Atomic Grouping

An atomic group is a group that, when the regex engine exits from it, automatically throws away all backtracking positions remembered by any tokens inside the group. Atomic groups are non-capturing. The syntax is `(?>group)`. Lookaround groups are also atomic. Atomic grouping is supported by most modern regular expression flavors, including the JGsoft flavor, Java, PCRE, .NET, Perl, Boost, and Ruby. Most of these also support possessive quantifiers, which are essentially a notational convenience for atomic grouping.

An example will make the behavior of atomic groups clear. The regular expression `a(bc|b)c` (capturing group) matches `abcc` and `abc`. The regex `a(?>bc|b)c` (atomic group) matches `abcc` but not `abc`.

When applied to `abc`, both regexes will match `a` to `a`, `bc` to `bc`, and then `c` will fail to match at the end of the string. Here their paths diverge. The regex with the capturing group has remembered a backtracking position for the alternation. The group will give up its match, `b` then matches `b` and `c` matches `c`. Match found!

The regex with the atomic group, however, exited from an atomic group after `bc` was matched. At that point, all backtracking positions for tokens inside the group are discarded. In this example, the alternation's option to try `b` at the second position in the string is discarded. As a result, when `c` fails, the regex engine has no alternatives left to try.

Of course, the above example isn't very useful. But it does illustrate very clearly how atomic grouping eliminates certain matches. Or more importantly, it eliminates certain match attempts.

Regex Optimization Using Atomic Grouping

Consider the regex `\b(integer|insert|in)\b` and the subject `integers`. Obviously, because of the word boundaries, these don't match. What's not so obvious is that the regex engine will spend quite some effort figuring this out.

`\b` matches at the start of the string, and `integer` matches `integer`. The regex engine makes note that there are two more alternatives in the group, and continues with `\b`. This fails to match between the `r` and `s`. So the engine backtracks to try the second alternative inside the group. The second alternative matches `in`, but then fails to match `s`. So the engine backtracks once more to the third alternative. `in` matches `in`. `\b` fails between the `n` and `t` this time. The regex engine has no more remembered backtracking positions, so it declares failure.

This is quite a lot of work to figure out `integers` isn't in our list of words. We can optimize this by telling the regular expression engine that if it can't match `\b` after it matched `integer`, then it shouldn't bother trying any of the other words. The word we've encountered in the subject string is a longer word, and it isn't in our list.

We can do this by turning the capturing group into an atomic group: `\b(?>integer|insert|in)\b`. Now, when `integer` matches, the engine exits from an atomic group, and throws away the backtracking positions it stored for the alternation. When `\b` fails, the engine gives up immediately. This savings can be significant when scanning a large file for a long list of keywords. This savings will be vital when your alternatives contain repeated tokens (not to mention repeated groups) that lead to catastrophic backtracking.

Don't be too quick to make all your groups atomic. As we saw in the first example above, atomic grouping can exclude valid matches too. Compare how `\b(?>integer|insert|in)\b` and `\b(?>in|integer|insert)\b` behave when applied to `insert`. The former regex matches, while the latter fails. If the groups weren't atomic, both regexes would match. Remember that alternation tries its alternatives from left to right. If the second regex matches `in`, it won't try the two other alternatives due to the atomic group.

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!
