Fast string concatenation in JavaScript

Attention: Shared Google-externally

Author: bmeurer@

Last Updated: 2019-03-21

TL;DR Fast String concatenation is hard in JavaScript. Engines have plenty of clever optimizations in place that make it non-trivial to find the best approach. This document outlines one approach that seems to result in somewhat good performance, and describes potential action items for V8 to further improve performance.

Link bit.ly/fast-string-concatenation-in-javascript

Bugs v8:6622, v8:8930, v8:8931, v8:8939

CLs 1498478, 1499497

Disclaimer *This is an investigation document, primarily targeted at engine implementers. Don't just go ahead and rewrite your code based on what you find here. Also keep in mind that oftentimes readability is far more important than squeezing out the last couple of milliseconds.*

Motivation

JavaScript doesn't have a native `StringBuilder` primitive (like in Java), instead `Strings` in JavaScript are generally constructed via `String` concatenation (either by means of the `+` operator or via the `String#concat()` method), by using (tagged) template literals, or by constructing an `Array` with the individual parts and then calling the `Array#join()` method on that.

Over the last decade, JavaScript engines have become very creative in providing fast string concatenation (also due to the lack of a more appropriate primitive). For example all engines adopted a technique called "ropes" to represent concatenated `Strings`, so `Strings` don't always hold the sequential list of characters, but can also instead point to other `Strings` that they are composed of.

These improvements yielded great speed-ups across various industry benchmarks. However it is also surprisingly difficult today to find a good code sequence to construct `Strings` in a performant way. Since engines don't always construct "ropes", as that would waste way too much memory for short `Strings`, it's sometimes surprising to see that performance differs depending on the exact code sequence used to concatenate the `Strings`.

```
const suffix = "1234";
let text = "ABCDEFGHIJKLMNOP";
text += "0" + suffix + "5";
```

```
const suffix = "1234";
let text = "ABCDEFGHIJKLMNOP";
text = text + "0" + suffix + "5";
```

Looking at the above snippets you'd probably think they are equivalent and interchangeable. Speaking only of the observable JavaScript result, they are totally interchangeable. However the performance of these is quite different and also the memory consumption. This is due to the way that JavaScript engines use "ropes".

It would take too long to explain all the details of how JavaScript engines represent strings internally, so I defer to Adventures in the land of substrings and RegExps by my colleague Vyacheslav Egorov, which has quite a bit of background on that. In short, the minimum number of characters to construct a "rope" (`ConsString` in V8 terminology) in the V8 JavaScript Engine is 13. Which means that in the example of

```
text += "0" + suffix + "5";
```

it will first construct a sequential string `"01234"` (from the `"0" + suffix`), then another sequential string for `"012345"` (from the `"0" + suffix + "5"`), and eventually construct a rope for the result `"ABCDEFGHIJKLMNOP012345"`. These intermediate sequential strings need to copy the characters, whereas the final rope just points to the individual substrings. On the other hand

```
text = text + "0" + suffix + "5";
```

will only create *ropes* for all substrings, which are faster to construct since they don't require copying characters. That might however come with a higher cost when you eventually need to *flatten the string*, which is V8 speak for turning a rope into a sequential string (for example when you need to pass it down the network or to some other API, or just start iterating its characters).

Case study: DOM to HTML serialization

One particular interesting use case that I've come across lately is the need to serialize an in-memory DOM tree to HTML that can be sent across the wire (for example in the context of server side rendering solutions like Angular Universal or Next.js). So I've created a test case that illustrates the surprising performance difference that can be observed with slightly changing the way the strings are constructed.

The test case is in bench-dom-serialize.js and it uses a hacked version of undom. The meaty part is in the different serialization functions:

```
function serializeNaiveInternal(el) {
  if (el.nodeType === 3) {
    return escape(el.textContent);
  } else {
```

