Possessive Quantifiers

The topic on repetition operators or quantifiers explains the difference between greedy and lazy repetition. Greediness and laziness determine the order in which the regex engine tries the possible permutations of the regex pattern. A greedy quantifier first tries to repeat the token as many times as possible, and gradually gives up matches as the engine backtracks to find an overall match. A lazy quantifier first repeats the token as few times as required, and gradually expands the match as the engine backtracks through the regex to find an overall match.

Because greediness and laziness change the order in which permutations are tried, they can change the overall regex match. However, they do not change the fact that the regex engine will backtrack to try all possible permutations of the regular expression in case no match can be found.

Possessive quantifiers are a way to prevent the regex engine from trying all permutations. This is primarily useful for performance reasons. You can also use possessive quantifiers to eliminate certain matches.

Of the regex flavors discussed in this tutorial, possessive quantifiers are supported by JGsoft, Java, and PCRE. That includes languages with regex support based on PCRE such as PHP, Delphi, and R. Ruby supports possessive quantifiers starting with Ruby 1.9, Perl supports them starting with Perl 5.10, and Boost starting with Boost 1.42.

How Possessive Quantifiers Work

Like a greedy quantifier, a possessive quantifier repeats the token as many times as possible. Unlike a greedy quantifier, it does *not* give up matches as the engine backtracks. With a possessive quantifier, the deal is all or nothing. You can make a quantifier possessive by placing an extra + after it. * is greedy, *? is lazy, and *+ is possessive. ++, ?+ and `{n,m}`+ are all possessive as well.

Let’s see what happens if we try to match `"[^"]*+"` against `"abc"`. The `"` matches the `"`. `[^"]` matches `a`, `b` and `c` as it is repeated by the star. The final `"` then matches the final `"` and we found an overall match. In this case, the end result is the same, whether we use a greedy or possessive quantifier. There is a slight performance increase though, because the possessive quantifier doesn’t have to remember any backtracking positions.

The performance increase can be significant in situations where the regex fails. If the subject is `"abc` (no closing quote), the above matching process happens in the same way, except that the second `"` fails. When using a possessive quantifier, there are no steps to backtrack to. The regular expression does not have any alternation or non-possessive quantifiers that can give up part of their match to try a different permutation of the regular expression. So the match attempt fails immediately when the second `"` fails.

Had we used `"[^"]*"` with a greedy quantifier instead, the engine would have backtracked. After the `"` failed at the end of the string, the `[^"]*` would give up one match, leaving it with `ab`. The `"` would then fail to match `c`. `[^"]*` backtracks to just `a`, and `"` fails to match `b`. Finally, `[^"]*` backtracks to match zero characters, and `"` fails `a`. Only at this point have all backtracking positions been exhausted, and does the engine give up the match attempt. Essentially, this regex performs as many needless steps as there are characters following the unmatched opening quote.

When Possessive Quantifiers Matter

The main practical benefit of possessive quantifiers is to speed up your regular expression. In particular, possessive quantifiers allow your regex to fail faster. In the above example, when the closing quote fails to match, we *know* the regular expression couldn’t possibly have skipped over a quote. So there’s no need to backtrack and check for the quote. We make the regex engine aware of this by making the quantifier possessive. In fact, some engines, including the JGsoft engine, detect that `[^"]*` and `"` are mutually exclusive when compiling your regular expression, and automatically make the star possessive.

Now, linear backtracking like a regex with a single quantifier does is pretty fast. It's unlikely you'll notice the speed difference. However, when you're nesting quantifiers, a possessive quantifier may save your day. Nesting quantifiers means that you have one or more repeated tokens inside a group, and the group is also repeated. That's when catastrophic backtracking often rears its ugly head. In such cases, you'll depend on possessive quantifiers and/or atomic grouping to save the day.

Possessive Quantifiers Can Change The Match Result

Using possessive quantifiers can change the result of a match attempt. Since no backtracking is done, and matches that would require a greedy quantifier to backtrack will not be found with a possessive quantifier. For example, `".*"` matches `"abc"` in `"abc"x`, but `".*+"` does not match this string at all.

In both regular expressions, the first `"` matches the first `"` in the string. The repeated dot then matches the remainder of the string `abc"x`. The second `"` then fails to match at the end of the string.

Now, the paths of the two regular expressions diverge. The possessive dot-star wants it all. No backtracking is done. Since the `"` failed, there are no permutations left to try, and the overall match attempt fails. The greedy dot-star, while initially grabbing everything, is willing to give back. It will backtrack one character at a time. Backtracking to `abc"`, `"` fails to match `x`. Backtracking to `abc`, `"` matches `"`. An overall match `"abc"` is found.

Essentially, the lesson here is that when using possessive quantifiers, you need to make sure that whatever you're applying the possessive quantifier to should not be able to match what should follow it. The problem in the above example is that the dot also matches the closing quote. This prevents us from using a possessive quantifier. The negated character class in the previous section cannot match the closing quote, so we can make it possessive.

Using Atomic Grouping Instead of Possessive Quantifiers

Technically, possessive quantifiers are a notational convenience to place an atomic group around a single quantifier. All regex flavors that support possessive quantifiers also support atomic grouping. But not all regex flavors that support atomic grouping support possessive quantifiers. With those flavors, you can achieve the exact same results using an atomic group.

Basically, instead of `X*+`, write `(?>X*)`. It is important to notice that both the quantified token X and the quantifier are inside the atomic group. Even if X is a group, you still need to put an extra atomic group around it to achieve the same effect. `(?:a|b)*+` is equivalent to `(?>(?:a|b)*)` but not to `(?>a|b)*`. The latter is a valid regular expression, but it won't have the same effect when used as part of a larger regular expression.

To illustrate, `(?:a|b)*+b` and `(?>(?:a|b)*)b` both fail to match `b`. `a|b` matches the `b`. The star is satisfied, and the fact that it's possessive or the atomic group will cause the star to forget all its backtracking positions. The second `b` in the regex has nothing left to match, and the overall match attempt fails.

In the regex `(?>a|b)*b`, the atomic group forces the alternation to give up its backtracking positions. This means that if an `a` is matched, it won't come back to try `b` if the rest of the regex fails. Since the star is outside of the group, it is a normal, greedy star. When the second `b` fails, the greedy star backtracks to zero iterations. Then, the second `b` matches the `b` in the subject string.

This distinction is particularly important when converting a regular expression written by somebody else using possessive quantifiers to a regex flavor that doesn't have possessive quantifiers. You could, of course, let a tool like RegexBuddy do the conversion for you.

Make a Donation

Did this website just save you a trip to the bookstore? Please make a donation to support this site, and you'll get a **lifetime of advertisement-free access** to this site!