

08 | 答疑：如何构建和使用V8的调试工具d8?

2020-04-02 李兵

《图解 Google V8》

课程介绍 >



讲述：李兵

时长 19:52 大小 18.20M



你好，我是李兵。

今天是我们第一单元的答疑环节，课后有很多同学留言问我关于 d8 的问题，所以今天我们就来专门讲讲，如何构建和使用 V8 的调试工具 d8。

d8 是一个非常有用的调试工具，你可以把它看成是 debug for V8 的缩写。我们可以使用 d8 来查看 V8 在执行 JavaScript 过程中的各种中间数据，比如作用域、AST、字节码、优化的二进制代码、垃圾回收的状态，还可以使用 d8 提供的私有 API 查看一些内部信息。

如何通过 V8 的源码构建 D8?

通常，我们没有直接获取 d8 的途径，而是需要通过编译 V8 的源码来生成 d8，接下来，我们就先来看看如何构建 d8。



其实并不难，总的来说，大体分为三部分。首先我们需要先下载 V8 的源码，然后再生成工程文件，最后编译 V8 的工程并生成 d8。

接下来我们就来具体操作一下。考虑到使用 Windows 系统的同学比较多，所以下面的操作，我们的默认环境是 Windows 系统，Mac OS 和 Liunx 的配置会简单一些。

安装 VPN


V8 并不是一个单一的版本库，它还引用了很多第三方的版本库，大多是版本库我们都无法直接访问，所以，在下载代码过程中，你得先准备一个 VPN。

下载编译工具链：depot_tools

有了 VPN，接下来我们需要下载编译工具链：**depot_tools**，后续 V8 源码的下载、配置和编译都是由 depot_tools 来完成的，你可以直接点击下载：[🔗 depot_tools bundle](#)。

depot_tools 压缩包下载到本地之后，解压压缩包，比如你解压到以下这个路径中：

```
1 C:\src\depot_tools
```


 复制代码

然后将这个路径添加到环境变量中，这样我们就可以在控制台中使用 gclient 了。

设置环境变量

接下来，还需要往系统环境变量中添加变量 DEPOT_TOOLS_WIN_TOOLCHAIN，值设为 0。

```
1 DEPOT_TOOLS_WIN_TOOLCHAIN = 0
```

 复制代码

这个环境变量的作用是告诉 deppt_tools，使用本地已安装的默认的 Visual Studio 版本去编译，否则 depot_tools 会使用 Google 内部默认的版本。

然后你可以在命令行中测试下是否可以使用：



```
1 gclient sync
```

安装 VS2019

在 Windows 系统下面，depot_tools 使用了 VS2019，因为 VS2019 自带了编译 V8 的编译器，所以需要安装 VS2019 时，安装时，你需要选择以下两项内容：

- Desktop development with C++；
- MFC/ATL support。

因为编译 V8 时，使用了这两项所提供的基础开发环境。

下载 V8 源码

安装了 VS2019，接下来就可以使用 depot_tools 来下载 V8 源码了，具体下载命令如下所示：

```
1 d:
2 mkdir v8
3 cd v8
4 fetch v8
5 cd v8
```

执行这个命令就会开始下载 V8 源码，这个过程可能比较漫长，下载时间主要取决于你的网速和 VPN 的速度。



```
命令提示符 - fetch v8
D:\>cd v9
D:\V9>fetch v8
Running: 'C:\Users\bzero\.vpython-root\3f2d72\Scripts\python.exe' 'C:\depot_tools\gclient.py' root
Running: 'C:\Users\bzero\.vpython-root\3f2d72\Scripts\python.exe' 'C:\depot_tools\gclient.py' config --spec 'solutions =
[
  {
    "url": "https://chromium.googlesource.com/v8/v8.git",
    "managed": False,
    "name": "v8",
    "deps_file": "DEPS",
    "custom_deps": {},
  },
]
Running: 'C:\Users\bzero\.vpython-root\3f2d72\Scripts\python.exe' 'C:\depot_tools\gclient.py' sync --with_branch_heads
1>_____ running 'git -c core.deltaBaseCacheLimit=2g clone --no-checkout --progress https://chromium.googlesource.com/
v8/v8.git D:\V9\_gclient_v8_mta8pt' in 'D:\V9'
1>Cloning into 'D:\V9\_gclient_v8_mta8pt'...
1>remote: Sending approximately 656.90 MiB ...
1>remote: Counting objects: 7675, done
1>remote: Finding sources: 100% (15/15)
1>Receiving objects: 68% (510924/743844), 497.09 MiB | 1.26 MiB/s
```

配置工程

代码下载完成之后，就需要配置工程了，我们使用 gn 来配置。

复制代码

```
1 cd v8
2
3
4 gn gen out.gn/x64.release --args='is_debug=false target_cpu="x64" v8_target_cpu="'
```

