RegExp builtins (re)design document

Attention: Shared Google-externally

Authors: bmeurer@chromium.org, jgruber@chromium.org

Last updated: 2017-04-21

Tracking bug: https://bugs.chromium.org/p/v8/issues/detail?id=5339

This document describes the design and implementation of the RegExp builtins. It deals with the `RegExp` constructor plus all its properties and the properties of its `prototype` property. It doesn't deal with the details of the actual implementation of the regular expression engine.

Background

There are two different regular expression engines available in V8, a simple interpreter based regular expression engine, and a sophisticated compiler that generates efficient machine code for the backtracking based NFA. Both engines are designed to be invoked more or less directly from C++ code; the interpreter itself is written in C++, so that's natural, while the compiler generates code chunks with C linkage and has its own special integration with the GC.

Traditionally V8 was using a *transactional runtime model*, where most of the runtime interaction with the core C++ code was not handlified and thus had to abort the transaction whenever it ran out of memory, go back to the runtime entry, which would then perform a GC, and retry the whole operation from the beginning. Obviously this model doesn't really work well if you need to perform side effects, i.e. like calling back into arbitrary JavaScript code. But even fairly innocent builtins like the `RegExp.prototype.flags` getter can already cause arbitrary JavaScript code to be run, since it does property lookups on the receiver object. So initially only the core of the RegExp builtins could be written in C++, and each builtin required at least one driver function written in JavaScript that drives the transactional core functionality, and takes care of the side effecting parts[1].

Motivation

This approach however limits the flexibility of the builtin implementation choices, as you always need to somehow keep the driver and the core implementation in sync and even expose additional intrinsics and runtime functions to the driver to access various implementation details (i.e. special in-object properties like the `flags` or the `source` internal fields of the `JSRegExp`

[1] This was actually not limited to `RegExp`. Almost all builtins had to be written this way initially in V8 due to the transactional runtime (and other design decisions), even if the core functionality was already implemented in C++ anyways.

class), which is then prone to errors and scatters the implementation unnecessarily. Specifically over the last 1 ½ years we ran into multiple issues with the `regexp.js` based implementation, that we now intend to address incrementally, namely:

1. Adding support for subclassing `RegExp` as required by the ES2015 specification tanked performance of the (important) RegExp benchmark in Octane by around 15% initially, and with some tricks we were able to reduce that to roughly 7%, which is still a lot that users have to pay for a feature that is probably not very useful in practice.
2. New features that are planned for an upcoming revision of the EcmaScript specification like *named captures* will likely cause further slow downs for a bunch of the `RegExp` builtins, because additional runtime calls are necessary to transfer data between the core and the driver.
3. Since the code in `regexp.js` is treated like arbitrary JavaScript, it is also subject to type feedback pollution, which means that as the builtins see different shapes of `JSRegExp` objects or the builtins take different execution paths, the overall performance degrades. In addition to that, the launch of Ignition will naturally slow down the code in `regexp.js` that was more or less carefully tuned for fullcodegen (and Crankshaft). This clearly goes against our strategy of predictable, good baseline performance[2].
4. Due to the scattered implementation of the RegExp builtins, there are transient states in the system where we have `JSRegExp` objects in the heap that are in an inconsistent state, i.e. not yet fully initialized and verifiable. One prominent example of this is the constructor itself which calls `%_NewObject` to create a `JSRegExp` object, whose internal fields are set to some undefined value, and then calls into `RegExpInitialize`, which can cause arbitrary JavaScript execution (due to the `TO_STRING` calls), and only then finally calls into `%RegExpInitializeAndCompile`, which brings the `JSRegExp` into a consistent state. This is currently somewhat OKish for the RegExp constructor case, but still unnecessarily complex, given that one could just atomically create and initialize the `JSRegExp` object and thereby avoid a bunch of bugs right away.
5. Since the core implementation (i.e. the execution of the actual regular expression) is only callable via C/C++, we need a special version of the `CEntryStub`, just to be able to actually execute a regular expression. This `RegExpExecStub` is several pages of tricky hand-written native code, and has a track record of a few somewhat hairy bugs associated with it. Since the `RegExpExecStub` is significantly faster than the C++ implementation in RegExpImpl::Exec, and it is critical to overall performance of RegExps, we will need to continue to rely on it for the time being. However, over time we can move all except for the actual invocation of the generated Irregexp code to CodeStubAssembler and remove most of the platform-dependent `RegExpExecStub.`

[2] Again, this problem is not limited to the RegExp builtins.

