Matching Nested Constructs with Balancing Groups

The .NET regex flavor has a special feature called balancing groups. The main purpose of balancing groups is to match balanced constructs or nested constructs, which is where they get their name from. A technically more accurate name for the feature would be capturing group subtraction. That's what the feature really does. It's .NET's solution to a problem that other regex flavors like Perl, PCRE, and Ruby handle with regular expression recursion. JGsoft V2 supports both balancing groups and recursion.

`(?<capture-subtract>regex)` or `(?'capture-subtract'regex)` is the basic syntax of a balancing group. It's the same syntax used for named capturing groups in .NET but with two group names delimited by a minus sign. The name of this group is "capture". You can omit the name of the group. `(?<-subtract>regex)` or `(?'-subtract'regex)` is the syntax for a non-capturing balancing group.

The name "subtract" must be the name of another group in the regex. When the regex engine enters the balancing group, it subtracts one match from the group "subtract". If the group "subtract" did not match yet, or if all its matches were already subtracted, then the balancing group fails to match. You could think of a balancing group as a conditional that tests the group "subtract", with "regex" as the "if" part and an "else" part that always fails to match. The difference is that the balancing group has the added feature of subtracting one match from the group "subtract", while a conditional leaves the group untouched.

If the balancing group succeeds and it has a name ("capture" in this example), then the group captures the text between the end of the match that was subtracted from the group "subtract" and the start of the match of the balancing group itself ("regex" in this example).

The reason this works in .NET is that capturing groups in .NET keep a stack of everything they captured during the matching process that wasn't backtracked or subtracted. Most other regex engines only store the most recent match of each capturing groups. When `(\w)+` matches `abc` then `Match.Groups[1].Value` returns `c` as with other regex engines, but `Match.Groups[1].Captures` stores all three iterations of the group: `a`, `b`, and `c`.

Looking Inside The Regex Engine

Let's apply the regex `(?'open'o)+(?'between-open'c)+` to the string `ooccc`. `(?'open'o)` matches the first `o` and stores that as the first capture of the group "open". The quantifier `+` repeats the group. `(?'open'o)` matches the second `o` and stores that as the second capture. Repeating again, `(?'open'o)` fails to match the first `c`. But the `+` is satisfied with two repetitions.

The regex engine advances to `(?'between-open'c)`. Before the engine can enter this balancing group, it must check whether the subtracted group "open" has captured something. It has captured the second `o`. The engine enters the group, subtracting the most recent capture from "open". This leaves the group "open" with the first `o` as its only capture. Now inside the balancing group, `c` matches `c`. The engine exits the balancing group. The group "between" captures the text between the match subtracted from "open" (the second `o`) and the `c` just matched by the balancing group. This is an empty string but it is captured anyway.

The balancing group too has `+` as its quantifier. The engine again finds that the subtracted group "open" captured something, namely the first `o`. The regex enters the balancing group, leaving the group "open" without any matches. `c` matches the second `c` in the string. The group "between" captures `oc` which is the text between the match subtracted from "open" (the first `o`) and the second `c` just matched by the balancing group.

The balancing group is repeated again. But this time, the regex engine finds that the group "open" has no matches left. The balancing group fails to match. The group "between" is unaffected, retaining its most recent capture.

The `+` is satisfied with two iterations. The engine has reached the end of the regex. It returns `oocc` as the overall match. `Match.Groups['open'].Success` will return `false`, because all the captures of that group were subtracted. `Match.Groups['between'].Value` returns "`oc`".

Matching Balanced Pairs

We need to modify this regex if we want it to match a balanced number of o's and c's. To make sure that the regex won't match `ooccc`, which has more c's than o's, we can add anchors: `^(?'open'o)+(?'-open'c)+$`. This regex goes through the same matching process as the previous one. But after `(?'-open'c)+` fails to match its third iteration, the engine reaches `$` instead of the end of the regex. This fails to match. The regex engine will backtrack trying different permutations of the quantifiers, but they will all fail to match. No match can be found.

