How do I write a regular expression that matches an IPv4 dotted address?

devblogs.microsoft.com/oldnewthing/20060522-08

May 22, 2006

Raymond Chen

Writing a regular expression that matches an IPv4 dotted address is either easy or hard, depending on how good a job you want to do. In fact, to make things easier, let's match only the decimal dotted notation, leaving out the hexadecimal variant, as well as the non-dotted variants.

For the purpose of this discussion, I'll restrict myself to the common subset of the regular expression languages shared by perl, JScript, and the .NET Framework, and I'll assume ECMA mode, wherein `\d` matches only the characters 0 through 9. (By default, in the .NET Framework, \d matches any decimal digit, not just 0 through 9.)

The easiest version is just to take any string of four decimal numbers separated by periods.

```
/^\d+\.\d+\.\d+\.\d+$/
```

This is nice as far as it goes, but it erroneously accepts strings like "448.90210.0.65535". A proper decimal dotted address has no value larger than 255. But writing a regular expression that matches the integers 0 through 255 is hard work because regular expressions don't understand arithmetic; they operate purely textually. Therefore, you have to describe the integers 0 through 255 in purely textual means.

- Any single digit is valid (representing 0 through 9).
- Any nonzero digit followed by another digit is valid (representing 10 through 99).
- A "1" followed by two digits is valid (100 through 199).
- A "2" followed by "0" through "4" followed by another digit is valid (200 through 249).
- A "25" followed by "0" through "5" is valid (250 throuth 255).

Given this textual breakdown of the integers 0 through 255, your first try would be something like this:

```
/^\d|[1-9]\d|1\d\d|2[0-4]\d|25[0-5]$/
```

This can be shrunk a bit by recognizing that the first two rules above could be combined into

Any digit, optionally preceded by a nonzero digit, is valid.

yielding

```
/^[1-9]?\d|1\d\d|2[0-4]\d|25[0-5]$/
```

Now we just have to do this four times with periods in between:

```
/^([1-9]?\d|1\d\d|2[0-4]\d|25[0-5])\.([1-9]?\d|1\d\d|2[0-4]\d|25[0-5])\.([1-9]?
\d|1\d\d|2[0-4]\d|25[0-5])\.([1-9]?\d|1\d\d|2[0-4]\d|25[0-5])$/
```

Congratulations, we have just taken a simple description of the dotted decimal notation in words and converted into a monstrous regular expression that is basically unreadable. Imagine you were maintaining a program and stumbled across this regular expression. How long would it take you to figure out what it did?

Oh, and it might not be right yet, because some parsers accept leading zeroes in front of each decimal value without affecting it. (For example, 127.0.0.001 is the same as 127.0.0.1. On the other hand, some parsers treat a leading zero as an octal prefix.) Updating our regular expression to accept leading decimal zeroes means that we now have

```
/^0*([1-9]?\d|1\d\d|2[0-4]\d|25[0-5])\.0*([1-9]?\d|1\d\d|2[0-4]\d|25[0-5])\.0*([1-9]?
\d|1\d\d|2[0-4]\d|25[0-5])\.0*([1-9]?\d|1\d\d|2[0-4]\d|25[0-5])$/
```

This is why I both love and hate regular expressions. They are a great way to express simple patterns. And they are a horrific way to express complicated ones. Regular expressions are probably the world's most popular write-only language.

Aha, but you see, all this time diving into regular expressions was a mistake. Because we failed to figure out what the actual problem was. This was a case of somebody "solving" half of their problem and then asking for help with the other half: "I have a string and I want to check whether it is a dotted decimal IPv4 address. I know, I'll write a regular expression! Hey, can anybody help me write this regular expression?"

The real problem was not "How do I write a regular expression to recognize a dotted decimal IPv4 address." The real problem was simply "How do I recognize a dotted decimal IPv4 address." And with this broader goal in mind, you recognize that limiting yourself to a regular expression only made the problem harder.

```
function isDottedIPv4(s)
{
 var match = s.match(/^(\d+)\.(\d+)\.(\d+)\.(\d+)$/);
 return match != null &&
        match[1] <= 255 && match[2] <= 255 &&
        match[3] <= 255 && match[4] <= 255;
}
WScript.StdOut.WriteLine(isDottedIPv4("127.0.0.001"));
WScript.StdOut.WriteLine(isDottedIPv4("448.90210.0.65535"));
WScript.StdOut.WriteLine(isDottedIPv4("microsoft.com"));
```

And this was just a simple dotted decimal IPv4 address. Woe unto you if you decide you want to parse e-mail addresses.

Don't make regular expressions do what they're not good at. If you want to match a simple pattern, then match a simple pattern. If you want to do math, then do math. As commenter Maurits put it, "The trick is not to spend time developing a combination hammer/screwdriver, but just use a hammer and a screwdriver.

Raymond Chen

Follow