Another reason to move away from the driver based implementation approach is increased flexibility for changing the underlying data structures (i.e. the internal layout of a `JSRegExp`), but we don't really expect that to be necessary, so that's not an important reason, just a nice goodie.

Goals

In order to address the issues mentioned above, the idea is to first come up with a reasonable baseline implementation that is merely a port of the existing functionality - mostly unchanged - to C++/TurboFan builtins, and reiterate from there to eventually address all the issues mentioned above.

The goals for the first iteration are (these must be addressed as soon as possible, i.e. should ideally be done by the end of Q3 or beginning of Q4 2016):

- Performance neutral (mostly) port of the existing functionality.
- Reduce dependencies on `%_NewObject` and the `JSBuiltinConstructStub`, which is responsible for the inconsistent state problem described above (afterwards only the current Promise implementation will still depend on this).
- Migrate all the functions in `string.js` that are related to the regular expression builtins to C++ as well and find appropriate abstractions in the runtime to improve maintainability of the relevant builtins and core parts.
- Remove the `RegExpConstructResultStub`, but continue to rely on `RegExpEntryStub` for the actual Irregexp entry point.

The middle- and long-term goals are (these should be addressed by the end of 2016, but are not on the critical path):

- Evaluate the need to/benefit of providing some builtins as TurboFan builtins, and eventually even inline some of that directly into TurboFan (via the `JSBuiltinReducer`). This has to be driven by performance data. For example, the source/flags accessors could benefit from this, or the String.prototype.replace method.
- Address the performance regression that was introduced with ES6 subclassing of regular expressions, and maybe even improve performance above the level we had initially. At least recovering the previous performance should be a matter of adding a "protector bit" on the `Isolate` similar to how we optimize @@species.
- Eventually remove the compatibility workarounds that we still have in our implementation (and that we will for now port to C++).
- Turn on support for named captures without any serious performance impact on the regular expression benchmarks in Octane, JetStream and SunSpider (i.e. prove that the new implementation is viable and gives us the expected flexibility).

Implementation

Initially, the plan was to migrate the JavaScript implementation of `RegExp` functions to native C++. This turned out to be too slow in practise, mostly since allocations, property access, substring construction, and entry into the generated Irregexp code are significantly slower[3] from C++ than what is possible from TurboFan/CodeStubAssembler (CSA). We therefore changed course and based the new `RegExp` implementation on CSA instead, allowing us not only to reach pre-ES6 levels of performance but to exceed them (often by a significant factor).

While some functions, e.g. `@@test`, are simple enough to implement completely in CSA, more complicated logic usually requires that the implementation is split between a CSA fast path for unmodified RegExp instances, and a slow path (usually implemented in C++ runtime) to handle other cases such as ES6 subclassing and monkeypatched `RegExp`. `@@replace` is an extreme example of this, as it currently dispatches to two different CSA fast paths, one runtime fast path, and falls back to the runtime slow path in other cases.

In fast paths, we can assume that the RegExp instance is completely unmodified, meaning we can perform fast reads and writes to internal properties such as `lastIndex`, and we can simply call our internal `RegExp.prototype.exec` implementation without first checking the `RegExp`'s `'exec'` property. Fast paths also guarantee non-observability of, e.g., `RegExp` property accesses and `exec` calls, allowing us to take implementation shortcuts for which the end state, but not necessarily intermediate states, are spec-compliant.

Fast path checks are based map checks: during bootstrapping, we store the RegExp function's initial map as well as the RegExp function's initial prototype's initial map on the context. During a fast path check, we compare these against the given regexp's map and prototype's map. If they match, the fast path is taken, otherwise we fall back to the slow path.

This fast path check differs from its previous form in `regexp.js`, which only performed an instance-type check and verified the '`exec`' property. This had both advantages (more cases were able to take the fast path) and disadvantages (it was not 100% correct, for instance when accessing lastIndex). See future work for our plan for more permissive fast path checks.

