Allow Oilpan in V8

Attention: Shared Google-externally

Status: Work in progress
Author(s): Michael Lippautz
Tracking bug: v8:13754

LGTMs

Name	Write (not) LGTM in this row
Leszek Swirski	LGTM
Adam Klein	LGTM

Proposal

Allow Oilpan (`cppgc`) types to be used in V8 internally and on V8's API surface. Currently such types are prohibited from being used via a basic DEPS rule. The dependency makes CppHeap mandatory for a V8 isolate. See below for instantiation details.

E.g., this would allow V8 to expose base classes that it could use itself that embedders can extend and work with.

```
C/C++
namespace v8 {

class ResourceBase : public cppgc::GarbageCollectedMixin {
 public:
  void Trace(cppgc::Visitor* visitor) const { visitor->Trace(v8_resource_);
}

 protected:
  v8::TracedReference<v8::Data> v8_resource_;
};

}  // namespace v8
```

V8 could then keep references to such objects internally, staying within GC semantics (see limitations). The main idea is that this can be used to more easily model lifetimes involving JS objects on the embedder side. E.g., this could avoid passing around roots (`v8::Global`) on the API surface which is prone to creating memory leaks.

Similar requests came up in the past in design discussions (sorry, Google-only).

Background

Oilpan is implemented in V8 and is integrated with V8's JavaScript garbage collector to provide the unified heap where objects with JS and C++ object graphs. Oilpan infrastructure is implemented behind the `cppgc` namespace and available through V8's API.

There's currently different ways to configure and use V8/Oilpan:

A. Unified heap (JS+Oilpan): Blink
B. V8 JS-only: Node
C. Stand-alone Oilpan: PDFium

The high level unified heap design (case A) suggests that JS and C++ heaps already go hand in hand. In practice, this is not the case as Blink requires an Oilpan heap early on during initialization before a V8 isolate is set up. The initialization sequence in Blink is as follows:

1. Initialize Oilpan (`CppHeap`)
2. Allocate objects on `CppHeap`
3. Initialize `Isolate`
4. Attach `CppHeap` to `Isolate`

In 1. and 2. `CppHeap` is in a detached state where we allow allocations but don't perform any garbage collection (except in testing configurations where the CppHeap is treated as stand-alone Oilpan heap).

3. and 4. are separate steps to allow for configuration A. In Blink there should be no allocations (and garbage collections) of any sort in between these steps (although it would be supported to do a JS-only collection as the heaps are still independent).

Changes

Dependencies

Allowing internals and API types to depend on Oilpan implies that it is not optional anymore. While initialization makes this transparent, all embedders start using Oilpan automatically this way.

Cost:

- There's minimal runtime performance cost in the GC that has to consider objects on an empty `CppHeap`. This cost should be negligible.
- There's cost in allocating a virtual memory cage of 4G for initializing a `CppHeap`. If this becomes a problem, we could lazily set up the cage on first allocation.

Initialization

`v8::Isolate` construction receives an additional parameter (via some sort) to a `CppHeap`. If `CppHeap` is not provided, V8 will create one implicitly that is available via `v8::Isolate::GetCppHeap()`.

A path for moving the API forward

Since migrating a class should go through regular V8 deprecation cycles, wrapper objects can be used to temporarily forward calls to malloced objects.

C/C++
```
namespace v8 {

// Malloced and currently in use by the embedder.
class OldFooBase {
 public:
  virtual void Bar() = 0;
};

// The new GC'ed API we want.
class NewFooBase : public GarbageCollectedMixin {
 public:
  virtual void Bar() = 0;
};

// Non public wrapper class.
class NewFooWrapper : public GarbageCollected<NewFoo>, NewFooBase {
 public:
  NewFoo(unique_ptr<OldFoo> foo) : foo_(std::move(foo)) {}

  void Bar() override { foo_->Bar(); }

  void Trace(cppgc::Visitor* visitor) const {}

 private:
  unique_ptr<OldFoo> foo_;
};

// Old API.
V8_DEPRECATE_SOON("Use RegisterFoo(NewFooBase*)")
void RegisterFoo(unique_ptr<OldFoo> old_foo) {
  auto* new_foo =
    MakeGarbageCollected<NewFooWrapper>(
      GetAllocationHandle(), std::move(old_foo));
  RegisterFoo(new_foo);
}

// New API.
void RegisterFoo(NewFooBase* new_foo_base) {
   // ...
}

}  // namespace v8
```

Limitations

JS to C++ references

JS to C++ references are currently modeled via wrapper/wrappable semantics using embedder fields. To directly point to C++ objects from regular JS objects another primitive needs to be added to the V8 GC.

Helpers

Blink provides a bunch of helper utilities to aid writing Oilpan code that are not directly available to V8:

- Clang plugin that helps finding mistakes in Oilpan usage: The plugin is built into the clang binary that is shipped via llvm-build. We could probably also use it on V8 code.
- Allocation helpers: Oilpan doesn't know about threads. Blink uses globals/TLS to pass around a heap handle that is used for allocation.
- Helpers around wrapping objects in roots.

Collections

As of today, the Oilpan core provided through V8 does not ship any collections. These are purely implemented in Blink with specific API helpers where necessary. Blink collections include vectors and hashmaps with support for weakness and even ephemerons. Moving (at least some of) them to Oilpan's core is a long-standing TODO for the team.