

Debugging support in V8

Søren Gjesse, Software Engineer

sgjesse@google.com

Debugger facilities

- Stop execution of the debuggee
- Break points from source positions
- Programmatic break points
- Single stepping
- Inspection of call stack
- Inspection of global variables
- Evaluation of expressions
- Heap inspection

Debugging native applications

- Native applications are debugged out of process
- "Simple" API's for talking to the debuggee
- Normally using "int 3" instruction (one byte opcode "0xCC") to handle break points
- Use CPU single stepping support to continue leaving the break point set
- Changing the return address on the stack is another option
- Single stepping using CPU support
- Additional debug info linking code with source
 - DWARF (ELF), stabs, PDB (Windows), ...

Windows debugger API

- DebugActiveProcess/DebugActiveProcessStop
- Run a loop with WaitForDebugEvent
 - Receive events for create process, create thread, exit process, exit thread, load dll, unload dll, exception
- Use Windows API to read/write process memory
- How to stop
 - Suspend all threads (priority issues)
 - Inject int 3 instructions (clear instruction cache)
 - Resume all threads and wait
 - Handle deadlock/message loop

Linux debugger API

- System call `ptrace`
 - Attach to running process or trace child process
 - Receive signals on events in traced process
 - Read/write memory of traced process
 - Read/write process information
 - Read/write CPU registers
 - Single stepping

Debugging JavaScript in V8

- V8 generates native code
 - No interpreter to hook into
- V8 is part of the embedding application
- Design criteria
 - No pay for no use
 - No need to enable (“just in time”)
 - No speed degrading when debugging
 - Use JavaScript to implement as much as possible

Breaking V8 execution

- Piggy-bag on stack overflow check
 - All functions start with a stack overflow check
 - All backwards branches contains a stack overflow check
 - The current stack limit is stored in a static variable
 - Raise the stack limit to its maximum to generate a break
- Cannot break until executing JavaScript
 - In Chrome javascript:void(0) can be executed to ensure JavaScript execution
- Programmatic break using **debugger** statement

Function entry

```
function f() {...
```

```
00 55          push ebp
01 8bec        mov ebp, esp
03 56          push esi
04 57          push edi
05 3b256cc56500  cmp esp, [address_of_stack_limit]
11 0f8305000000  jnc 22
17 e816f3feff    call STUB, StackCheck
22 ...
```

Backwards branch

```
... while (true) { .. } ..
```

```
...
```

```
22 3b256cc56500    cmp esp, [address_of_stack_limit]
```

```
28 0f8305000000    jnc 39
```

```
34 e819dffeff      call STUB, StackCheck
```

```
39 ...
```

```
XX YYYY          jmp 22
```

Source position info

- Use the relocation info to link source positions to the generated code
- During parsing position information is collected in the AST
 - Two types of positions: statement position and position
- During code generation this position information is added to the relocation info
- When setting break points these positions are scanned for the closest code position
 - NOTE: The generated code is not linear from the source

Breakpoints

- Break points are set in native code
- Patch code at selected places where register content is well known
 - Calls to stubs (mostly inline cache stubs)

```
e80355ffff    call XXX
```

- Function return

```
8be5        mov esp, ebp
5d          pop ebp
c20400      ret 0x4
```

Breakpoints

- Relocation info indicate all stub calls
- Code moving supports changing stub calls
- Additional relocation info for return code
 - Added GC support for return patched
- Create a copy of the code before patching
 - Use copy to restore code
 - Use copy to run the right code when returning from debugger
 - Update the copy with IC transitions to keep this running as before

Breakpoints and GC

- At the break location some registers might contain objects
 - Knowing this precisely which is required
- As most of the debugger is written in JavaScript GC can happen
- When continuing register values might not be the same as before

Getting from break to JavaScript

- Copy all registers to a static memory area
- Build a new JavaScript frame on the stack
- Push the registers containing object pointers to the stack
- Build a special transition to C++ frame where all registers are pushed
- Control ends up in the runtime system
- Setup JavaScript object for execution state
- Call the debug event listener with the event and the execution state

