Performance tips for JavaScript in V8

- The Most Important Advice
- Hidden Classes
  - Therefore
- Numbers
  - Therefore
- Arrays
  - Therefore
- JavaScript Compilation
- The Full Compiler
  - Therefore
- The Optimizing Compiler
  - Therefore
- De-optimization
  - Therefore
- Other V8 Tools
- In Summary
- References

Daniel Clifford gave an excellent talk at Google I/O on tips and tricks to improve JavaScript performance in V8. Daniel encouraged us to "demand faster" - to carefully analyze performance differences between C++ and JavaScript, and write code mindfully of how JavaScript works. A summary of the most important points of Daniel's talk are captured in this article, and we'll also keep this article updated as performance guidance changes.

The Most Important Advice

It's important to put any performance advice into context. Performance advice is addictive, and sometimes focusing on deep advice first can be quite distracting from the real issues. You need to take a holistic view of the performance of your web application - before focusing on these performance tip, you should probably analyze your code with tools like PageSpeed and get your score up. This will help you avoid premature optimization.

The best basic advice for getting good performance in Web applications is:

- Be prepared before you have (or notice) a problem
- Then, identify and understand the crux of your problem
- Finally, fix what matters

In order to accomplish these steps, it can be important to understand how V8 optimizes JS, so you can write code mindful of the JS runtime design. It's also important to learn about the tools available and how they can help you. Daniel goes into some more explanation of how to use the developer tools in his talk; this document just captures some of the most important points of the V8 engine design.

So, on to the V8 tips!

Hidden Classes

JavaScript has limited compile-time type information: types can be changed at runtime, so it's natural to expect that it is expensive to reason about JS types at compile time. This might lead you to question how JavaScript performance could ever get anywhere close to C++. However, V8 has hidden types created internally for objects at runtime; objects with the same hidden class can then use the same optimized generated code.

For example:

```
function Point(x, y) {
  this.x = x;
  this.y = y;
}

var p1 = new Point(11, 22);
var p2 = new Point(33, 44);
// At this point, p1 and p2 have a shared hidden class
p2.z = 55;
// warning! p1 and p2 now have different hidden classes!```
```

Until the object instance p2 has additional member ".z" added, p1 and p2 internally have the same hidden class - so V8 can generate a single version of optimized assembly for JavaScript code that manipulates either p1 or p2. The more you can avoid causing the hidden classes to diverge, the better performance you'll obtain.

Therefore

- Initialize all object members in constructor functions (so the instances don't change type later)
- Always initialize object members in the same order

Numbers

V8 uses tagging to represent values efficiently when types can change. V8 infers from the values that you use what number type you are dealing with. Once V8 has made this inference, it uses tagging to represent values efficiently, because these types can change dynamically. However, there is sometimes a cost to changing these type tags, so it's best to use number types consistently, and in general it is most optimal to use 31-bit signed integers where appropriate.

For example:

```
var i = 42;  // this is a 31-bit signed integer
var j = 4.2;  // this is a double-precision floating point number```
```

Therefore

- Prefer numeric values that can be represented as 31-bit signed integers.

Arrays

In order to handle large and sparse arrays, there are two types of array storage internally:

- Fast Elements: linear storage for compact key sets
- Dictionary Elements: hash table storage otherwise

It's best not to cause the array storage to flip from one type to another.

Therefore

- Use contiguous keys starting at 0 for Arrays
- Don't pre-allocate large Arrays (e.g. > 64K elements) to their maximum size, instead grow as you go
- Don't delete elements in arrays, especially numeric arrays
- Don't load uninitialized or deleted elements:

```
for (var b = 0; b &lt; 10; b++) {
  a[0] |= b;  // Oh no!
}
//vs.
a = new Array();
a[0] = 0;
for (var b = 0; b &lt; 10; b++) {
  a[0] |= b;  // Much better! 2x faster.
}
```

Also, Arrays of doubles are faster - the array's hidden class tracks element types, and arrays containing only doubles are unboxed (which causes a hidden class change).However, careless manipulation of Arrays can cause extra work due to boxing and unboxing - e.g.

```
var a = new Array();
a[0] = 77;   // Allocates
a[1] = 88;
a[2] = 0.5;   // Allocates, converts
a[3] = true; // Allocates, converts```
```

is less efficient than:

```
var a = [77, 88, 0.5, true];
```

because in the first example the individual assignments are performed one after another, and the assignment of `a[2]` causes the Array to be converted to an Array of unboxed doubles, but then the assignment of `a[3]` causes it to be re-converted back to an Array that can contain any values (Numbers or objects). In the second case, the compiler knows the types of all of the elements in the literal, and the hidden class can be determined up front.

- Initialize using array literals for small fixed-sized arrays
- Preallocate small arrays (<64k) to correct size before using them
- Don't store non-numeric values (objects) in numeric arrays
- Be careful not to cause re-conversion of small arrays if you do initialize without literals.