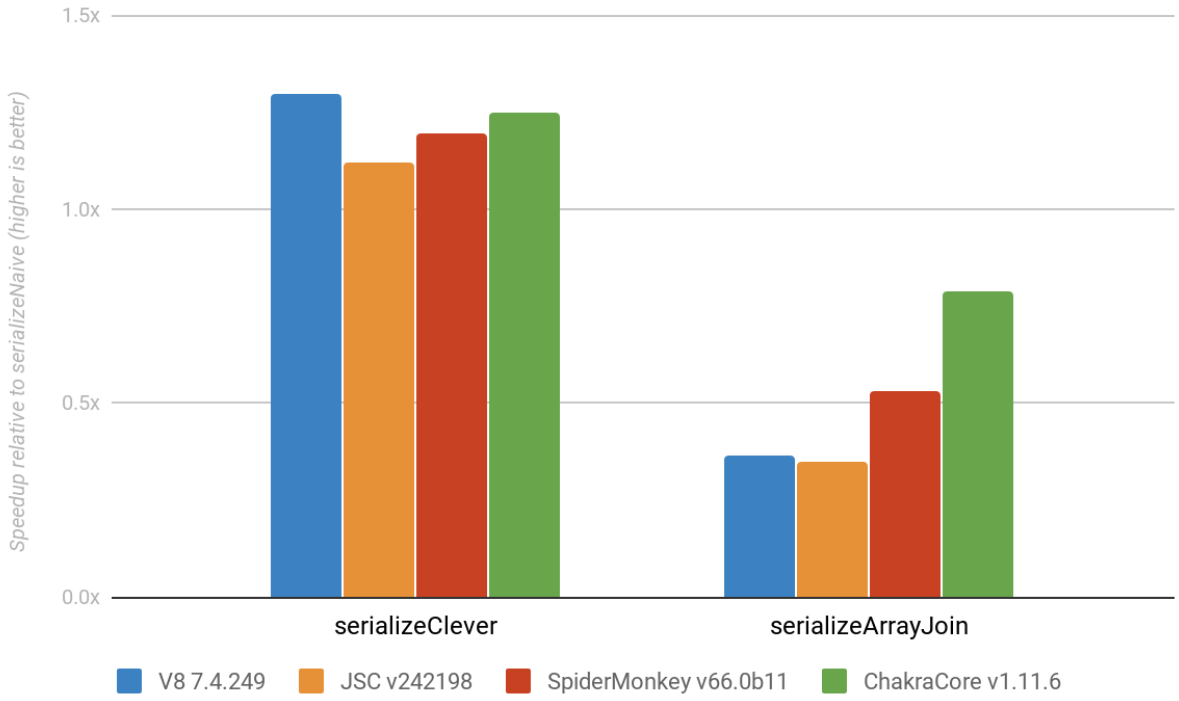
```
    const nodeName = el.nodeName;
    let s = '<' + nodeName;
    const attributes = el.attributes;
    if (attributes !== null) {
      for (const [key, value] of attributes) {
        s += ' ' + key + '="' + escapeAttr(value) + '"';
      }
    }
    s += '>';
    for (let c = el.firstChild; c !== null; c = c.nextSibling) {
      s += serializeNaiveInternal(c);
    }
    s += '</' + nodeName + '>';
    return s;
  }
}
```

The naive baseline implementation looks a lot like a casual developer would probably write this. The clever one on the other hand

```
// More clever string concatenation to focus on ConsStrings instead of
// intermediate SeqStrings
function serializeCleverInternal(el, s) {
  if (el.nodeType === 3) {
    return s + escape(el.textContent);
  } else {
    const nodeName = el.nodeName;
    s = s + '<' + nodeName;
    const attributes = el.attributes;
    if (attributes !== null) {
      for (const [key, value] of attributes) {
        s = s + ' ' + key + '="' + escapeAttr(value) + '"';
      }
    }
    s = s + '>';
    for (let c = el.firstChild; c !== null; c = c.nextSibling) {
      s = serializeCleverInternal(c, s);
    }
    s = s + '</' + nodeName + '>';
    return s;
  }
}
```

uses knowledge about the "ropes" and JavaScript engine internals. It's important to note that `s += '</' + nodeName + '>'` corresponds to `s = s + (('</' + nodeName) + '>')`, while `s = s + '</' + nodeName + '>'` is `s = ((s + '</') + nodeName) + '>'`. The observable result is the same, but the details of the string construction are different.