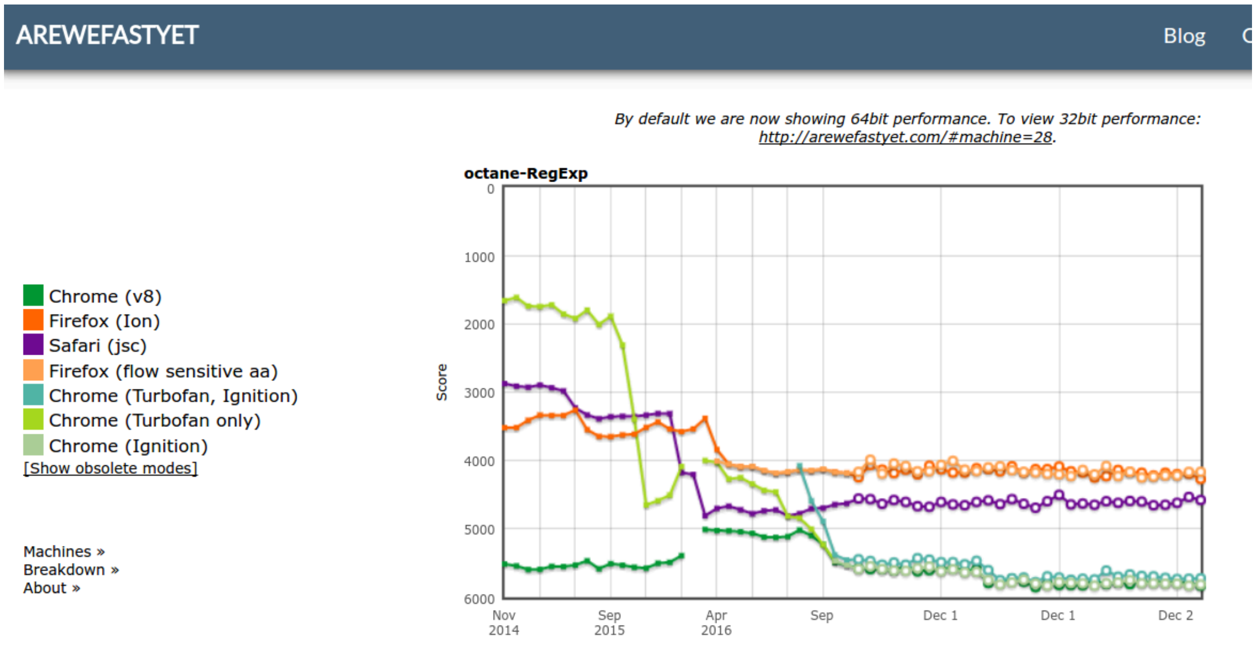
`RegExpLastMatchInfo` is our internal storage for the results of the last RegExp match, required e.g. by the '`input`' and '`lastMatch`' properties. It used to be implemented as a plain array-like `JSObject` with various fields mapped to specific indices. This is not necessary anymore now that all accesses are from C++/CSA. The last match info is now stored in a more efficient and convenient custom `FixedArray` type (`RegExpMatchInfo`).

[3] To demonstrate: forcing the original implementation to use the C++ runtime implementations of `%_SubString`, `%_RegExpConstructResultStub`, and `%_RegExpExec` instead of the natively implemented stub versions resulted in a performance loss of 54%.

