Optimizing class instance member initializers in V8

Attention - this doc is public and shared with the world!

Contact: Joyee Cheung <joyee@igalia.com>, Caitlin Potter <caitp@igalia.com>
Contributors:
Status: Inception | **Draft** | Accepted | Done

LGTMs needed

Name	**Write (not) LGTM in this row**
leszeks@chromium.org	
verwaest@chromium.org	
...	

Objective

To improve performance of the class instance member initialization in V8 and promote usage of these features.

Microbenchmark patch: https://chromium-review.googlesource.com/c/v8/v8/+/2999030

Background

Currently, when a instance member initializer is defined in a class, like this

```
const a = 1;
class A {
  #a = a;
  #b = this.#a;
  constructor() {
    // does work
  }
}
```

```
new A();
```

V8 generates the bytecode for the initializer as an instance member function:

```
// Load private name symbol for #a into r1
LdaImmutableCurrentContextSlot [2]
Star r1
// Load a into r2
LdaImmutableContextSlot <context>, [2], [1]
ThrowReferenceErrorIfHole [0]
Star r2
Mov <this>, r0
// Store the value of a with private name symbol #a into the instance
// that is, #a = a;
CallRuntime [AddPrivateField], r0-r2
// Load private name symbol for #b into r1
LdaImmutableCurrentContextSlot [3]
Star r1
// Load private name symbol for #a
LdaImmutableCurrentContextSlot [2]
// Load the value of #a from the instance into r2
LdaKeyedProperty <this>, [0]
Star r2
Mov <this>, r0
// Store the value of #a into #b, that is, #b = this.#a;
CallRuntime [AddPrivateField], r0-r2
LdaUndefined
Return
```

which is stored into the class as a symbol property at class evaluation time.

```
...
CallRuntime [DefineClass], r3-r5
Star r3
CreateClosure [5], [1], #0
Star r4
StaNamedProperty r2, [6], [0]
```

When the constructor of the class is invoked, V8 loads the initializer from the class (the JSFunction) and invokes it.

```
// Load the initializer from the constructor
LdaNamedProperty <closure>, [0], [0]
```

```
JumpIfUndefined [11]
Star r1
// Invoke the initializer
CallProperty0 r1, <this>, [2]
Mov <this>, r0
LdaUndefined
Return
```

Design

Inline initializers in the class constructor

See https://bugs.chromium.org/p/v8/issues/detail?id=9888#c6

WIP at https://chromium-review.googlesource.com/c/v8/v8/+/2944249

Note: the WIP improves the performance of class evaluation by about 30-50%, and the performance of class instantiation by less than 10%, this is probably because:

- We are only removing a CreateClosure - StaNamedProperty sequence for class evaluation and a LdaNamedProperty - CallProperty0 sequence for the instance constructor
- The CallProperty0 would be inlined when it goes to TurboFan

Currently we create a synthetic instance member initializer function when the class is evaluated. For example with this class:

```
class A {
  #a = 1;
  constructor() { }
  getA() { return this.#a; }
  #b = 2;
}
```

At class evaluation time, we generate a function from a `InitializeClassMembersStatement` AST node that contains a list of `ClassLiteralProperty` including the initializers for `#a` and `#b`, and we record its source code as:

```
  #a = 1;
  constructor() { }
  getA() { return this.#a; }
  #b = 2;
```

Which is actually invalid (i.e. can't be reparsed as a function) but it works most of the time, since we always generate code for the synthetic function when the class is evaluated, at that point we'll just use the `InitializeClassMembersStatement` AST node created when the entire class is parsed (this is done eagerly, that is, even if the constructor is never invoked). The code for the initializer function is usually alive as long as the class is alive, so they won't normally be lazily parsed. However with `FunctionCodeHandling::kClear` that is no longer true so the snapshot support is currently broken.

Currently the initializer function can already be inlined in the constructor by TurboFan. We can inline the `InitializeClassMembersStatement` AST node into the constructor `FunctionLiteral`'s body so that when generating the bytecode for baseline, we do not have to load and call a synthetic initializer function. The design doc about public class fields has described what it takes to implement this..

Reparsing the class body from constructor

As explained above, the code we currently record for initialization is invalid and can't be reparsed. Inlining the initializers into the constructor means they may be lazily parsed along with the constructor, so we will reparse the entire class body when the constructor is invoked to collect the initializers. When the initialization is done in the derived constructor, we will take the initialization statements out from the constructor and visit them right after super() is invoked so the fields are initialized in time.

Note that even if we reparse the class body, we can skip the non-field members (like `getA()` above) so the additional overhead should be limited. We also get to fix the snapshot support bug if lazy parsing of the initializers is made possible.

Disabling new.target in the initializers

Within the initializers, `new.target` is supposed to be undefined. We can implement this by temporarily assigning `undefined` to the new.target variable before the initialization statements are visited, and restoring it back later in the bytecode generator. We may even optimize further and skip this dance if `new.target` is never used in the initializers (either directly or through things like eval).

Making it work with the step-debugger

To make sure that step-debugging continues to work, given a source position that's somewhere in the initializer, we need to be able to tell that it is associated with the constructor SFI.

The current plan is to always store the class body positions into the ScopeInfo of the class scope if it needs member initialization. When looking for candidate SFIs for a position within the initializers, if we can tell that a SFI is a constructor with member initializers (available in the flags), we look into the ScopeInfo of its outer class scope, and include the SFI if the current position falls into the class body. Then when choosing the closest SFI we patch the calculation and make sure that the constructor SFI is chosen when necessary.