But the regex `^(?'open'o)+(?'-open'c)+$` still matches `ooc`. The matching process is again the same until the balancing group has matched the first `c` and left the group 'open' with the first `o` as its only capture. The quantifier makes the engine attempt the balancing group again. The engine again finds that the subtracted group "open" captured something. The regex enters the balancing group, leaving the group "open" without any matches. But now, `c` fails to match because the regex engine has reached the end of the string.

The regex engine must now backtrack out of the balancing group. When backtracking a balancing group, .NET also backtracks the subtraction. Since the capture of the first `o` was subtracted from "open" when entering the balancing group, this capture is now restored while backtracking out of the balancing group. The repeated group `(?'-open'c)+` is now reduced to a single iteration. But the quantifier is fine with that, as `+` means "once or more" as it always does. Still at the end of the string, the regex engine reaches `$` in the regex, which matches. The whole string `ooc` is returned as the overall match. `Match.Groups['open'].Captures` will hold the first `o` in the string as the only item in the CaptureCollection. That's because, after backtracking, the second `o` was subtracted from the group, but the first `o` was not.

To make sure the regex matches `oc` and `oocc` but not `ooc`, we need to check that the group "open" has no captures left when the matching process reaches the end of the regex. We can do this with a conditional. `(?(open)(?!))` is a conditional that checks whether the group "open" matched something. In .NET, having matched something means still having captures on the stack that weren't backtracked or subtracted. If the group has captured something, the "if" part of the conditional is evaluated. In this case that is the empty negative lookahead `(?!)`. The empty string inside this lookahead always matches. Because the lookahead is negative, this causes the lookahead to always fail. Thus the conditional always fails if the group has captured something. If the group has not captured anything, the "else" part of the conditional is evaluated. In this case there is no "else" part. This means that the conditional always succeeds if the group has not captured something. This makes `(?(open)(?!))` a proper test to verify that the group "open" has no captures left.

The regex `^(?'open'o)+(?'-open'c)+(?(open)(?!))$` fails to match `ooc`. When `c` fails to match because the regex engine has reached the end of the string, the engine backtracks out of the balancing group, leaving "open" with a single capture. The regex engine now reaches the conditional, which fails to match. The regex engine will backtrack trying different permutations of the quantifiers, but they will all fail to match. No match can be found.

The regex `^(?'open'o)+(?'-open'c)+(?(open)(?!))$` does match `oocc`. After `(?'-open'c)+` has matched `cc`, the regex engine cannot enter the balancing group a third time, because "open" has no captures left. The engine advances to the conditional. The conditional succeeds because "open" has no captures left and the conditional does not have an "else" part. Now `$` matches at the end of the string.

Matching Balanced Constructs

`^(?:(?'open'o)+(?'-open'c)+)+(?(open)(?!))$` wraps the capturing group and the balancing group in a non-capturing group that is also repeated. This regex matches any string like `ooocooccocccoc` that contains any number of perfectly balanced o's and c's, with any number of pairs in sequence, nested to any depth. The balancing group makes sure that the regex never matches a string that has more c's at any point in the string than it has o's to the left of that point. The conditional at the end, which must remain outside the repeated group, makes sure that the regex never matches a string that has more o's than c's.

`^(?>(?'open'o)+(?'-open'c)+)+(?(open)(?!))$` optimizes the previous regex by using an atomic group instead of the non-capturing group. The atomic group, which is also non-capturing, eliminates nearly all backtracking when the regular expression cannot find a match, which can greatly increase performance when used on long strings with lots of o's and c's but that aren't properly balanced at the end. The atomic group does not change how the regex matches strings that do have balanced o's and c's.

`^m*(?>(?>(?'open'o)m*)+(?>(?'-open'c)m*)+)+(?(open)(?!))$` allows any number of letters `m` anywhere in the string, while still requiring all o's and c's to be balanced. `m*` at the start of the regex allows any number of m's before the first o. `(?'open'o)+` was changed into `(?>(?'open'o)m*)+` to allow any number of m's after each o. Similarly, `(?'-open'c)+` was changed into `(?>(?'-open'c)m*)+` to allow any number of m's after each c.