如果是 Mac 系统，你可以使用：

复制代码

```
1 gn gen out/gn --ide=xcode
```

用它来生成工程。

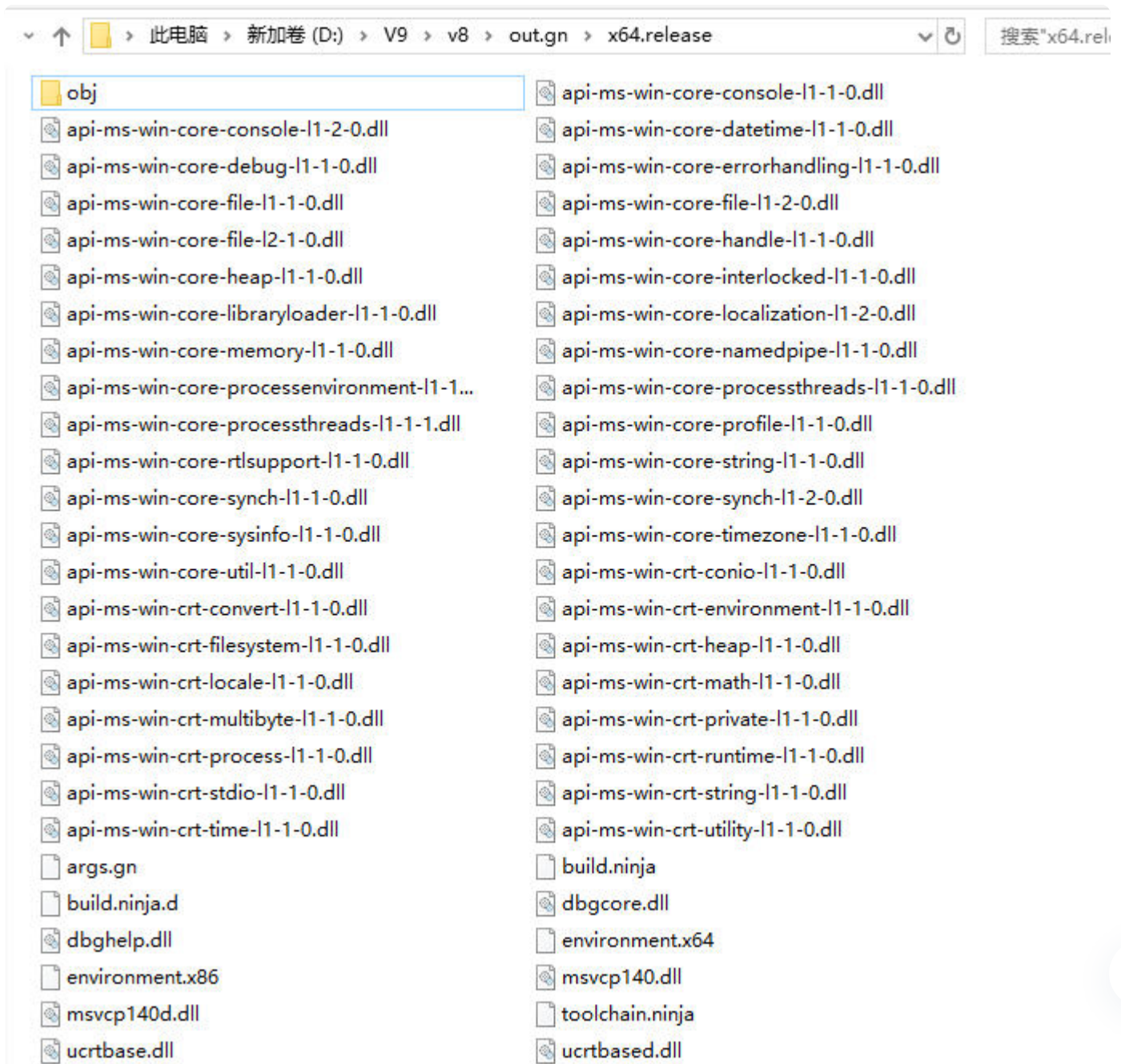
gn 是一个跨平台的构建系统，用来构建 Ninja 工程，Ninja 是一个跨平台的编译系统，比如可以通过 gn 构建 Chromium 还有 V8 的工程文件，然后使用 Ninja 来执行编译，可以使用 gn 和 Ninja 来配合使用构建跨平台的工程，这些工程可以在 MacOS、Linux、Windows 等平台上进行编译。在 gn 之前，Google 使用了 gyp 来构建，由于 gn 的效率更高，所以现在都在使用 gn。



下面我们来看下生成 V8 工程的一些基础配置项：

- is_debug = false 编译成 release 版本；
- is_component_build = true 编译成动态链接库而不是很大的可执行文件；
- symbol_level = 0 将所有的 debug 符号放在一起，可以加速二次编译，并加速链接过程；
- ide = vs2019 ide=xcode。

