Optimizing JavaScript Standard Library Functions in JSC

Jul 28, 2021
by Tadeu Zagallo
@tadeuzagallo

After three years working on JavaScriptCore (JSC), I recently had the opportunity to work on optimizing one of our standard library functions for the first time. I thought it'd be interesting to share what I learned about how they work in JSC and how we make them faster.

How are standard library functions implemented in JSC?

The JavaScript standard library functions include all the prototype functions for Array, Object, etc, and in this post we'll look at `Function.prototype.toString`.

In JSC, standard library functions can either be implemented in native code (C++) or in JavaScript. When using JavaScript, we can use a special syntax for operations that are not allowed for regular JavaScript programs. Standard library functions written in JavaScript can be called like any other JavaScript function, and will also run through the same optimizations in each of our tiers, including inlining. Meanwhile, standard library functions written in C++ require the VM to make calls to native code, which is more expensive, and while the implementation will be optimized by the C++ compiler, its internals are completely opaque to the caller JS function, so there can be no inlining. The reason I mention inlining is that it allows the caller to “see” inside the function body, which unlocks many further optimizations to be done beyond the function boundary, some of which we’ll see in this example.

Making standard library functions faster

Let’s start with a reduced test case to see how can we optimize `Function.prototype.toString`:

```
function f() { /* ... some code here ... */ }

function g() {
  return f.toString();
}
```

When we call `g`, JSC will start by executing it in the interpreter (called LLInt), and if we continue calling it enough times it will eventually be promoted through the 3 compilers in JSC: the non-optimizing Baseline compiler and our optimizing DFG and FTL compilers. You can read more about our execution tiers on this post from back when the FTL was first introduced, but since then the FTL has gotten a new backend.

The first step before we run any code in any tier is to convert it from source code to bytecode (again, you can read a lot more about our bytecode here). After that, our DFG and FTL compilers also have their own representation of the code, that we call an Intermediate Representation or IR for short. For this post I’ll take the liberty of using some pseudocode that is somewhere in between our initial bytecode and DFG IR. This way we can see some of the details that are only available in DFG but without exposing too much low level complexity that is not relevant for our example. Here’s what our pretend IR could look like for `g`:

```
function g():
    f = Lookup("f") // look for the variable "f" in
    toString = Get(f, "toString") // access the "toS
                                  // respecting the
                                  // lookup, includi
    result = Call(toString, f) // call the toString
    Return(result) // return our result to the calle
```

Caching

The first optimization implemented wasn't specific to standard library functions at all: the result of calling toString on a function never changes, so we can cache it.

Our implementation of `Function.prototype.toString` is written in C++ and has to handle a few special cases. One such case is when we are calling `toString` on a native function implemented in C++, but for the common case where it's a regular JavaScript function, we have to look at the source code for the function. Since the source can't be changed while it's being executed, and the name of the function also cannot be changed, this result can be cached. This is completely transparent to our IR, which means we don't have to change anything in the compiler and we already get a great speedup.

Speculation

When we start trying to optimize any JavaScript program we quickly face several challenges, since pretty much everything in JavaScript can be mutated at runtime, including our standard library functions. One way that modern JavaScript VMs get around these challenges is by *speculating* that certain facts about the program will not change during its execution, but since we can't ever be sure of that, we need a way to fall back if our assumptions become invalid. We have an amazing post that goes deep into how we speculate in JSC, but for our current purposes all we need to know is that we can compile a function and say the code we generated is only valid so long as some assumptions are valid. This also isn't specific to standard library functions, but it'll be important later to see how all these optimizations interact with each other yielding more than the sum of their parts.

In this case we can speculate that `f` will never change, for our example let's assume that it's always called with an object allocated at address `0x123456`. This is done through a mechanism called watchpoints, which is beyond the scope of this post, but the TL;DR is that this code will immediately become invalid if anyone assigns to `f`, which would produce a different result. Using other watchpoints we can also speculate that `f.toString` will always resolve to `Function.prototype.toString` which already leads to much better code:

```
function g():
    f = 0x123456 // speculated value of Lookup("f")
    toString = Function.prototype.toString // specul
    result = Call(toString, f)
    Return(result)
```

Intrinsics

Next, JSC has the concept of *intrinsics*. Intrinsics in JSC are when the compiler utilizes the knowledge that it's calling a well known standard library function to emit optimized code inline. This is even more powerful than regular function inlining, since it can emit only the fast path inline and it works even if the standard library function was written in C++. In this case, the intrinsic tells us we are calling `Function.prototype.toString` and we can emit a new instruction, `FunctionToString`, instead of the generic function call. Our new pseudo instruction will try to load the cached value we computed in our very first optimization, but for the slow case, where we don't have cached value yet, it will still call the C++ implementation. Computing the `toString` for the first time requires some really complex code that we don't want to emit inline for every call to `toString`. Our code with our new instruction might look like this:

```
function g():
    f = 0x123456
    result = FunctionToString(f)
    Return(result)
```

And diving a little deeper, the implementation of our new `FunctionToString` instruction might look like the following:

```
function g():
    f = 0x123456
    result = Load(f, "cachedToString") // Load a spe
                                       // much faste
                                       // access, no
    if (!result) {
        // For the slow case where we don't have a c
        // to calling the C++ code
        toString = Function.prototype.toString
        result = Call(toString, f)
    }
    Return(result)
```

Abstract Interpretation

The next step in speeding up our example is using our Abstract Interpreter (AI). The idea behind the AI is that it understands what our IR instructions will do to its operands at runtime, and if all of its operands are known (i.e. we proved they will always have the same value), we can try to compute the result of our instruction at compile time. In this case, since we already speculated that `f` will be `0x123456`, when we're compiling our `FunctionToString` instruction we can try to load the `cachedToString` property from the object at `0x123456`. If that succeeds our generated code will look more like this:

```
function g():
    result = "function f() { /* … some code here ..
    Return(result)
```

That's much better!

Recap

We can speed up the `f.toString()` call in our example by:

- Caching the result
- Speculating `f` will always be the same object
- Speculating that `f.toString` will always resolve to `Function.prototype.toString`
- Adding an Intrinsic and a new instruction, `FunctionToString`, that loads the cached value directly whenever it's available
- Teaching our abstract interpreter that if we already know which function we are stringifying, and its `toString` value has already been computed, we can just use the cached value as a constant.

If you want to dive deeper into the actual implementation you can find the commit here and if you have any questions feel free to reach out to me on Twitter. ■

Next

PCM:
Click Fraud Prevention and

Attribution Sent to Advertiser

Learn more

Previously

Release Notes for Safari Technology Preview 128

Learn more