And getting back again

- When control is back in the runtime system determine where to continue
 - In the copy of the code if break point is still installed
 - In the actual code if break point was removed during break point processing
- When exiting from C++ the register values pushed are copied to a static memory area
- When exiting the additional JavaScript frame overwrite the registers containing pointers
- Restore all registers from the static memory area
- Jump to the address where to continue

Stepping

- Handle stepping by flooding functions with breakpoints
 - Current function
 - Top exception handler
 - Functions looked up for inline cache misses
- For each step it is checked whether a new source position is reached

Break points from source

- Script break points
 - Based on script "name" and position
 - Added each time the same source is compiled again (e.g. page reloaded in Chrome)
- When scripts are added through the API "name" and offset can be supplied

Web page excerpt

```
...  
<script>  
  var mcount = 0  
  for (var i = 0; i < 10; i++) {  
    alert(i);  
  }  
</script>  
<div onclick='f()' onmouseover='mcount++'>  
  ...  
</div>  
<script>  
  function f() {  
    ...  
  }  
</script>  
...
```

Inspecting the stack

- When a break is hit execution ends up in JavaScript
- Adding primitives to the VM to allow traversal of the execution stack
 - GetFrameCount
 - GetFrameDetails

Evaluating expressions

- Evaluate a piece of JavaScript in the context of any stack frame
- Create a new context with contains all the locals and parameters from the stack frame
- Compile a function which evals the the expression
 - `function (arguments, __source__) {
 return eval (__source__);
}`
- Used for for conditional break points

Mirrors

- Mirrors are reflection objects exposed to the debugger
- Provides more details than normal JavaScript reflection
 - Property attributes
 - Exposes additional objects like StackFrame and Script
- Implemented in JavaScript using some additional VM primitives
 - DebugGetProperty, DebugInterceptorInfo, DebugLocalPropertyNames, DebugLocalElementNames, ...
- Serializable as JSON

Mirror hierarchy

- Mirror
 - ValueMirror
 - UndefinedMirror
 - NullMirror
 - NumberMirror
 - StringMirror
 - ObjectMirror
 - FunctionMirror
 - UnresolvedFunctionMirror
 - ArrayMirror
 - DateMirror
 - RegExpMirror
 - ErrorMirror
 - PropertyMirror
 - InterceptorPropertyMirror
 - AccessorMirror
 - FrameMirror
 - ScriptMirror

Heap traversal

- Ensure only live objects in the heap
 - Perform 1-2 GC's
- Find "active" script code
- Find objects referenced by a given object (implemented on mirror)
- Find objects constructed by a given function (implemented on mirror)

Debugger API

- Two different API's
- One in process function based
 - C or JavaScript callback registered
- One message based (can be used out of process)
 - Send a command by passing in a string
 - C message handler receiving a string registered
- A DebugBreak function to stop execution

Debugger API

```
typedef void (*DebugEventCallback) (DebugEvent event,  
                                     Handle<Object> exec_state,  
                                     Handle<Object> event_data,  
                                     Handle<Value> data);
```

```
typedef void (*DebugMessageHandler) (const uint16_t* message,  
                                     int length,  
                                     void* data);
```

Message based protocol

```
{ "seq" : <number>,
  "type" : "event",
  "event" : "break",
  "body" : { "invocationText" : <text representation of the stack frame>,
            "sourceLine" : <source line where execution is stopped>,
            "sourceColumn" : <column within the source line>,
            "sourceLineText" : <text for the source line>,
            "script" : { name : <resource name of the ...>
                       lineOffset : <line offset within ...>
                       columnOffset : <column offset within ...>
                       lineCount : <number of lines in the script>
                     }
            "breakpoints" : <array of break point numbers hit if any>
          }
}

{ "seq" : <number>,
  "type" : "request",
  "command" : "setbreakpoint",
  "arguments" : { "type" : <"function" or "script">
                "target" : <function expression or script identification>
                "line" : <line in script or function>
                "position" : <character position within the line>
                "enabled" : <initial enabled state. True or false, default is true>
                "condition" : <string with break point condition>
                "ignoreCount" : <number specifying the number of break point hits to ignore, ...>
              }
}
```

Future

- Debugger UI in Chrome
- Profiler
 - Sample based (cheap)
 - Instrumented (expensive)
 - Flay/hierarchical
- Data break points
- Heap statistics