для серверного JavaScript это может стать серьёзной уязвимостью, если регулярные выражения используются для обработки пользовательских данных.

Пример

Допустим, у нас есть строка, и мы хотим проверить, что она состоит из слов `\w+`, после каждого слова может быть пробел `\s?`.

Используем регулярное выражение `^(\w+\s?)*$`, которое задаёт 0 или более таких слов.

Проверим, чтобы убедиться, что оно работает:

```
let regexp = /^(\w+\s?)*$/;

alert( regexp.test("A good string") ); // true
alert( regexp.test("Bad characters: $@#") ); // false
```

Результат верный. Однако, на некоторых строках оно выполняется очень долго. Так долго, что интерпретатор JavaScript «зависает» с потреблением 100% процессора.

Если вы запустите пример ниже, то, скорее всего, ничего не увидите, так как JavaScript «подвиснет». В браузере он перестанет реагировать на другие события и, скорее всего, понадобится перезагрузить страницу, так что осторожно с этим:

```
let regexp = /^(\w+\s?)*$/;
let str = "An input string that takes a long time or ev
// этот поиск будет выполняться очень, очень долго
alert( regexp.test(str) );
```

Некоторые движки регулярных выражений могут справиться с таким поиском, но большинство из них – нет.

Упрощённый пример

В чём же дело? Почему регулярное выражение «зависает»?

Чтобы это понять, упростим пример: уберём из него пробелы `\s?`. Получится `^(\w+)*$`.

И, для большей наглядности, заменим `\w` на `\d`. Получившееся регулярное выражение тоже будет «зависать», например:

```
let regexp = /^(\d+)*$/;

```

```
let str = "0123456789012345678901234567890123456789!";

// этот поиск будет выполняться очень, очень долго
alert( regexp.test(str) );
```

В чём же дело, что не так с регулярным выражением?

Внимательный читатель, посмотрев на `(\d+)*`, наверняка удивится, ведь оно какое-то странное. Квантификатор `*` здесь выглядит лишним. Если хочется найти число, то с тем же успехом можно искать `\d+`.

Действительно, это регулярное выражение носит искусственный характер, но, разобравшись с ним, мы поймём и практический пример, данный выше. Причина их медленной работы одинакова. Поэтому оставим как есть.

Что же происходит во время поиска `^(\d+)*$` в строке `123456789!` (укоротим для ясности), почему всё так долго?

1. Первым делом, движок регулярных выражений пытается найти `\d+`. Плюс `+` является жадным по умолчанию, так что он хватает все цифры, какие может:

   ```
   \d+.......
   (123456789)!
   ```

   Затем движок пытается применить квантификатор `*`, но больше цифр нет, так что звёздочка ничего не даёт.

   Далее по шаблону ожидается конец строки `$`, а в тексте символ `!`, так что соответствий нет:

   ```
              X
   \d+........$
   (123456789)!
   ```

2. Так как соответствие не найдено, то «жадный» квантификатор `+` уменьшает количество повторений, возвращается на один символ назад.

   Теперь `\d+` – это все цифры, за исключением последней:

   ```
   \d+.......
   (12345678)9!
   ```

3. Далее движок снова пытается продолжить поиск, начиная уже с позиции `(9)`.

   Звёздочка `(\d+)*` теперь может быть применена – она даёт второе число `9`:

   ```
   \d+.......\d+
   (12345678)(9)!
   ```

   Затем движок ожидает найти `$`, но это ему не удаётся, ведь строка оканчивается на `!`:

   ```
               X
   \d+.......\d+
   (12345678)(9)!
   ```

4. Так как совпадения нет, то поисковый движок продолжает отступать назад. Общее правило таково: последний жадный квантификатор уменьшает количество повторений до тех пор, пока это возможно. Затем понижается предыдущий «жадный» квантификатор и т.д.

   Перебираются все возможные комбинации. Вот их примеры.