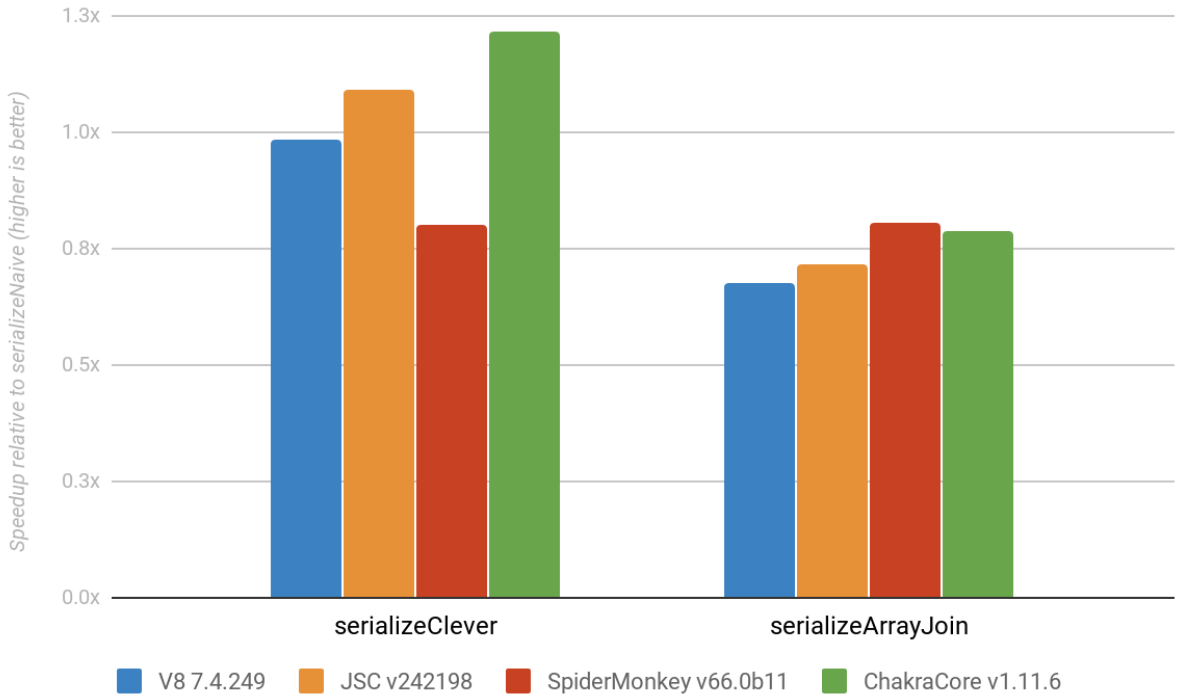
I've also included a version using `Array#join()` for reference. *You will not believe what happened next...*



As we can see from the graph, the performance of the *clever approach* is generally better across all the major JavaScript engines (which is probably due to the fact that they all use similar tricks to speed up string concatenation).

Flattening the rope in the end

However there's a catch with the benchmark above: We don't ever access the content of the final string, which means we keep it as a rope until it dies. It's interesting to repeat the experiment while also *forcibly flattening* the rope to a sequential string in the end. We can just enforce this by accessing the first character in each iteration of the test (via `String#charCodeAt()` method). This can be found in bench-dom-serialize-flatten.js.

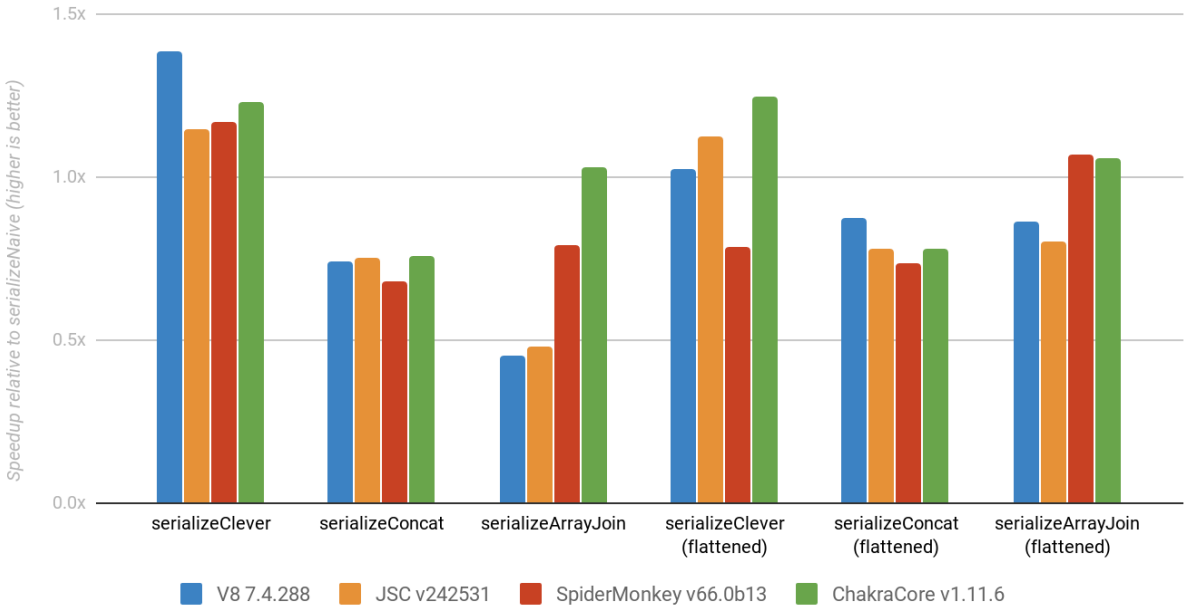


The results are slightly different in this case. However for V8 that doesn't mean the *clever approach* doesn't make sense, since in the specific case of HTML serialization we'll have to encode the string characters via UTF-8 and put it into an `ArrayBuffer` anyways in order to be able to hand it off to the network. And so we could as well imagine not doing the flattening separately, but rather as part of the UTF-8 encoding and copying to `ArrayBuffer`.

