Upcoming Regular Expression Features

By Jakob Gruber

Jakob is an engineer working on V8

By Yang Guo

Yang is an engineer working on V8

ES2015 introduced many new features to the JavaScript language, including significant improvements to the regular expression syntax with the Unicode (`/u`) and sticky (`/y`) flags. But development has not stopped since then. In tight collaboration with other members at TC39 (the ECMAScript standards body), the V8 team has proposed and co-designed several new features to make regular expressions even more powerful.

These features are currently being proposed for inclusion in the JavaScript specification. Even though the proposals have not been fully accepted, they are already at Stage 3 in the TC39 process. We have implemented these features behind a flag (see below) in order to be able to provide timely design and implementation feedback to the respective proposal authors before the specification is finalized.

This blog post gives you a preview of this exciting future. If you'd like to follow along with the upcoming examples, enable experimental JavaScript features at **`chrome://flags/#enable-javascript-harmony`**.

Named Captures

Regular expressions can contain so-called captures (or groups), which can capture a portion of the matched text. So far, developers could only refer to these captures by their index, which is determined by the position of the capture within the pattern.

```
const pattern = /(\d{4})-(\d{2})-(\d{2})/u;
const result = pattern.exec('2017-07-10');
// result[0] === '2017-07-10'
// result[1] === '2017'
// result[2] === '07'
// result[3] === '10'
```

But regular expressions are already notoriously difficult to read, write, and maintain, and numeric references can add further complications. For instance, in longer patterns it can be tricky to determine the index of a particular capture:

```
/(?:(.)(.(?<=[^(])(.)))/  // Index of the last capture?
```

And even worse, changes to a pattern can potentially shift the indices of all existing captures:

```
/(a)(b)(c)\3\2\1/     // A few simple numbered backreferences.
/(.)(a)(b)(c)\4\3\2/  // All need to be updated.
```

Named captures are an upcoming feature that helps mitigate these issues by allowing developers to assign names to captures. The syntax is similar to Perl, Java, .Net, and Ruby:

```
const pattern = /(?<year>\d{4})-(?<month>\d{2})-(?<day>\d{2})/u;
const result = pattern.exec('2017-07-10');
// result.groups.year === '2017'
// result.groups.month === '07'
// result.groups.day === '10'
```

Named captures can also be referenced by named backreferences and through `String.prototype.replace`:

```
// Named backreferences.
/(?<LowerCaseX>x)y\k<LowerCaseX>/.test('xyx');  // true

// String replacement.
const pattern = /(?<fst>a)(?<snd>b)/;
'ab'.replace(pattern, '$<snd>$<fst>');                                  // 'ba'
'ab'.replace(pattern, (m, p1, p2, o, s, {fst, snd}) => fst + snd);  // 'ba'
```

Full details of this new feature are available in the specification proposal.

dotAll Flag

By default, the `.` atom in regular expressions matches any character except for line terminators:

```
/foo.bar/u.test('foo\nbar');   // false
```

A proposal introduces dotAll mode, enabled through the `/s` flag. In dotAll mode, `.` matches line terminators as well.

```
/foo.bar/su.test('foo\nbar');  // true
```

Full details of this new feature are available in the specification proposal.

Unicode Property Escapes

Regular expression syntax has always included shorthands for certain character classes. `\d` represent digits and is really just `[0-9]`; `\w` is short for word characters, or `[A-Za-z0-9_]`.

With Unicode awareness introduced in ES2015, there are suddenly many more characters that could be considered numbers, for example the circled digit one: ①; or considered word characters, for example the Chinese character for snow: 雪.

Neither of these can be matched with `\d` or `\w`. Changing the meaning of these shorthands would break existing regular expression patterns.

Instead, new character classes are being introduced. Note that they are only available for Unicode-aware regular expressions denoted by the `/u` flag.

```
/\p{Number}/u.test('①');      // true
/\p{Alphabetic}/u.test('雪');  // true
```

The inverse can be matched by with `\P`.

```
/\P{Number}/u.test('①');      // false
/\P{Alphabetic}/u.test('雪');  // false
```

The Unicode consortium defines many more ways to classify code points, for example math symbols or Japanese Hiragana characters:

```
/^\p{Math}+$/u.test('∛∞∉');                              // true
/^\p{Script_Extensions=Hiragana}+$/u.test('ひらがな');  // true
```

The full list of supported Unicode property classes can be found in the current specification proposal. For more examples, take a look at this informative article.

Lookbehind Assertions

Lookahead assertions have been part of JavaScript's regular expression syntax from the start. Their counterpart, lookbehind assertions, are finally being introduced. Some of you may remember that this has been part of V8 for quite some time already. We even use lookbehind asserts under the hood to implement the Unicode flag specified in ES2015.

The name already describes its meaning pretty well. It offers a way to restrict a pattern to only match if preceded by the pattern in the lookbehind group. It comes in both matching and non-matching flavors:

```
/(?<=\$)\d+/.exec('$1 is worth about ¥123');  // ['1']
/(?<!\$)\d+/.exec('$1 is worth about ¥123');  // ['123']
```

For more details, check out our previous blog post dedicated to lookbehind assertions, and examples in related V8 test cases.

Acknowledgements

This blog post wouldn't be complete without mentioning some of the people that have worked hard to make this happen: especially language champions Mathias Bynens, Dan Ehrenberg, Claude Pache, Brian Terlson, Thomas Wood, Gorkem Yakin, and Irregexp guru Erik Corry; but also everyone else who has contributed to the language specification and V8's implementation of these features.

We hope you're as excited about these new regular expression features as we are!

Last updated July 2, 2018.