Когда первое число `\d+` содержит 7 цифр, а дальше число из 2 цифр:

```
1              X
2  \d+......\d+
3  (1234567)(89)!
```

Когда первое число содержит 7 цифр, а дальше два числа по 1 цифре:

```
1                X
2  \d+......\d+\d+
3  (1234567)(8)(9)!
```

Когда первое число содержит 6 цифр, а дальше одно число из 3 цифр:

```
1              X
2  \d+.......\d+
3  (123456)(789)!
```

Когда первое число содержит 6 цифр, а затем два числа:

```
1                X
2  \d+.....\d+ \d+
3  (123456)(78)(9)!
```

...И так далее.

Существует много способов как разбить на числа набор цифр `123456789`. Если быть точным, их 2^n-1, где `n` – длина набора.

В случае `n=20` их порядка миллиона, при `n=30` – ещё в тысячу раз больше. На их перебор и тратится время.

Что же делать?

Может нам стоит использовать «ленивый» режим?

К сожалению, нет: если мы заменим `\d+` на `\d+?`, то регулярное выражение всё ещё будет «зависать». Поменяется только порядок перебора, но не общее количество комбинаций.

Некоторые движки регулярных выражений содержат хитрые проверки и конечные автоматы, которые позволяют избежать полного перебора в таких ситуациях или кардинально ускорить его, но не все движки и не всегда.

Назад к словам и строкам

В начальном примере, когда мы ищем слова по шаблону `^(\w+\s?)*$` в строке вида `An input that hangs!`, происходит то же самое.

Дело в том, что каждое слово может быть представлено как в виде одного `\w+`, так и нескольких:

```
1  (input)
2  (inpu)(t)
3  (inp)(u)(t)
4  (in)(p)(ut)
5  ...
```

Человеку очевидно, что совпадения быть не может, так как эта строка заканчивается на восклицательный знак `!`, а по регулярному выражению в конце должен быть символ `\w` или пробел `\s`. Но движок этого не знает.

Он перебирает все комбинации того, как регулярное выражение `(\w+\s?)*` может «захватить» каждое слово, включая варианты как с пробелами `(\w+\s)*`, так и без `(\w+)*` (пробелы `\s?` ведь не обязательны). Этих вариантов очень много, отсюда и сверхдолгое время выполнения.

Как исправить?

Есть два основных подхода, как это исправить.

Первый – уменьшить количество возможных комбинаций.

Перепишем регулярное выражение так: `^(\w+\s)*\w*` – то есть, будем искать любое количество слов с пробелом `(\w+\s)*`, после которых идёт (но не обязательно) обычное слово `\w*`.

Это регулярное выражение эквивалентно предыдущему (ищет то же самое), и на этот раз всё работает:

```
let regexp = /^(\w+\s)*\w*$/;
let str = "An input string that takes a long time or ev
alert( regexp.test(str) ); // false
```

Почему же проблема исчезла?

Теперь звёздочка `*` стоит после `\w+\s` вместо `\w+\s?`. Стало невозможно разбить одно слово на несколько разных `\w+`. Исчезли и потери времени на перебор таких комбинаций.

Например, с предыдущим шаблоном `(\w+\s?)*` слово `string` могло быть представлено как два подряд `\w+`:

```
\w+\w+
string
```

Предыдущий шаблон из-за необязательности `\s` допускал варианты `\w+`, `\w+\s`, `\w+\w+` и т.п.

С переписанным шаблоном `(\w+\s)*`, такое невозможно: может быть `\w+\s` или `\w+\s\w+\s`, но не `\w+\w+`. Так что общее количество комбинаций сильно уменьшается.

Запрет возврата

Переписывать регулярное выражение не всегда удобно, и не всегда очевидно, как это сделать.

Альтернативный подход заключается в том, чтобы запретить возврат для квантификатора.

Движок регулярных выражений проверяет множество вариантов, которые для человека являются очевидно ошибочными.

