Fixing the scope chain for the implementation of private methods

Author: joyee@igalia.com
Last updated: March 4, 2020
Bug: https://bugs.chromium.org/p/v8/issues/detail?id=10098
CL: https://chromium-review.googlesource.com/c/v8/v8/+/2056889

The issue

Given a snippet like this

```
class Outer {
  #method() {}
  factory() {
    class Inner {
      constructor() {  }
    }
    return Inner;
  }
  run(obj) {
    obj.#method();
  }
}

const instance = new Outer();
const Inner = instance.factory();
instance.run(new Inner());
```

Since V8 only threads scopes that need `Context` (and `ScopeInfo`) at runtime, and here the `DeclarationScope` of `factory` and the `ClassScope` of `Inner` do not need that, the serialization and deserialization of the scope chain would result in an information loss. We have a deserialized scope chain like this when generating the code for the constructor of `Inner`:

```
global { // (0x10b001248) (-1, -1)
  // 2 heap slots
```

```
  class { // (0x10b001440) (-1, -1)
    // strict mode scope
    // 2 heap slots
    // local vars:
    CONST .brand;  // (0x10b0015a8) context[3]
    // brand var:
    CONST .brand;  // (0x10b0015a8) context[3]

    function Inner () { // (0x10b0016f0) (93, 100)
      // strict mode scope
      // will be compiled
    }
  }
}
```

Where the `ScopeInfo` of the `DeclarationScope` for the `Inner` constructor is supposed to be “optimized away”. Then effectively the scope chain we can restore at code generation time is equivalent to

```
class Outer {
  #method() {}
  constructor() {  }  // Inner’s constructor
  run(obj) {
    obj.#method();
  }
}
```

With this, the instances constructed out of the `Inner` constructor would erroneously have access to `Outer`'s private methods, as if it is `Outer`’s constructor. Therefore, when generating bytecode for the `Inner` constructor, we need to find another way to know that we cannot rely on the immediate outer scope of `Inner` to check whether the brand initialization is necessary - or, more generally, that there are intermediate scopes omitted during scope chain, serialization and deserialization.

Similarly, the scope chain is also unreliable at debug time/runtime either, so we cannot tell whether a class constructor contains static methods/accessors or not from its scope chain. For context, this issue was discovered in https://chromium-review.googlesource.com/c/v8/v8/+/1955664

This issue does not apply to the private fields, however, because those are guarded by a `requires_instance_members_initializer` field on the `SharedFunctionInfo`, which are always allocated(?) and do not suffer from information loss this way, therefore the bytecode generator knows not to emit private field initialization code for constructors of classes without private fields.

Solutions

~~Idea 1: Use the ScopeInfo of the constructor's DeclarationScope~~

The issues comes from the fact that the scopes of the constructors are not threaded in the way they appear in the source code. ~~And in the example above, the `DeclarationScope` of `Inner` does not have `ScopeInfo` allocated either, so this would not work.~~ The JSFunctions do always have their own `ScopeInfo` allocated, no matter the scope is empty or not, but the empty scopes are not threaded into the `ScopeInfo` chain (that is, at code generation time, they cannot be accessed as `outer_scope()` through the scopes pointed by AST nodes). Nonetheless, they are still accessible if the corresponding SFI(`SharedFunctionInfo`)s can be located.

One plausible solution would be, instead of saving the information in the `ClassScope`s, which are only accessible as `outer_scope()` of the constructors (and we know this link is broken), try pushing the information into the `DeclarationScope`s of the constructors, and use it directly at code generation time. A possible medium to store this information is the `ScopeInfo` of the constructor's `DeclarationScope`. However we realized that the `ScopeInfos` are only created themselves after code generation, so this would not be viable.

~~Idea 2: Use the preparse data to pass the information~~

Since the `ScopeInfo`s are created too late, then another possible medium would be the preparse data, since they are created before the bytecode generation. This might be done by locating and updating the scopes of the class constructors in `RewriteClassLiteral()` with information regarding whether the class contains any private methods.

However, we realized that this still would not work, since the preparse data is created too early - right after the corresponding function is parsed, and before `RewriteClassLiteral()` is called. This means if the declaration of the first private

method appears after the declaration of the constructor, then we would not be able to set the bit in the preparse data for the constructor correctly since cannot anticipate the existence of the private method at that point.

~~Idea 3: Reuse the private_name_lookup_skips_outer_class bit on SFI.~~

As described earlier, the bits on SFI do not suffer from information loss. The `private_name_lookup_skips_outer_class` on SFI is used to annotate classes like `Inner` in the following snippet:

```
class Outer extends class Inner {
  constructor() { this.#method; }
} {
  #method
}
```

Here in the scope chain we have at code-generation time, `Outer`'s ClassScope is the outer scope of `Inner`'s `ClassScope`. Although this bit is only used up until scope analysis time, and after that it is not used at code-generation time as the failures would be emitted as early errors.