Reduce the cost of runtime calls (e.g. AddPrivateField)

See https://bugs.chromium.org/p/v8/issues/detail?id=10793

Currently we invoke `CreateDataProperty`, `AddPrivateField` and `AddPrivateBrand` as runtime calls in the constructor when they are just a property existence check and a keyed property load, and this incurs performance overhead.

We could implement this without using a runtime call. Some options include:

~~1. Use existing bytecodes to implement the runtime calls~~

~~Add two new bytecodes that allow loading/storing owned keyed properties while bypassing the interceptors and traps. Then we can replace `AddPrivateField` etc. with bytecodes generated using these new operations.~~ Instead of

```
CallRuntime [AddPrivateField], r0-r2
```

We may perform something like

```
// Load the private name symbol into r1
LdaImmutableCurrentContextSlot [2]
Star r1
// Load the hole into r2
LdaTheHole
Star r2
// Check that the private field isn't already defined
Ldar r1
Mov <this>, r0
LdaKeyedProperty r0
TestReferenceEqual r2
JumpIfFalse
LdaSmi
Star
CallRuntime [NewTypeError]
```

```
// ...do whatever that saves the private field value into the accumulator...
// Store the value into the instance using the private name symbol
Ldar r1
Mov <this>, r0
StaKeyedProperty r0, r1
```

However, we can only implement the private field operations with this approach. We can't just use `LdaNamedProperty/StaNamedProperty` for public fields because that would result in [[Set]] semantics instead of [[Define]] semantics. This also complicates LdaKeyedProperty a bit since right now it simply throws on missing private symbol properties but to reuse it for initialization some special paths need to be added so that it returns a hole or undefined for initialization.

2. Add new bytecodes or change existing bytecodes for defining owned public properties

To make [[Define]] semantics work for public fields, we could add two new bytecodes, or change existing `StaNamedProperty` so that they take arguments to ignore interceptors/traps when storing properties.

Patch adding a new bytecode and ICs for the define operation:
https://chromium-review.googlesource.com/c/v8/v8/+/2795831/ (landed)

The preferred approach is to add new bytecode `StaKeyedPropertyAsDefine`, which skips prototype lookups and traps when adding the property:

```
LdaImmutableCurrentContextSlot [2]
Star0
LdaImmutableContextSlot <context>, [2], [1]
ThrowReferenceErrorIfHole [0]
StaKeyedPropertyAsDefine <this>, r0, [0]
LdaImmutableCurrentContextSlot [3]
Star0
LdaImmutableCurrentContextSlot [2]
LdaKeyedProperty <this>, [2]
StaKeyedPropertyAsDefine <this>, r0, [4]
LdaUndefined
Return
```

This would call into a new IC, with a slot in the feedback per field denoting that [[Define]] semantics is intended.

~~3. Implement inlined intrinsics~~

We can implement inlined intrinsics for `AddPrivateField` etc. This way we don't have to add or change the bytecodes.

4. New ICs

We can add ICs for `AddPrivateField` and friends, this may be done together with option 3 or 2.

This has been done in https://chromium-review.googlesource.com/c/v8/v8/+/2795831/

The new IC can share most of its code with KeyedStoreIC. We differentiate the semantics of the operation via a feedback slot indicating that [[Define]] semantics is intended.

To perform existence checks for private names, checks need to be in place in:

- KeyedStoreIC::Store (to handle IC misses or in case the map is deprecated)
- StoreIC::Store (reused by KeyedStoreIC)

For the KeyedStoreIC::Store case, we can add a new runtime function Runtime::DefineClassField similar to the existing Runtime::SetObjectProperty to perform the checks we need.

The new IC do need a different slow stub and different transition handling to avoid being leaked into StoreIC's megamorphic stub cache. We also need to skip generating the prototype handlers in the AccessorAssembler, and go to the slow stub when the private property being defined already exists (CL).

Elide the brand and private field existence checks for base classes

Take this class for example:

```
const a = 1;
class A {
  #a = a;
  #b = this.#a;
  #c() {}
  constructor() {
    // does work
  }
}
new A();
```

We currently perform 3 existence checks in the constructor: 1 for the private brand (since there's a private method `#c`), and 2 for the private fields `#a` and `#b`. Theoretically, if the class is a base class, the checks can be merged into one (if the class extends from other classes, this may not be true when the initializer throws and we are left with a partially initialized object).

One possible solution is to always declare and check a brand on the instance (we only do this once even if there are multiple private methods present), and remove the existence checks in the private field definitions. This means for classes with only private fields but no private methods, there would be the additional overhead of storing a brand property in the instances, in return the initialization would be faster (and the more private fields there are, the worthier this is).

Adjusting the initial map based on class field initializers

When people initialize the fields, the static information available in the initializers tend to be useful to determine how the initial map should look like for the class. This is particularly true for private fields since users have to declare them in advance.

Things that could be adjusted using the information we know from the field initializers:

- initial in-object property size
- initial property count
- …(more to think about)

Take this class from Node.js core for example:

```
class Event {
  #type = undefined;
  #defaultPrevented = false;
  #cancelable = false;
  #timestamp = lazyNow();

  #bubbles = false;
  #composed = false;
  #propagationStopped = false;
}
```

We know at least that there would be 7 private fields and we know that 5 of them are booleans. The #type field is intended to be a string, and #timestamp is something returned from a function. We could increase the initial in-object property size and property count accordingly if we are certain the initial slack isn't enough.

Other ideas...