This is the generic solution for matching balanced constructs using .NET's balancing groups or capturing group subtraction feature. You can replace `o`, `m`, and `c` with any regular expression, as long as no two of these three can match the same text.

`^[^()]*(?>(?>(?'open'\()[^()]*)+(?>(?'-open'\))[^()]*)+)+(?(open)(?!))$` applies this technique to match a string in which all parentheses are perfectly balanced.

Backreferences To Subtracted Groups

You can use backreferences to groups that have their matches subtracted by a balancing group. The backreference matches the group's most recent match that wasn't backtracked or subtracted. The regex `(?'x'[ab]){2}(?'-x')\k'x'` matches `aaa`, `aba`, `bab`, or `bbb`. It does not match `aab`, `abb`, `baa`, or `bba`. The first and third letters of the string have to be the same.

Let's see how `(?'x'[ab]){2}(?'-x')\k'x'` matches `aba`. The first iteration of `(?'x'[ab])` captures `a`. The second iteration captures `b`. Now the regex engine reaches the balancing group `(?'-x')`. It checks whether the group "x" has matched, which it has. The engine enters the balancing group, subtracting the match `b` from the stack of group "x". There are no regex tokens inside the balancing group. It matches without advancing through the string. Now the regex engine reaches the backreference `\k'x'`. The match at the top of the stack of group "x" is `a`. The next character in the string is also an `a` which the backreference matches. `aba` is found as an overall match.

When you apply this regex to `abb`, the matching process is the same, except that the backreference fails to match the second `b` in the string. Since the regex has no other permutations that the regex engine can try, the match attempt fails.

Matching Palindromes

`^(?'letter'[a-z])+[a-z]?(?:\k'letter'(?'-letter'))+(?(letter)(?!))$` matches palindrome words of any length. This regular expression takes advantage of the fact that backreferences and capturing group subtraction work well together. It also uses an empty balancing group as the regex in the previous section.

Let's see how this regex matches the palindrome `radar`. `^` matches at the start of the string. Then `(?'letter'[a-z])+` iterates five times. The group "letter" ends up with five matches on its stack: `r`, `a`, `d`, `a`, and `r`. The regex engine is now at the end of the string and at `[a-z]?` in the regex. It doesn't match, but that's fine, because the quantifier makes it optional. The engine now reaches the backreference `\k'letter'`. The group "letter" has `r` at the top of its stack. This fails to match the void after the end of the string.

The regex engine backtracks. `(?'letter'[a-z])+` is reduced to four iterations, leaving `r`, `a`, `d`, and `a` on the stack of the group "letter". `[a-z]?` matches `r`. The backreference again fails to match the void after the end of the string. The engine backtracks, forcing `[a-z]?` to give up its match. Now "letter" has `a` at the top of its stack. This causes the backreference to fail to match `r`.

More backtracking follows. `(?'letter'[a-z])+` is reduced to three iterations, leaving `d` at the top of the stack of the group "letter". The engine again proceeds with `[a-z]?`. It fails again because there is no `d` for the backreference to match.

Backtracking once more, the capturing stack of group "letter" is reduced to `r` and `a`. Now the tide turns. `[a-z]?` matches `d`. The backreference matches `a` which is the most recent match of the group "letter" that wasn't backtracked. The engine now reaches the empty balancing group `(?'-letter')`. This matches, because the group "letter" has a match `a` to subtract.

The backreference and balancing group are inside a repeated non-capturing group, so the engine tries them again. The backreference matches `r` and the balancing group subtracts it from "letter"'s stack, leaving the capturing group without any matches. Iterating once more, the backreference fails, because the group "letter" has no matches left on its stack. This makes the group act as a non-participating group. Backreferences to non-participating groups always fail in .NET, as they do in most regex flavors.

`(?:\k'letter'(?'-letter'))+` has successfully matched two iterations. Now, the conditional `(?(letter)(?!))` succeeds because the group "letter" has no matches left. The anchor `$` also matches. The palindrome `radar` has been matched.

Make a Donation

Page URL: https://www.regular-expressions.info/balancing.html
Page last updated: 22 November 2019
Site last updated: 08 April 2020