We are no longer accepting contributions to Documentation. Please see our post on meta.

Greedy and Lazy quantifiers

Examples

Greediness versus Laziness

32

Given the following input:

```
aaaaaAlazyZgreeedyAlaaazyZaaaaa
```

We will use two patterns: one greedy: `A.*Z`, and one lazy: `A.*?Z`. These patterns yield the following matches:

- `A.*Z` yields 1 match: `AlazyZgreeedyAlaaazyZ` (examples: Regex101, Rubular)
- `A.*?Z` yields 2 matches: `AlazyZ` and `AlaaazyZ` (examples: Regex101, Rubular)

First focus on what `A.*Z` does. When it matched the first `A`, the `.*`, being greedy, then tries to match as many `.` as possible.

```
aaaaaAlazyZgreeedyAlaaazyZaaaaa
     \_____________________/
      A.* matched, Z can't match
```

Since the `Z` doesn't match, the engine backtracks, and `.*` must then match one fewer `.`:

```
aaaaaAlazyZgreeedyAlaaazyZaaaaa
     \____________________/
      A.* matched, Z can't match
```

This happens a few more times, until it finally comes to this:

```
aaaaaAlazyZgreeedyAlaaazyZaaaaa
     \_________________/
      A.* matched, Z can now match
```

Now `Z` can match, so the overall pattern matches:

```
aaaaaAlazyZgreeedyAlaaazyZaaaaa
     \_________________/
      A.*Z matched
```

By contrast, the reluctant (lazy) repetition in `A.*?Z` first matches as few `.` as possible, and then taking more `.` as necessary. This explains why it finds two matches in the input.

Here's a visual representation of what the two patterns matched:

```
aaaaaAlazyZgreeedyAlaaazyZaaaaa
     \____/l      \_____/l      l = lazy
     \_________g_______/       g = greedy
```

Example based on answer made by polygenelubricants.

The POSIX standard does not include the `?` operator, so many POSIX regex engines do not have lazy matching. While refactoring, especially with the "greatest trick ever", may help match in some cases, the only way to have true lazy matching is to use an engine that supports it.

discuss

edited Sep 11 '16 at 5:23 +2

Boundaries with multiple matches

2

When you have an input with well defined boundaries and are expecting more than one match in your string, you have two options:

- Using lazy quantifiers;
- Using a negated character class.

Consider the following:

You have a simple templating engine, you want to replace substrings like `$[foo]` where `foo` can be any string. You want to replace this substring with whatever based on the part between the `[]`.

You can try something like `\$\[(.*)\]`, and then use the first capture group.

The problem with this is if you have a string like `something $[foo] lalala $[bar] something else` your match will be

```
something $[foo] lalala $[bar] something else
          | \_____CG1_____/|
          \_______Match_____/
```

The capture group being `foo] lalala $[bar` which may or may not be valid.

You have two solutions

1. Using laziness: In this case making `*` lazy is one way to go about finding the right things. So you change your expression to `\$\[(.*?)\]`
2. Using negated character class : `[^\]]` you change your expression to `\$\[([^\]]*)\]`.

In both solutions, the result will be the same:

```
something $[foo] lalala $[bar] something else
          | \_/|       | \_/|
          \____/       \____/
```

With the capture group being respectively `foo` and `bar`.

Using negated character class reduces backtracking issue and may save your CPU a lot of time when it comes to large inputs.

discuss

edited Mar 10 at 23:05

Syntax

Parameters

Quantifiers	Description
`?`	Match the preceding character or subexpression 0 or 1 times (preferably 1).
`*`	Match the preceding character or subexpression 0 or more times (as many as possible).
`+`	Match the preceding character or subexpression 1 or more times (as many as possible).
`{n}`	Match the preceding character or subexpression exactly *n* times.
`{min,}`	Match the preceding character or subexpression *min* or more times (as many as possible).
`{0,max}`	Match the preceding character or subexpression *max* or fewer times (as close to *max* as possible).
`{min,max}`	Match the preceding character or subexpression at least *min* times but no more than *max* times (as close to *max* as possible).
Lazy Quantifiers	**Description**
`??`	Match the preceding character or subexpression 0 or 1 times (preferably 0).
`*?`	Match the preceding character or subexpression 0 or more times (as few as possible).
`+?`	Match the preceding character or subexpression 1 or more times (as few as possible).
`{n}?`	Match the preceding character or subexpression exactly *n* times. No difference between greedy and lazy version.
`{min,}?`	Match the preceding character or subexpression *min* or more times (as close to *min* as possible).
`{0,max}?`	Match the preceding character or subexpression *max* or fewer times (as few as possible).
`{min,max}?`	Match the preceding character or subexpression at least *min* times but no more than *max* times (as close to *min* as possible).

Remarks

Greediness

A *greedy* quantifier always attempts to repeat the sub-pattern as many times as possible before exploring shorter matches by backtracking.

Generally, a greedy pattern will match the longest possible string.

By default, all quantifiers are greedy.

Laziness

A *lazy* (also called *non-greedy* or *reluctant*) quantifier always attempts to repeat the sub-pattern as *few* times as possible, before exploring longer matches by expansion.

Generally, a lazy pattern will match the shortest possible string.

To make quantifiers lazy, just append `?` to the existing quantifier, e.g. `+?`, `{0,5}?`.

Concept of greediness and laziness only exists in backtracking engines

The notion of greedy/lazy quantifier only exists in backtracking regex engines. In non-backtracking regex engines or POSIX-compliant regex engines, quantifiers only specify the upper bound and lower bound of the repetition, without specifying how to find the match -- those engines will always match the left-most longest string regardless.

edited Mar 21 at 20:44 +3

Still have a question about Greedy and Lazy quantifiers? Ask Question