RegExp Match Indices

Implementation Feedback from JavaScriptCore

November 2020

https://github.com/tc39/proposal-regexp-match-indices

Currently Stage 3

Proposal Synopsis

Adds **`indices`** property to **`Matches`** value returned from **`RegExp.prototype.exec`** and related **`RegExp`** functions.

`Matches.indices` is an array of arrays, one array for the main match and each captured sub pattern or **`undefined`** for unmatched subpatterns.

Each sub array has 2 elements, [*start*, *end*], where *start* is the index in the original string where that (sub)pattern matched and end is 1 more than the string index at the the end of the match.

`/(\w+) (\w+)/.exec("one two").indices` => **`[[0, 7], [0, 3], [4, 7]]`**

Concerns Raised by V8

At December 2019 TC 39 plenary meeting, V8 raised two implementation issues with the proposal, increased memory usage and performance hit.

- Increased memory due to allocating more objects to build `Matches` obj.
- Performance hit due to:
  - More allocations for every successful match.
  - More objects to GC.
- These issues penalized all RegExp use.
- V8 decided to opt for lazy materialization of `Matches.indices`.
  - Intercept access to `Matches.indices` and re-execute the match.
  - Only penalizes `Matches.indices` usage.

JSC's Implementation Experience

- Initial implementation, always materialize `Matches.indices`.
  - 17% perf slowdown for Octane2 regexp.
  - ~1% on JetStream2 overall.
- Second implementation, save indices from matching, lazily materialize `Matches.indices` when first accessed.
  - 3% perf slowdown for Octane2 regexp.
  - ~1% on JetStream2 overall..
- Third implementation, when `Matches.indices` first accessed re-execute match and materialize. (Very similar to V8's approach)
  - <.5% perf slowdown for Octane2 regexp.
  - Doesn't appear to impact JetStream2 perf.

Success?

So both V8 and JSC have (the same) path forward to implement RegExp Match Indices. But…

- While both implementations nearly eliminated performance and memory issues in their respective engines, the feature is not implemented optimally!
  - Both implementation match twice to populate `Matches.indices`.
  - Complicates performance sensitive code.
  - Will developers avoid the feature because of this.

History of Performance Concerns

- Google wasn't the first to raise memory and perf concerns.
  The first issue for this proposal in GitHub was
  Performance overhead of always adding the offsets object.
  - Redoing the match was considered unacceptable.
- Various alternatives to mitigate performance concerns have been proposed
  - New RegExp flag
  - Optional callback function
  - New RegExp functions
  - Optional value or option bag

Can we Revisit Perf Mitigations?

Since developers knows if they want **`.indices`**, they can let engines know.

- Two places developers can indicate intent, the **`RegExp`** or its use.
- I propose intent is with the **`RegExp`**, specifically by adding a new flag.
  - Likely the best performing option. **`BuiltInRegExp`** takes the **`RegExp`** as its first argument and engines probably use this abstraction.
  - Proposal champions at one point proposed adding **`o`**(offsets?) flag.
  - Another idea is **`n`** (numeric results). Note: Perl and .NET have **`n`** options.
  - We tried this in JSC, our fourth implementation.
    No perf issues and almost as clean as our first direct implementation.
  - JSC uses a “want **`.indices`**” flag internally for subsequent matches.

Summary

- JSC tried 4 implementations and had similar perf issues as V8.
- Current stage 3 conformant implementation (re-execute the match) has reasonable performance, but punishes those who use this feature.
- Suggest that we reconsider developer intent, specifically a new RegExp flag.

Discussion

Thank you!