6th September 2019

Catastrophic backtracking

Some regular expressions are looking simple, but can execute veeeeeery long time, and even “hang” the JavaScript engine.

Sooner or later most developers occasionally face such behavior, because it’s quite easy to create such a regexp.

The typical symptom – a regular expression works fine sometimes, but for certain strings it “hangs”, consuming 100% of CPU.

In such case a web-browser suggests to kill the script and reload the page. Not a good thing for sure.

For server-side JavaScript it may become a vulnerability if regular expressions process user data.

Example

Let’s say we have a string, and we’d like to check if it consists of words `\w+` with an optional space `\s?` after each.

We’ll use a regexp `^(\w+\s?)*$`, it specifies 0 or more such words.

In action:

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

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

It seems to work. The result is correct. Although, on certain strings it takes a lot of time. So long that JavaScript engine “hangs” with 100% CPU consumption.

If you run the example below, you probably won’t see anything, as JavaScript will just “hang”. A web-browser will stop reacting on events, the UI will stop working. After some time it will suggest to reloaad the page. So be careful with this:

```
let regexp = /^(\w+\s?)*$/;
let str = "An input string that takes a long time or even makes this regexp t

// will take a very long time
alert( regexp.test(str) );
```

Some regular expression engines can handle such search, but most of them can’t.

Simplified example

What's the matter? Why the regular expression "hangs"?

To understand that, let's simplify the example: remove spaces `\s?`. Then it becomes `^(\w+)*$`.

And, to make things more obvious, let's replace `\w` with `\d`. The resulting regular expression still hangs, for instance:

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

let str = "012345678901234567890123456789!";

// will take a very long time
alert( regexp.test(str) );
```

So what's wrong with the regexp?

First, one may notice that the regexp `(\d+)*` is a little bit strange. The quantifier `*` looks extraneous. If we want a number, we can use `\d+`.

Indeed, the regexp is artificial. But the reason why it is slow is the same as those we saw above. So let's understand it, and then the previous example will become obvious.

What happens during the search of `^(\d+)*$` in the line `123456789!` (shortened a bit for clarity), why does it take so long?

1. First, the regexp engine tries to find a number `\d+`. The plus `+` is greedy by default, so it consumes all digits:

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

   Then it tries to apply the star quantifier, but there are no more digits, so it the star doesn't give anything.

   The next in the pattern is the string end `$`, but in the text we have `!`, so there's no match:

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

2. As there's no match, the greedy quantifier `+` decreases the count of repetitions, backtracks one character back.

   Now `\d+` takes all digits except the last one:

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

3. Then the engine tries to continue the search from the new position (9).

The star `(\d+)*` can be applied – it gives the number `9`:

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

The engine tries to match `$` again, but fails, because meets `!`:

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

4. There’s no match, so the engine will continue backtracking, decreasing the number of repetitions. Backtracking generally works like this: the last greedy quantifier decreases the number of repetitions until it can. Then the previous greedy quantifier decreases, and so on.

   All possible combinations are attempted. Here are their examples.

   The first number `\d+` has 7 digits, and then a number of 2 digits:

   ```
                X
   \d+......\d+
   (1234567)(89)!
   ```

   The first number has 7 digits, and then two numbers of 1 digit each:

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

   The first number has 6 digits, and then a number of 3 digits:

   ```
                X
   \d+.......\d+
   (123456)(789)!
   ```

   The first number has 6 digits, and then 2 numbers:

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

   …And so on.

There are many ways to split a set of digits `123456789` into numbers. To be precise, there are `2^n-1`, where `n` is the length of the set.

For `n=20` there are about 1 million combinations, for `n=30` – a thousand times more. Trying each of them is exactly the reason why the search takes so long.

What to do?

Should we turn on the lazy mode?

Unfortunately, that won't help: if we replace `\d+` with `\d+?`, the regexp will still hang. The order of combinations will change, but not their total count.

Some regular expression engines have tricky tests and finite automations that allow to avoid going through all combinations or make it much faster, but not all engines, and not in all cases.

Back to words and strings

The similar thing happens in our first example, when we look words by pattern `^(\w+\s?)*$` in the string `An input that hangs!`.

The reason is that a word can be represented as one `\w+` or many:

```
(input)
(inpu)(t)
(inp)(u)(t)
(in)(p)(ut)
...
```

For a human, it's obvious that there may be no match, because the string ends with an exclamation sign `!`, but the regular expression expects a wordly character `\w` or a space `\s` at the end. But the engine doesn't know that.

It tries all combinations of how the regexp `(\w+\s?)*` can "consume" the string, including variants with spaces `(\w+\s)*` and without them `(\w+)*` (because spaces `\s?` are optional). As there are many such combinations, the search takes a lot of time.

How to fix?

There are two main approaches to fixing the problem.

The first is to lower the number of possible combinations.

Let's rewrite the regular expression as `^(\w+\s)*\w*` – we'll look for any number of words followed by a space `(\w+\s)*`, and then (optionally) a word `\w*`.

This regexp is equivalent to the previous one (matches the same) and works well:

```
let regexp = /^(\w+\s)*\w*$/;
let str = "An input string that takes a long time or even makes this regex to

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

Why did the problem disappear?

