Refactoring private name lookup

Author: Joyee Cheung (joyee@igalia.com)

Background

https://bugs.chromium.org/p/v8/issues/detail?id=7468

We need to separate the lookup of private names from normal variables to speed them up and to avoid expensive creation of synthetic strings.

Strategy

ClassScope

Create a subclass `ClassScope` with an optional list of unresolved private names and a private name map:

```
class ClassScope : public Scope {
  struct RareData : public ZoneObject {
    explicit RareData(Zone* zone) : private_name_map(zone) {}
    base::ThreadedList<VariableProxy, VariableProxy::UnresolvedNext>
      unresolved_private_names;
    VariableMap private_name_map;
  };

  RareData* rare_data_ = nullptr;
}
```

Here we put the private name map and unresolved private names in `RareData` to save memory for classes without private members.

Declare private names

When a private name declaration is encountered, create a `VariableProxy` and add it to the private name map of the class scope. If there is already a `VariableProxy` with the same name in the map, throw an error. Note that private names declared in inner classes can shadow those with the same name in the outer classes.

Tracking unresolved private names

Whenever a private name access is encountered, find the closest outer class scope, and push a variable proxy to its `unresolved_private_names` .

If there are no class scope in the scope chain, throw an error.

Resolving private names

~~Updating `Scope::AnalyzePartially` for preparsed functions~~

Updating `Parser::SkipFunction` for preparsed methods

Because the preparser resets the zone where unresolved private name proxies may be created after preparsing a method, we need make sure that when resolving the private names after parsing the class literal later, the variable proxies in the unresolved list of the class scope are still all in the correct zone.

~~When `Scope::AnalyzePartially` is called during parsing, since the outer class scope may not be complete yet, we simply find the closest enclosing class scope, and copy all the unresolved private names to the correct zone, then save the new list in the closest class scope.~~

When preparsing methods in `Parser::SkipFunction`, we save the current tail of the unresolved private name list. After the method is preparsed, if there is an error, we reset the tail to the previous state. If the method is preparsed successfully, we migrate the private names added between the save tail and the new tail to the correct zone.

Updating `DeclarationScope::Analyze` during code generation

When

- A previously preparsed method is reparsed to generate code because it's called
- eval is called to access private names

We need to resolve the private names from the class scope closest to the `DeclarationScope`s of these methods/evaled code.

When `DeclarationScope::Analyze` is called, which calls `DeclarationScope::AllocateVariables` to resolve variables parsed from the method.

1. Find the closest outer class scope of the `DeclarationScope`
2. If there are unresolved private names in the closest class scope, then it should come from current method (because we are not doing a full reparse of the entire class, just this method alone)
3. Resolve all unresolved private names in this class scope recursively using the scope info of all class scopes in the scope chain.

~~Question: is it okay to add unresolved private names again in the deserialized scope when doing a full parse of the function for code generation?~~ It is fine to push unresolved private names to the closest class scope during a full parse, because we are not doing a full reparse of the entire class, just this method alone, so all unresolved private names should come from the current method.

Eager resolution when parsing class literals

Whenever a class literal finishes parsing, we can eagarly resolve all unresolved private names inside. For names that cannot be resolved in the current class scope, push them to the closest outer class scope - note that since the outer class scope is not yet complete, the private name may still be resolvable later.

If there are no more outer class scopes, throw an error.

Questions:

1. ~~Will we encounter lazily parsed functions inside the class scopes whose unresolved proxies need to be migrated to a new zone later?~~ It is possible and can be solved by saving the tail of the unresolved private names before preparsing a method, then migrate every private names added after that to the correct zone after the method is preparsed.
2. ~~Is it safe to just store the remaining unresolved private names that cannot be resolved in this class scope in the outer class scope?~~ It is safe to just push the unresolved private names to the outer scope.

Analysis of examples

(In the following examples, when we refer to a class name e.g. `C`, we are referring to its associated `ClassScope`. And when we refer to a function, we may be referring to its `DeclarationScope`)

Private names declared after access in methods & declared before access in initializer

```
class C {
  getB() { return this.#b; }
  #a = 1;
  #b = this.#a;
}
new C().getB();
```

With lazy parsing

1. When parsing `getB()`, add unresolved #b to C
2. After preparsing `getB()`, migrate `#b` in C's unresolved private names to the correct zone (This is why we need to update `Parser::SkipFunction()`)
3. When parsing `#a = 1`, declare `#a` in C
4. When parsing `#b = this.#a`, declare #b in C, add unresolved #a to C
5. After parsing the class literal, resolve #b and #a with the private name map of C
6. After preparsing, save the local variables (#a and #b) of C in scope info
7. `new C().getB()` triggers a full parse of `getB` (This is why we need to update `DeclarationScope::Analyze`)
8. When allocating variables for getB, resolved #b is pushed to the closest class scope C, find it in the scope info, and redeclare it in C's private name map

Without lazy parsing

1, 3, 4, 5 above but all done in the full parser

Accessing already declared private names

```
class C {
  #b = 1;
```

```
  getB() { return this.#b; }
}

new C().getB();
```

With lazy parsing

1. When parsing `#b = 1`, declare #b in C
2. When parsing `getB()`, add #b to C's unresolved private names
3. After preparsing `getB()`, migrate `#b` in C's unresolved private names to the correct zone.
4. After parsing the class literal, resolve `#b` in getB()
5. After preparsing, save the local variable #b in C's scope info
6. `new C().getB()` triggers a full parse of `getB()`. When parsing `getB()`, add #b to C's unresolved private names
7. When allocating variables for `getB()`, resolve #b pushed to the closest class scope C, find it in the scope info, and redeclare it in C's private name map

Without lazy parsing

1, 2, 4 above but done in the full parser instead.