Например, в шаблоне `(\d+)*$` для человека очевидно, что в `(\d+)*` не нужно «откатывать» `+`. От того, что вместо одного `\d+` у нас будет два независимых `\d+\d+`, ничего не изменится:

```
\d+........
(123456789)!

\d+...\d+....
(1234)(56789)!
```

Если говорить об изначальном примере `^(\w+\s?)*$`, то хорошо бы исключить возврат для `\w+`. То есть, для `\w+` нужно искать только одно слово целиком, максимально возможной длины. Не нужно уменьшать количество повторений `\w+`, пробовать разбить слово на два `\w+\w+`, и т.п.

В современных регулярных выражениях для решения этой проблемы придумали захватывающие (possessive) квантификаторы, которые такие же как жадные, но не делают возврат (то есть, по сути, они даже проще, чем жадные).

Также есть «атомарные скобочные группы» – средство, запрещающее возврат внутри скобок.

К сожалению, в JavaScript они не поддерживаются, но есть другое средство.

Опережающая проверка в помощь!

Мы можем исключить возврат с помощью опережающей проверки.

Шаблон, захватывающий максимальное количество повторений `\w` без возврата, выглядит так: `(?=(\w+))\1`.

Расшифруем его:

- Опережающая проверка `?=` ищет максимальное количество `\w+`, доступных с текущей позиции.
- Содержимое скобок вокруг `?=...` не запоминается движком, поэтому оборачиваем `\w+` внутри в дополнительные скобки, чтобы движок регулярных выражений запомнил их содержимое.
- ...И чтобы далее в шаблоне на него сослаться обратной ссылкой `\1`.

То есть, мы смотрим вперед – и если там есть слово `\w+`, то ищем его же `\1`.

Зачем? Всё дело в том, что опережающая проверка находит слово `\w+` целиком, и мы захватываем его в шаблон посредством `\1`. Поэтому мы реализовали, по сути, захватывающий квантификатор `+`. Такой шаблон захватывает только полностью слово `\w+`, не его часть.

Например, в слове `JavaScript` он не может захватить только `Java`, и оставить `Script` для совпадения с остатком шаблона.

Вот, посмотрите, сравнение двух шаблонов:

```
1  alert( "JavaScript".match(/\w+Script/)); // JavaScript
2  alert( "JavaScript".match(/(?=(\w+))\1Script/)); // nul
```

1. В первом варианте `\w+` сначала забирает слово `JavaScript` целиком, потом `+` постепенно отступает, чтобы попробовать найти оставшуюся часть шаблона, и в конце концов находит (при этом `\w+` будет соответствовать `Java`).
2. Во втором варианте `(?=(\w+))` осуществляет опережающую проверку и видит сразу слово `JavaScript`, которое `\1` целиком захватывает в совпадение, так что уже нет возможности найти `Script`.

Внутрь `(?=(\w+))\1` можно вместо `\w` вставить и более сложное регулярное выражение, при поиске которого квантификатор `+` не должен делать возврат.

> **На заметку:**
>
> Больше о связи захватывающих квантификаторов и опережающей проверки вы можете найти в статьях Regex: Emulate Atomic Grouping (and Possessive Quantifiers) with LookAhead и Mimicking Atomic Groups.

Перепишем исходный пример, используя опережающую проверку для запрета возврата:

```
1  let regexp = /^((?=(\w+))\2\s?)*$/;
2
3  alert( regexp.test("A good string") ); // true
4
5  let str = "An input string that takes a long time or ev
6
7  alert( regexp.test(str) ); // false, работает и быстро
```

Здесь внутри скобок стоит `\2` вместо `\1`, так как есть ещё внешние скобки. Чтобы избежать путаницы с номерами скобок, можно дать скобкам имя, например `(?<word>\w+)`.


```
// скобки названы ?<word>, ссылка на них \k<word>
let regexp = /^((?=(?<word>\w+))\k<word>\s?)*$/;

let str = "An input string that takes a long time or ev

alert( regexp.test(str) ); // false

alert( regexp.test("A correct string") ); // true
```

Проблему, которой была посвящена эта глава, называют «катастрофический возврат» (catastrophic backtracking).

Мы разобрали два способа её решения:

- Уменьшение возможных комбинаций переписыванием шаблона.
- Запрет возврата.

Комментарии

перед тем как писать...