> **Note** This might require new APIs to accomplish. Maybe even on the C++ API depending on what the exact SSR setup looks like.

[Update 2019-03-06] Improved performance test

I created an improved version of the performance test in `bench-dom-serialize-v2.js`, which measures the overhead of *rope flattening* (via `String#charCodeAt()` at the end) and also includes a test case using String#concat().



So the `serializeClever` version is still significantly faster than the `serializeNaive` version, even with flattening (at least for V8).

Potential improvements

Now considering the findings, there might be some concrete steps we can take to improve performance even further (or also improve the performance of the *naive approach*).

[Done] Remove kOneByteDataHint

Bug v8:6622
CL 1498478

We still support a so-called *one byte data hint* for two byte strings, which was introduced in the early days of WebKit. This is used to indicate that a two-byte string only has one-byte payload. Not sure whether this is being used consistently, and whether this has any benefit at all. But couple of places, including the general path for string concatenation, check this flag (potentially unnecessary).

> **Note** This is done as of 1498478 and even gave a small **2-3% improvement** on the `serializeClever` test case in bench-dom-serialize.js above.

[Done] TurboFan generates *crippled* `ConsStrings`

Bug v8:8951
CL 1499497

When we inline `ConsString` creation into TurboFan, the generated code doesn't check at runtime that neither left nor right are empty strings. Which means TurboFan can generate these crippled `ConsStrings`. The rest of the system gracefully deals with this, but it's just wasting memory and potentially costing us performance.

Instead TurboFan should check properly that neither left nor right is the empty string, and only then construct a `ConsString`.

[Done] More fine-grained `String` feedback for the `Add` bytecode

Bug v8:8931
CL 1499497

When a site `left+right` has always created `ConsStrings` so far we should collect appropriate feedback for the Add bytecode in Ignition, and then in TurboFan instead of going through the generic `StringAdd_CheckNone` builtin we should check that the conditions for the `ConsString` creation are met (resulting string length is at least 13 characters), and just have the `ConsString` allocation inlined into the optimized code.

We could also imagine having even more fine-grained feedback about the kind of `String` that was created, i.e. when it was always a `SeqOneByteString`, we could easily inline the allocation of that.

[Partially done] Faster String concatenation

Bug v8:8939
CL 1499497

Explore ways to reduce overhead of string concatenation (and later flattening).

Flattened `ConsStrings` vs. `ThinStrings`

Bug v8:8930

When the concept of `ThinStrings` was introduced into V8 we didn't update the existing `ConsString` flattening to use `ThinStrings` instead. That means we still need to check

certain constraints before creating a `ConsString`. Specifically a `ConsString` cannot have its right side point to the empty string unless the left side is a `SeqString` or an `ExternalString` (as we still use this constellation to identify *flattened `ConsStrings`*). We should probably fix that, making it cheaper to construct `ConsStrings` without much checking upfront.

> **Background** (from jkummerow@) The background is simply that `ThinStrings` were introduced to make property lookups faster by giving us a way to internalize any string we want. Previously there were some strings that simply couldn't be internalized, forcing property load/store operations onto the slow path every single time they showed up as a key. Frameworks like Ember really like using dynamically computed strings as property keys. There's no hard reason why we couldn't use `ThinStrings` to replace flattened `ConsStrings`, and I'm not opposed to doing that. That said:
>
> 1. Simply creating `ThinStrings` (pointing to internalized strings as before) instead of flattened `ConsStrings` will put more load on the internalized string table, which may or may not have negative side effects. (That's why I haven't done that in the past. My thinking was: different problems, different solutions.)
> 2. Relaxing the invariant that `ThinStrings` always point to internalized strings would make `ThinString` handling more expensive; in particular load/store ICs would need another check on the type of the target string. You'd also have to modify a bunch of string handling code to ensure it is sufficiently generic to handle any combination of string pointers (e.g. currently, a `ThinString` can't refer to a `ConsString`, and we have code that doesn't check for that case -- hopefully there are enough DCHECKs in place ;-)).
>
> If your research indicates that doing 2. would drop more checks than it adds, then by all means go for it. :-)

[Idea] Conversions at bytecode generation

Source tweet

Like in `s += a + b + c` being compiled as roughly this:

- If a is a string, return `a + b + c`.
- If b is not a string, do `s + (a + b + c)`.
- Etc., if you'd like to specialize more generally.

It requires some computational lookahead, but it'd simplify string building.

[Idea] Automatic String Builders

Doc Automatic String Builders through Token-holding Sliced Strings

This is an earlier idea by verwaest@, which would use our `SlicedStrings` to build up some kind of automatic or implicit string builders.

Resources

1. Adventures in the land of substrings and RegExps