See v8:9177 and the implementation doc for details.

This bit is similar to the bit that we need to fix the issue at hand, in that it also denotes "the outer class scope on the scope chain is not trust-worthy" - for private name resolution.

But looking closer, this bit essentially carries different information from what we need here. When the outer class scope on the scope chain is not trust-worthy for private name resolution, it can be inferred that the class is not trust-worthy for private brand initialization either:

```
class Outer extends class Inner {
  constructor() { this.#method; }
} {
  #method() {}
}
```

It is not the other way around, however - even when the outer class scope on the scope chain is not trust-worthy for private brand initialization, it can still be trust-worthy for private name resolution:

```
class Outer {
  #method() {}
  factory() {
    class Inner {
      constructor() { this.#method()  }
    }
    return Inner;
  }
}
```

Therefore, a new bit seems necessary. However we have already used up all the bits in SFI's 32-bit field and adding one more bit would lead to additional memory overhead to all SFI and thus undesirable. If we do want to use SFI to solve this issue, we need to squeeze a bit out of SFI's bitfield.

~~Idea 4: Using UncompiledData to avoid using bits on SFI~~

CL: https://chromium-review.googlesource.com/c/v8/v8/+/2032626

This was explored since the available bits on the SFI are scarce and extending the size of the bitfields on the SFI would be costly, as SFI themselves are allocated a lot.

To reduce the memory impact, a possible solution would be to put the bits into the `UncompiledData` pointed to by the SFIs. As its name implies, `UncompiledData` are only kept before code generation and are discarded after that, so increasing the size of them would be less costly than increasing the size of SFIs.

This bit can be maintained with the following modifications:

1. Take a bit on the `FunctionLiteral` AST nodes to store the information about whether its a class constructor that needs private brand initialization.
2. Add a bitfield to `UncompiledData` to store similar information - note that due to the alignment requirement, this means we actually have to increase the size of `UncompiledData` by 64 bits in the worst case
3. In `RewriteClassLiteral` of both the preparse and the full parser, if we detect that the class needs a private brand, mark the bit in the AST node of the constructor

4. In `SharedFunctionInfo::InitFromFunctionLiteral()`, where we create the `UncompiledData` and the AST is still accessible, pass the information from the AST to the the `UncompiledData`
5. When reparsing the constructor in `Parser::ParseFunction()` to generate bytecode, update the AST with the `UncompiledData` pointed by the SFI available there. We have to fixup the inner AST like this since the class scope surrounding the constructor is not reparsed along with it, so we need to get this information from the previously saved `UncompiledData`.
6. When generating `ScopeInfo` for the constructors after bytecode generation, we also need to save an additional bit on the `ScopeInfo`, because in the current pipeline, if an error is thrown at runtime and the source positions need to be allocated for the stack trace, the bytecode needs to be regenerated with an AST with `ScopeInfo` and a SFI whose `UncompiledData` is already flushed, and the regenerated bytecode has to match the code previously generated with `UncompiledData`. So for consistency `ScopeInfo` need to maintain this information as well.

This is still not very ideal, as there needs to be a 31-bit or 63-bit padding for reasons mentioned in 2, and it is tricky to maintain side-channel information this way.

Idea 5: Squeezing some bits out from SFI

CL: https://chromium-review.googlesource.com/c/v8/v8/+/2056889

To go back using bits on SFI, one possible approach is to free up the `IsClassConstructor` bit on SFI using the information available in `FunctionKind`, similar to what's done in https://chromium-review.googlesource.com/c/v8/v8/+/1482915. However, this could lead to performance regressions since then in the machine code generated to detect whether a function is a class constructor, we would have to do a range check instead of simply masking bits off.

Eventually we found that `expected_nof_properties` on SFI currently takes 16 bits, which is unnecessary since the value of it is capped at `JSObject::kMaxInObjectProperties` which is 256 at the moment. So it should be safe to shrink this field to 8 bits, freeing up 8 more bits for us to pass the information about private brands around.

With a bit on the SFI, the information can be maintained as follows:

1. We add a second 8-bit bit field to the SFI with one bit used to maintain the information we need
2. After the entire class is parsed in the full parser, if it contains any private instance methods, we mark a bit in the `FunctionLiteral` AST node for the class constructor
3. In `SharedFunctionInfo::InitFromFunctionLiteral()` this bit is passed from the AST to the SFI
4. In `ParseInfo::ParseInfo` this bit is passed from SFI to `ParseInfo`
5. When reparsing the constructor for code generation in `Parser::DoParseFunction()` we pass the bit from the `ParseInfo` to the AST (we have to obtain this information this way here since only the constructor is reparsed, and the surrounding class is not)
6. To update the SFI created in lazy compilation that is kept at runtime, we also pass this bit from the AST to SFI again in `SetSharedFunctionFlagsFromLiteral()`

With approach we could also fix the issue for inspecting static private methods at runtime using another bit from the freed bits.