Using Backreferences To Match The Same Text Again

Backreferences match the same text as previously matched by a capturing group. Suppose you want to match a pair of opening and closing HTML tags, and the text in between. By putting the opening tag into a backreference, we can reuse the name of the tag for the closing tag. Here's how: `<([A-Z][A-Z0-9]*)\b[^>]*>.*?</\1>`. This regex contains only one pair of parentheses, which capture the string matched by `[A-Z][A-Z0-9]*`. This is the opening HTML tag. (Since HTML tags are case insensitive, this regex requires case insensitive matching.) The backreference `\1` (backslash one) references the first capturing group. `\1` matches the exact same text that was matched by the first capturing group. The `/` before it is a literal character. It is simply the forward slash in the closing HTML tag that we are trying to match.

To figure out the number of a particular backreference, scan the regular expression from left to right. Count the opening parentheses of all the numbered capturing groups. The first parenthesis starts backreference number one, the second number two, etc. Skip parentheses that are part of other syntax such as non-capturing groups. This means that non-capturing parentheses have another benefit: you can insert them into a regular expression without changing the numbers assigned to the backreferences. This can be very useful when modifying a complex regular expression.

You can reuse the same backreference more than once. `([a-c])x\1x\1` matches `axaxa`, `bxbxb` and `cxcxc`.

Most regex flavors support up to 99 capturing groups and double-digit backreferences. So `\99` is a valid backreference if your regex has 99 capturing groups.

Looking Inside The Regex Engine

Let's see how the regex engine applies the regex `<([A-Z][A-Z0-9]*)\b[^>]*>.*?</\1>` to the string `Testing <B><I>bold italic</I></B> text`. The first token in the regex is the literal `<`. The regex engine traverses the string until it can match at the first `<` in the string. The next token is `[A-Z]`. The regex engine also takes note that it is now inside the first pair of capturing parentheses. `[A-Z]` matches `B`. The engine advances to `[A-Z0-9]` and `>`. This match fails. However, because of the star, that's perfectly fine. The position in the string remains at `>`. The word boundary `\b` matches at the `>` because it is preceded by `B`. The word boundary does not make the engine advance through the string. The position in the regex is advanced to `[^>]`.

This step crosses the closing bracket of the first pair of capturing parentheses. This prompts the regex engine to store what was matched inside them into the first backreference. In this case, `B` is stored.

After storing the backreference, the engine proceeds with the match attempt. `[^>]` does not match `>`. Again, because of another star, this is not a problem. The position in the string remains at `>`, and position in the regex is advanced to `>`. These obviously match. The next token is a dot, repeated by a lazy star. Because of the laziness, the regex engine initially skips this token, taking note that it should backtrack in case the remainder of the regex fails.

The engine has now arrived at the second `<` in the regex, and the second `<` in the string. These match. The next token is `/`. This does not match `I`, and the engine is forced to backtrack to the dot. The dot matches the second `<` in the string. The star is still lazy, so the engine again takes note of the available backtracking position and advances to `<` and `I`. These do not match, so the engine again backtracks.

The backtracking continues until the dot has consumed `<I>bold italic`. At this point, `<` matches the third `<` in the string, and the next token is `/` which matches `/`. The next token is `\1`. Note that the token is the backreference, and not `B`. The engine does not substitute the backreference in the regular expression. Every time the engine arrives at the backreference, it reads the value that was stored. This means that if the engine had backtracked

beyond the first pair of capturing parentheses before arriving the second time at `\1`, the new value stored in the first backreference would be used. But this did not happen here, so `B` it is. This fails to match at `I`, so the engine backtracks again, and the dot consumes the third `<` in the string.

Backtracking continues again until the dot has consumed `<I>bold italic</I>`. At this point, `<` matches `<` and `/` matches `/`. The engine arrives again at `\1`. The backreference still holds `B`. `\1` matches `B`. The last token in the regex, `>` matches `>`. A complete match has been found: `<B><I>bold italic</I></B>`.