Now the star `*` goes after `\w+\s` instead of `\w+\s?`. It became impossible to represent one word of the string with multiple successive `\w+`. The time needed to try such combinations is now saved.

For example, the previous pattern `(\w+\s?)*` could match the word `string` as two `\w+`:

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

The previous pattern, due to the optional `\s` allowed variants `\w+`, `\w+\s`, `\w+\w+` and so on.

With the rewritten pattern `(\w+\s)*`, that's impossible: there may be `\w+\s` or `\w+\s\w+\s`, but not `\w+\w+`. So the overall combinations count is greatly decreased.

Preventing backtracking

It's not always convenient to rewrite a regexp. And it's not always obvious how to do it.

The alternative approach is to forbid backtracking for the quantifier.

The regular expressions engine tries many combinations that are obviously wrong for a human.

E.g. in the regexp `(\d+)*$` it's obvious for a human, that `+` shouldn't backtrack. If we replace one `\d+` with two separate `\d+\d+`, nothing changes:

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

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

And in the original example `^(\w+\s?)*$` we may want to forbid backtracking in `\w+`. That is: `\w+` should match a whole word, with the maximal possible length. There's no need to lower the repetitions count in `\w+`, try to split it into two words `\w+\w+` and so on.

Modern regular expression engines support possessive quantifiers for that. They are like greedy ones, but don't backtrack (so they are actually simpler than regular quantifiers).

There are also so-called "atomic capturing groups" – a way to disable backtracking inside parentheses.

Unfortunately, in JavaScript they are not supported. But there's another way.

Lookahead to the rescue!

We can prevent backtracking using lookahead.

The pattern to take as much repetitions of `\w` as possible without backtracking is: `(?=(\w+))\1`.

Let's decipher it:

- Lookahead `?=` looks forward for the longest word `\w+` starting at the current position.
- The contents of parentheses with `?=...` isn't memorized by the engine, so wrap `\w+` into parentheses. Then the engine will memorize their contents
- …And allow us to reference it in the pattern as `\1`.

That is: we look ahead – and if there's a word `\w+`, then match it as `\1`.

Why? That's because the lookahead finds a word `\w+` as a whole and we capture it into the pattern with `\1`. So we essentially implemented a possessive plus `+` quantifier. It captures only the whole word `\w+`, not a part of it.

For instance, in the word `JavaScript` it may not only match `Java`, but leave out `Script` to match the rest of the pattern.

Here's the comparison of two patterns:

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

1. In the first variant `\w+` first captures the whole word `JavaScript` but then `+` backtracks character by character, to try to match the rest of the pattern, until it finally succeeds (when `\w+` matches `Java`).
2. In the second variant `(?=(\w+))` looks ahead and finds the word `JavaScript`, that is included into the pattern as a whole by `\1`, so there remains no way to find `Script` after it.

We can put a more complex regular expression into `(?=(\w+))\1` instead of `\w`, when we need to forbid backtracking for `+` after it.

Please note:

There's more about the relation between possessive quantifiers and lookahead in articles Regex: Emulate Atomic Grouping (and Possessive Quantifiers) with LookAhead and Mimicking Atomic Groups.

Let's rewrite the first example using lookahead to prevent backtracking:

```
let regexp = /^((?=(\w+))\2\s?)*$/;

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

let str = "An input string that takes a long time or even makes this regex to

alert( regexp.test(str) ); // false, works and fast!
```

Here `\2` is used instead of `\1`, because there are additional outer parentheses. To avoid messing up with the numbers, we can give the parentheses a name, e.g. `(?<word>\w+)`.

```
// parentheses are named ?<word>, referenced as \k<word>
let regexp = /^((?=(?<word>\w+))\k<word>\s?)*$/;

let str = "An input string that takes a long time or even makes this regex to

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

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

The problem described in this article is called “catastrophic backtracking”.

We covered two ways how to solve it:

- Rewrite the regexp to lower the possible combinations count.
- Prevent backtracking.

Previous lesson Next lesson

Share Tutorial map

Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen…)