JavaScript Compilation

Although JavaScript is a very dynamic language, and original implementations of it were interpreters, modern JavaScript runtime engines use compilation. V8 (Chrome's JavaScript) has two different Just-In-Time (JIT) compilers, in fact:

- The "Full" compiler, which can generate good code for any JavaScript
- The Optimizing compiler, which produces great code for most JavaScript, but takes longer to compile.

The Full Compiler

In V8, the Full compiler runs on all code, and starts executing code as soon as possible, quickly generating good but not great code. This compiler assumes almost nothing about types at compilation time - it expects that types of variables can and will change at runtime. The code generated by the Full compiler uses Inline Caches (ICs) to refine knowledge about types while program runs, improving efficiency on the fly.

The goal of Inline Caches is to handle types efficiently, by caching type-dependent code for operations; when the code runs, it will validate type assumptions first, then use the inline cache to shortcut the operation. However, this means that operations that accept multiple types will be less performant.

Therefore

- Monomorphic use of operations is preferred over polymorphic operations

Operations are monomorphic if the hidden classes of inputs are always the same - otherwise they are polymorphic, meaning some of the arguments can change type across different calls to the operation. For example, the second add() call in this example causes polymorphism:

```
function add(x, y) {
  return x + y;
}

add(1, 2);      // + in add is monomorphic
add("a", "b");  // + in add becomes polymorphic```
```

The Optimizing Compiler

In parallel with the full compiler, V8 re-compiles "hot" functions (that is, functions that are run many times) with an optimizing compiler. This compiler uses type feedback to make the compiled code faster - in fact, it uses the types taken from ICs we just talked about!

In the optimizing compiler, operations get speculatively inlined (directly placed where they are called). This speeds execution (at the cost of memory footprint), but also enables other optimizations. Monomorphic functions and constructors can be inlined entirely (that's another reason why monomorphism is a good idea in V8).

You can log what gets optimized using the standalone "d8" version of the V8 engine:

```
d8 --trace-opt primes.js
```

(this logs names of optimized functions to stdout.)

Not all functions can be optimized, however - some features prevent the optimizing compiler from running on a given function (a "bail-out"). In particular, the optimizing compiler currently bails out on functions with try {} catch {} blocks!

Therefore

- Put perf-sensitive code into a nested function if you have try {} catch {} blocks:

```
function perf_sensitive() {
  // Do performance-sensitive work here
}

try {
  perf_sensitive()
} catch (e) {
  // Handle exceptions here
}
```

This guidance will probably change in the future, as we enable try/catch blocks in the optimizing compiler. You can examine how the optimizing compiler is bailing out on functions by using the "--trace-opt" option with d8 as above, which gives you more information on which functions were bailed out:

```
d8 --trace-opt primes.js
```

De-optimization

Finally, the optimization performed by this compiler is speculative - sometimes it doesn't work out, and we back off. The process of "deoptimization" throws away optimized code, and resumes execution at the right place in "full" compiler code. Reoptimization might be triggered again later, but for the short term, execution slows down. In particular, causing changes in the hidden classes of variables after the functions have been optimized will cause this deoptimization to occur.

Therefore

- Avoid hidden class changes in functions after they are optimized

You can, as with other optimizations, get a log of functions that V8 had to deoptimize with a logging flag:

```
d8 --trace-deopt primes.js
```

Other V8 Tools

By the way, you can also pass V8 tracing options to Chrome on startup:

```
"/Applications/Google Chrome.app/Contents/MacOS/Google Chrome" --js-flags="--trace-opt --
trace-deopt"```
```

In addition to using the developer tools profiling, you can also use d8 to do profiling:

```
% out/ia32.release/d8 primes.js --prof
```

This uses the built-in sampling profiler, which takes a sample every millisecond and writes v8.log.

In Summary

It's important to indentify and understand how the V8 engine works with your code to prepare to build performant JavaScript. Once more, the basic advice is:

- Be prepared before you have (or notice) a problem
- Then, identify and understand the crux of your problem
- Finally, fix what matters

This means you should ensure the problem is in your JavaScript, by using other tools like PageSpeed first; possibly reducing to to pure JavaScript (no DOM) before collecting metrics, and then use those metrics to locate bottlenecks and eliminate the important ones. Hopefully Daniel's talk (and this article) will help you understand better how V8 runs JavaScript - but be sure to focus on optimizing your own algorithms, too!

References

- Daniel's talk on YouTube
- Daniel's slide deck
- I-want-to-optimize-my-JS-application-on-V8 checklist by Vyacheslav Egorov, V8 Engineer

---

□□□ #2
□ □□□ □□□ 2026-10-31 22:18:23 UTC
□ □□□ □□□ 2026-10-31 22:18:25 UTC