Backtracking Into Capturing Groups

You may have wondered about the word boundary `\b` in the `<([A-Z][A-Z0-9]*)\b[^>]*>.*?</\1>` mentioned above. This is to make sure the regex won't match incorrectly paired tags such as `<boo>bold</b>`. You may think that cannot happen because the capturing group matches `boo` which causes `\1` to try to match the same, and fail. That is indeed what happens. But then the regex engine backtracks.

Let's take the regex `<([A-Z][A-Z0-9]*)[^>]*>.*?</\1>` without the word boundary and look inside the regex engine at the point where `\1` fails the first time. First, `.*?` continues to expand until it has reached the end of the string, and `</\1>` has failed to match each time `.*?` matched one more character.

Then the regex engine backtracks into the capturing group. `[A-Z0-9]*` has matched `oo`, but would just as happily match `o` or nothing at all. When backtracking, `[A-Z0-9]*` is forced to give up one character. The regex engine continues, exiting the capturing group a second time. Since `[A-Z][A-Z0-9]*` has now matched `bo`, that is what is stored into the capturing group, overwriting `boo` that was stored before. `[^>]*` matches the second `o` in the opening tag. `>.*?</` matches `>bold</`. `\1` fails again.

The regex engine does all the same backtracking once more, until `[A-Z0-9]*` is forced to give up another character, causing it to match nothing, which the star allows. The capturing group now stores just `b`. `[^>]*` now matches `oo`. `>.*?</` once again matches `>bold<`. `\1` now succeeds, as does `>` and an overall match is found. But not the one we wanted.

There are several solutions to this. One is to use the word boundary. When `[A-Z0-9]*` backtracks the first time, reducing the capturing group to `bo`, `\b` fails to match between `o` and `o`. This forces `[A-Z0-9]*` to backtrack again immediately. The capturing group is reduced to `b` and the word boundary fails between `b` and `o`. There are no further backtracking positions, so the whole match attempt fails.

The reason we need the word boundary is that we're using `[^>]*` to skip over any attributes in the tag. If your paired tags never have any attributes, you can leave that out, and use `<([A-Z][A-Z0-9]*)>.*?</\1>`. Each time `[A-Z0-9]*` backtracks, the `>` that follows it fails to match, quickly ending the match attempt.

If you don't want the regex engine to backtrack into capturing groups, you can use an atomic group. The tutorial section on atomic grouping has all the details.

Repetition and Backreferences

As I mentioned in the above inside look, the regex engine does not permanently substitute backreferences in the regular expression. It will use the last match saved into the backreference each time it needs to be used. If a new match is found by capturing parentheses, the previously saved match is overwritten. There is a clear difference between `([abc]+)` and `([abc])+`. Though both successfully match `cab`, the first regex will put `cab` into the first backreference, while the second regex will only store `b`. That is because in the second regex, the plus caused the pair of parentheses to repeat three times. The first time, `c` was stored. The second time, `a`, and the third time `b`. Each time, the previous value was overwritten, so `b` remains.

This also means that `([abc]+)=\1` will match `cab=cab`, and that `([abc])+=\1` will not. The reason is that when the engine arrives at `\1`, it holds `b` which fails to match `c`. Obvious when you look at a simple example like this one, but a common cause of difficulty with regular expressions nonetheless. When using backreferences, always double check that you are really capturing what you want.

Useful Example: Checking for Doubled Words

When editing text, doubled words such as “the the” easily creep in. Using the regex `\b(\w+)\s+\1\b` in your text editor, you can easily find them. To delete the second word, simply type in `\1` as the replacement text and click the Replace button.

Parentheses and Backreferences Cannot Be Used Inside Character Classes

Parentheses cannot be used inside character classes, at least not as metacharacters. When you put a parenthesis in a character class, it is treated as a literal character. So the regex `[(a)b]` matches `a`, `b`, `(`, and `)`.

Backreferences, too, cannot be used inside a character class. The \1 in a regex like `(a)[\1b]` is either an error or a needlessly escaped literal 1. In JavaScript it’s an octal escape.

Make a Donation

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