Accessing undeclared private names

```
class C {
  getB() { return this.#b; }
  #a = 1;
}
```

With lazy parsing

1. When parsing `getB()`, add unresolved #b to C
2. After preparsing `getB()`, migrate #b in C's unresolved private names to the correct zone
3. When parsing `#a = 1`, declare #a in C
4. After parsing the class literal, resolve #b, we could not find it in C's private name map and there's no outer class scope, so throw an error.

Without lazy parsing

1, 3, 4 above but all done in full parser

Accessing private names when there is an error

```
class C {
  #a = 1;
  getB() {
    this.#a = class D { #a; #a; }
  }
}
```

With lazy parsing

1. When parsing the `#a = 1` in `class C`, declare `#a` in the private name map of C
2. When parsing `this.#a` in `getB()`, push #a to C's unresolved private name list
3. When parsing the first `#a` in `class D`, declare `#a` in the private name map
4. When parsing the second `#a` in `class D`, since it can already be found in the private name map, throw an variable redeclaration error which is unidentifiable by the preparser
5. After preparsing `getB()` since there is an unidentifiable error, reset the unresolved private name list to the state before preparsing getB(), so the list is reset to empty
6. Reparse and error

Without lazy parsing

TBD

Accessing private names declared in outer class scope

```
class X {
  fn() {
    class C {
      getB(obj) { return obj.#b; }
    }
    return (new C).getB(this);
  }
  #b = 1;
}

new X().fn();
```

With lazy parsing

1. When parsing `getB()`, add unresolved #b to C
2. After parsing C, try resolving the private names in C, cannot find #b in C, so move it to outer class scope X
3. After preparsing `fn()`, migrate #b in X's unresolved private names to the correct zone
4. When parsing #b = 1, declare #b in X's private name map
5. After parsing X, resolve #b and find it in X's private name map
6. After preparsing, save the local variable #b in X's scope info
6. `new X().fn()` triggers a full parse of `fn()`. When parsing `getB()`, add unresolved #b to C
7. When allocating variables for `getB()`, resolve #b recursively, and find it in X's scope info.

Without lazy parsing

1, 2, 4, 5 above but all done in full parser

Accessing private names not declared in current or outer class scope

```
class X {
  fn() {
    class C {
      getB(obj) { return obj.#b; }
    }
    return (new C).getB(this);
  }
}

new X().fn();
```

With lazy parsing

1. When parsing `getB()`, add unresolved #b to C
2. After parsing C, try resolving the private names, cannot find it in C, so move it to outer class scope X
3. After preparsing `fn()`, migrate #b in X's unresolved private names to the correct zone
4. After parsing X, try to resolve #b and throw an error

Without lazy parsing

1, 2, 4 above

Accessing private names declared in outer class scope from methods inside initializer

```
class X {
  #a = class { getB(obj) { return obj.#b; } }
  fn() {
    return (new (this.#a)()).getB(this);
  }
  #b = 1;
}

new X().fn();
```

With lazy parsing

1. When parsing `getB()`, add unresolved #b to the anonymous class
2. After parsing the anonymous class, try resolving the private names, cannot find it in the class, so move it to outer class scope X
3. declare #a in X's private name map
4. When parsing fn, add unresolved #a to X
5. After parsing fn, migrate #a to the correct zone
6. When parsing `#b = 1`, declare #b in X's private name map
7. After parsing X, resolve #b and find it in X's private name map
8. After preparsing, save the local variable #b in X's scope info
9. `new X().fn()` triggers a full parse of `fn()`.
10. When parsing `fn()`, add unresolved #a to X
11. Resolve #a from X's scope info
12. `.getB(this)` triggers a full parse of `getB()`
13. add unresolved #b to X
14. Resolve #b from X's scope info

Without lazy parsing

1, 2, 3, 4, 6, 7 above but all done in the full parser

Accessing private names from eval()

```
class C {
  #a = 1;
  getA() { return eval('this.#a'); }
}
(new C).getA();
```

With lazy parsing

1. When parsing `#a = 1`, declare #a in C's private name map
2. After parsing C, save the scope info of C
3. When executing eval inside `(new C).getA()`, parse `this.#a`
4. Add unresolved #a to the closest class scope C
5. When analyzing the scope of `eval()`'ed code, we resolve the unresolved private name #a in the closest class scope C, and find it from C's scope info

Without lazy parsing

Same, but done in full parser

Accessing undeclared private names from eval()

```
class C {
  getA() { return eval('this.#a'); }
}

(new C).getA();
```

1. After parsing C, save the scope info of C
2. When executing eval inside `(new C).getA()`, parse `this.#a`, add unresolved `#a` to the closest class scope C
3. When analyzing the scope of eval, we resolve unresolved the private names in the closest class scope C, but cannot find it and throw an error

Without lazy parsing

Same, but done in full parser