v8windbg

V8windbg is a WinDbg extension for the V8 engine. It adjusts the behavior of the Locals pane and corresponding `dx` commands to display useful data when inspecting V8 object types. It is intended to be as robust as possible in dumps with limited memory, and should work equally well in live sessions, crash dumps, and time travel debugging.

Building

Run `autoninja v8windbg` in your output directory.

Using

In WinDbgX, run `.load path\to\your\output\dir\v8windbg.dll` to load the extension. To inspect V8 objects, use the Locals window or the `dx` command as usual.

Important notes:

- The version of v8windbg must exactly match the version and build configuration of the process you're debugging. (To find the version number of a module in a crash dump, enter `lm` and click the module name, or run `lmDvm modulename`.)
- V8windbg relies on detailed symbols (symbol_level = 2).
- Ensure also that WinDbg can load the symbols (.pdb file) for the module containing V8.
- Cross-architecture debugging is possible in some cases:
  - To debug an x86 process on x64, load the x86 build of v8windbg.
  - To debug an ARM64 process on x64, load the ARM64 simulator build of v8windbg (built with target_cpu="x64" and v8_target_cpu="arm64").

As well as improving the Locals pane behavior, v8windbg also provides a few functions that can be called from within `dx` commands:

- `@$v8object()` returns information about the fields of a tagged V8 value, passed in as a plain number like `dx @$v8object(0x34f49880471)`. This invokes the same logic that is used for the locals pane. You may also pass a type hint as an optional second parameter if you find that v8windbg is not inferring the correct type (which can happen when the memory for the object's Map wasn't collected in a crash dump). The type hint is a fully-qualified C++ class name, like `dx @$v8object(0x34f49880471, "v8::internal::JSArray")`.
- `@$curisolate()` gets the Isolate pointer for the current thread, if the current thread has a JavaScript Isolate associated.
- `@$jsstack()` returns a list of the JS stack frames, including information about script and function.

Tip:: to see what objects are present in a chunk of heap memory, you can cast it to an array of `TaggedValue`, like this:

```
dx (v8::internal::TaggedValue(*)[64])0x34f49880450
```

Architecture

V8windbg uses the DataModel as much as possible as opposed to the older DbgEng APIs. It uses the WRL COM APIs due to limitations in Clang's support for C++/WinRT COM.

Where possible, v8windbg uses the cross-platform v8_debug_helper library to avoid depending on V8 internals.

The source in `./base` is a generic starting point for implementing a WinDbg extension. The V8-specific implementation under `./src` then implements the two functions declared in `dbgext.h` to create and destroy the extension instance.

`./src` file index:

- `cur-isolate.{cc,h}` implements the `IModelMethod` for `@$curisolate()`.
- `js-stack.{cc,h}` implements the `IModelMethod` for `@$jsstack()`. Its result is a custom object that supports iteration and indexing.
- `local-variables.{cc,h}` implements the `IModelPropertyAccessor` that provides content to show in the Locals pane for stack frames corresponding to builtins or runtime-generated code.
- `object-inspection.{cc,h}` contains various classes that allow the debugger to show fields within V8 objects.
- `v8-debug-helper-interop.{cc,h}` makes requests to the V8 postmortem debugging API, and converts the results into simple C++ structs.
- `v8windbg-extension.{cc,h}` is responsible for initializing the extension and cleaning up when the extension is unloaded.

When the extension is initialized (`Extension::Initialize()`):

- It registers a "parent model" for all known V8 object types, such as `v8::internal::HeapObject` and `v8::internal::Symbol`. Any time WinDbg needs to represent a value with one of these types, it creates an `IModelObject` representing the value and attaches the parent model. This particular parent model supports `IStringDisplayableConcept` and `IDynamicKeyProviderConcept`, meaning the debugger will call a custom method every time it wants to get a description string or a list of fields for any of these objects.
- It registers a different parent model, with a single property getter named "Value", for handle types such as `v8::internal::Handle<*>`. The "Value" getter returns the correctly-typed tagged pointer contained by the handle.
- It overrides the getter functions for "LocalVariables" and "Parameters" on the parent model for stack frames. When the user selects a stack frame,

WinDbg calls these getter functions to determine what it should show in the Locals pane.
- It registers the function aliases such as `@$curisolate()`.

The `./test` directory contains a test function that exercises v8windbg. It does not require WinDbg, but uses DbgEng.dll and DbgModel.dll from the Windows SDK (these are slightly older versions of the same modules used by WinDbg). The test function launches a separate d8 process, attaches to that process as a debugger, lets d8 run until it hits a breakpoint, and then checks the output of a few `dx` commands.

Debugging the extension

To debug the extension, launch a WinDbgx instance to debug with an active target, e.g.

```
windbgx \src\github\v8\out\x64.debug\d8.exe -e "console.log('hello');"
```

or

```
windbgx \src\github\v8\out\x64.debug\d8.exe c:\temp\test.js
```

The WinDbgx process itself does not host the extensions, but uses a helper process. Attach another instance of WinDbgx to the `enghost.exe` helper process, e.g.

```
windbgx -pn enghost.exe
```

Set a breakpoint in this second session for when the extension initializes, e.g.

```
bm v8windbg!DebugExtensionInitialize
```

..and/or whenever a function of interest is invoked, e.g.

- `bp v8windbg!CurrIsolateAlias::Call` for the invocation of `@$curisolate()`
- `bp v8windbg!GetHeapObject` for the interpretation of V8 objects.

Load the extension in the target debugger (the first WinDbg session), which should trigger the breakpoint.

```
.load "C:\\src\\github\\v8windbg\\x64\\v8windbg.dll"
```

Note: For D8, the below is a good breakpoint to set just before any script is run:

```
bp d8_exe!v8::Shell::ExecuteString
```

..or the below for once the V8 engine is entered (for component builds):

```
bp v8!v8::Script::Run
```

Then trigger the extension code of interest via something like `dx source` or `dx @$curisolate()`.