Results

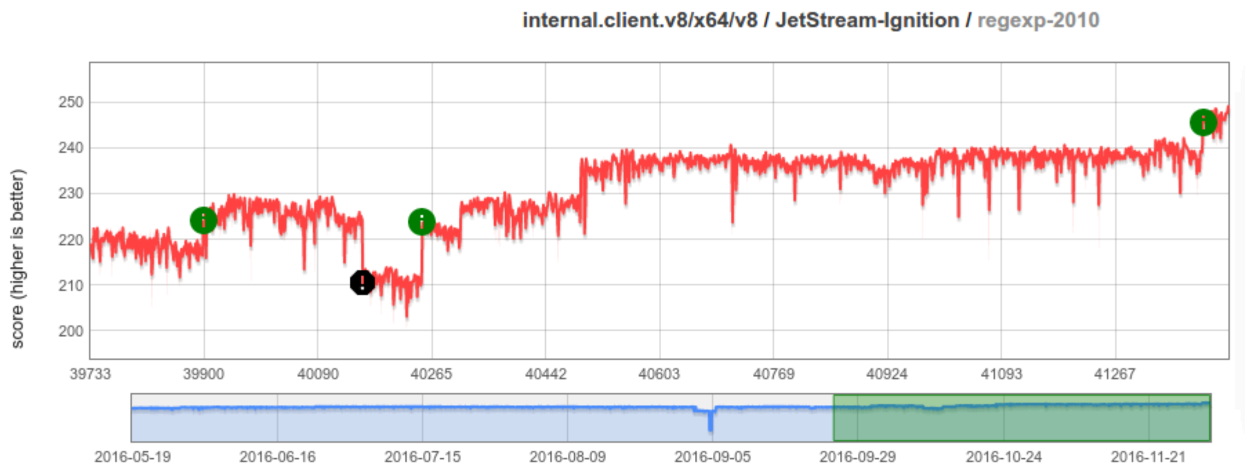
Performance results are better than expected, not only achieving our goals of approaching pre-ES6 performance, but actually exceeding it significantly.

According to arewefastyet.com, current octane/regexp scores are 16-40% better (compared to pre-migration, depending on configuration) and at least 5% better (compared to pre-ES6).



Other notable results according in comparison to the prior JS implementation in regexp.js:

- 9%-21% speedup on ss/unpack-code.
- 18% speedup on ss/validate-input.
- 13% speedup on jetstream/regexp-2010:



Our new RegExp microbenchmarks give a more detailed breakdown by functionality by measuring only individual functions (or functionalities such as flag accesses) and distinguishing between fast and slow paths. The following results compare against pre-migration on the fast path:

- `RegExp` constructor: 11% slower
- `exec`: 8% faster
- flag accesses: 66% faster
- `@@match`: 37% faster
- `replace`: 5% faster
- `@@search`: 11% faster
- `split`: 108% faster
- `test`: 25% faster

It would be quite simple to achieve similar gains for the RegExp constructor by moving the implementation to CSA, but it has not yet been deemed necessary since it is expected to be used only infrequently (RegExp literals take another code path).

`regexp.js` has been removed, and the implementation has moved to `builtins-regexp.cc`, `runtime-regexp.cc` (slow-paths), and `regexp/regexp-utils.cc` (functionality used by both runtime and builtins).

Due to performance reasons, most RegExp functions have been implemented in TurboFan, i.e. CodeStubAssembler (CSA). `RegExpConstructResultStub` has been moved to CSA. `RegExpExecStub` has not been modified and is called directly from the CSA implementation.

We no longer depend on `%_NewObject` and `JSBuiltinConstructStub`. Once the final remaining use in `promise.js` has been ported, these can be removed entirely.

string.js has remained largely untouched. StringSearch and StringMatch could easily be ported to TF with a fast path for unmodified JSRegExp arguments. StringReplace is a bit more difficult since it relies on GetSubstition, %StringIndexOf, and %StringReplaceOneCharWithString. While all of these functions could also be moved to C++, I'd expect a significant performance penalty due mainly to the required GetProperty and SubString calls.

The RegExpEntryStub has been completely replaced (CL). The replacement CSA implementation simply calls into generated Irregexp code (which has C-linkage) by using CallCFunction9. Removal of the stub (and its associated exit frame) has been made possible by removing the only throw site from the native call.

The current named captures proposal is fully supported (tracking bug) without performance regressions for the standard case. Creation of the groups object could still be improved in future work.

Finally, compat workarounds have been removed and several minor bugs were found and fixed in the process.

Future Work

There are several follow-up tasks that should be started in the near future:

- Fast-path checks should be made more permissive to allow e.g. monkey-patched RegExp prototypes to take the fast path as long as vital methods and properties have not been modified (see https://bugs.chromium.org/p/v8/issues/detail?id=5577).