Static Decorators Early Feedback

Attention: Shared Google-externally

Author: syg@chromium.org

Last Updated: 2019-12-02

What “Static” Ought to Mean

The current static proposal is an attempt to improve startup performance concerns as raised by V8 engineers. For an implementation, I’d like the “static” qualifier to mean the following:

1. That there is a one-time cost that is paid during parsing or the initial compile.
    a. Decorators should have no per-class literal evaluation cost by design.
    b. Engines should be able to generate code for decorators during the initial compile. That is, modules or class bodies should **not** need to be reparsed/recompiled.
2. That the common use cases that look declarative actually act declarative. Once a module’s decorator imports are all loaded, what a decorator does should be knowable without running code. Further, decorators in the common case should not result in dynamism such as `Object.defineProperty`. Escape hatches should exist, but the common uses should be achievable declaratively.

Implementation Constraints

Motivations

There are (at least) three relevant implementation choices made by V8 and Chrome that are important to the design of decorator:

1. **ClassBoilerplate**. After a class literal is parsed, all { instance, static } x { methods, accessors } are gathered and put into a boilerplate structure that has the same lifetime as the bytecode. When the class is defined, the shape of the constructor and the prototype can be precomputed up to the point of all the methods and accessors.

    The important invariant here is that once the boilerplate is computed prior to evaluation, it can be left alone and be deep copied when e.g. passing around classes. Decorators must be static enough that the boilerplate would not need mutation per class literal evaluation, or that the initial boilerplate structure is unconditionally, immediately amended by calls to `Object.defineProperty`.

2. **Synthetic Members Initializer Functions**. All instance fields are collated into a synthesized function which is called per instance construction. Similarly, all static fields are collated into another function which is called when the class is defined. These functions are compiled eagerly when the class literal itself is compiled, as the AST for the class literal (and all its elements) will be discarded by evaluation time.

   The important invariant here is that, in the same vein as the invariant for the boilerplate, that once these initializer functions are compiled, they should remain undisturbed. Decorators must be static enough that the initializer functions do not need to be recompiled per class literal evaluation.

3. **Parallel Module Fetching, Parsing, and Bytecode Compilation.** A module's dependencies are fetched, parsed, and bytecode compiled in parallel. In other words, currently codegen is done on the unlinked module.

   The important invariant here is that static decorators should not require post-linking bytecode compilation for the entire module. Doing so creates a serialization point and requires either reparsing the entire module or keeping the AST around, both of which have unacceptable performance penalties.

Constraints

No Reshaping

Decorators, for the common use cases, should act declaratively. After decorators run and the class is defined, the initial instance shape should be stable (at least up to methods, because field initializers have always been "anything goes", like deleting other fields or freezing the instance). Concretely this means both that the ClassBoilerplate optimization must remain possible in light of decorators, and that common use of decorators must not result in code that immediately invalidates the boilerplate result.

Bytecode Compilation Should Be Done Once

When decorators remove or replace fields, public or private, with or without initializers, requires the initializer functions to be recompiled and the boilerplate to be fixed up. This can be accomplished several ways: reparsing the class literal, representing the class literal as IR, keeping ASTs around, bytecode patching, or rearchitecting how module graphs are loaded. After discussion with other V8 engineers, rearchitecting module loading is the most robust and palatable choice despite the significant amount of work required.

Currently modules are able to be compiled without knowing the rest of the module graph. In fact, a module's dependencies are not known until *after* it is compiled. To support static decorators without recompilation or reparsing, the module graph needs to be compiled in dependency order. If module A imports @foo from module B, B needs to be compiled before A so A has access to the definition of @foo. This suggests the following architecture:

1. Perform a fast preparse that discovers the dependencies of a module.
2. Perform step 1 until the entire module graph is discovered.
3. Topologically sort the graph so modules are parsed and compiled in the right order for static decorators.

Step 1 is an unknown. A full parse is required to extract the decorators from an import currently.

Step 3 raises the question of cyclic imports of static decorators. How would they work, if at all? For regular imports, circularity in the import itself is the issue. For static decorators, circularity in at the module level is problematic since the initial compile depends on it.

Skepticism

The design space for decorators seems to me like the following matrix. In matrix tree below, by "expressive", I mean supporting many disparate use cases: declarative class element transformation, imperative action on class elements, metadata, etc.

Dynamic Expressive	Static Expressive
Dynamic Restrictive	Static Restrictive

Dynamic Expressive is what was objected to when the proposal tried to advance to Stage 3.

V8 objected to Dynamic: by inviting programmers to use a declarative-looking feature only to have it not behave declaratively encourages writing non-performant programs, especially at startup.

JSC objected to Expressive: intrinsic to an expressive proposal is the complexity of reflecting the structures needed to enable that expressivity. In the Dynamic Expressive proposal, reflecting all the class elements became too heavyweight.

So, now here we are exploring the possibility of static decorators. It seems like performance wise, with enough engineering, a Static design *may* be possible to overcome V8's original objections. But can the feature satisfy both the generality and expressiveness constraints and the implementation constraints?

I believe Static Expressive will run afoul of the same complexity concerns. Expressivity for decorators seems to me to require either powerful reflection, like in Java's annotations, or powerful syntax transformation, like in languages with syntax macros, or a powerful "constexpr" DSL, which is probably morally equivalent to syntax macros.

That leaves Static Restrictive, like re-scoping the proposal to only consider the class element metaprogramming use case. I am skeptical this restriction of scope is possible or desirable for the end user. I'm skeptical this is possible because the mental model of restricted static decorators is elusive. Are they more like ad-hoc syntax macros? (But they can't be because many common use cases aren't expressible with declarative syntax, cf @readonly.) What about their scopes? In other words, because JS does not *have* a static metaprogramming model, the complexity cliff remains very, very tall even with use cases restricted. I'm skeptical this is desirable because, well, static metaprogramming doesn't jive with the rest of JS. I don't feel like we can avoid discussing staging, and the committee has come down on the question of staging before.

This design space is already a difficult one, and adding in production engines' constraints makes it all the more difficult. All of this leads me back to thinking about alternatives that do not require implementation in engines, such as the original idea of annotations and standardizing pure syntax, or attempt to break new ground by specifying a new stage of evaluation ("compilation"?). But that won't solve the dynamism-of-generated-code concern for the same reasons that the complexity cliff for the static case is high: we don't *have* declarative syntax for many common use cases, just `Object.defineProperty`.

Where does that leave us?