工程生成好之后，你就可以去 v8\out.gn\x64.release 这个目录下查看生成的工程文件。如下图所示：




生成了 d8 的工程配置文件，接下来就可以编译 d8 了，你可以使用下面的命令：

 复制代码

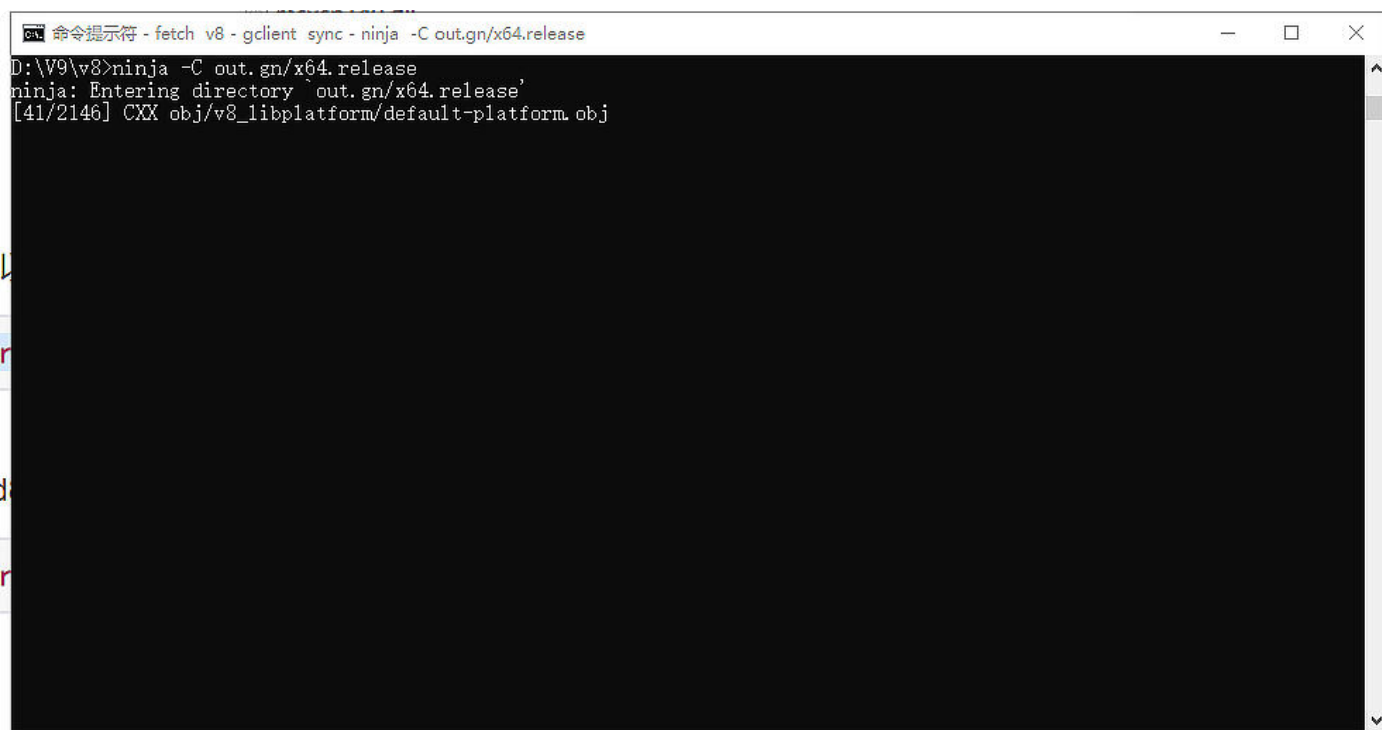
```
1 ninja -C out.gn/x64.release
```

如果想编译特定目标，比如 d8，可以使用下面的命令：

 复制代码

```
1 ninja -C out.gn/x64.release d8
```

这个命令只会编译和 d8 所依赖的工程，然后就开始执行编译流程了。如下图所示：



A screenshot of a Windows command prompt window. The title bar reads "命令提示符 - fetch v8 - gclient sync - ninja -C out.gn/x64.release". The command prompt shows the following text: "D:\V9\v8>ninja -C out.gn/x64.release", "ninja: Entering directory `out.gn/x64.release'", and "[41/2146] CXX obj/v8_libplatform/default-platform.obj". The rest of the terminal window is black, indicating the compilation process is ongoing or has completed.

编译时间取决于你硬盘读写速度和 CPU 的个数，比如我的电脑是 10 核 CPU，ssd 硬盘，整个编译过程大概花费了 15 分钟。

最终编译结束之后，你就可以去 v8\out.gn\x64.release 查看生成的文件，如下图所示：



此电脑 > 新加卷 (D:) > V9 > v8 > out.gn > x64.release >					搜索"x64.release"
名称	修改日期	类型	大小		
zlib_bench.exe.pdb	2020/3/28 9:30	Program Debug...	744 KB		
environment.x64	2020/3/28 9:25	X64 文件	5 KB		
environment.x86	2020/3/28 9:25	X86 文件	5 KB		
bytecode_builtins_list_generator.exe	2020/3/28 9:30	应用程序	79 KB		
ccctest.exe	2020/3/28 9:42	应用程序	23,076 KB		
cppgc_unittests.exe	2020/3/28 9:30	应用程序	667 KB		
d8.exe	2020/3/28 9:39	应用程序	217 KB		
generate-bytecode-expectations.exe	2020/3/28 9:39	应用程序	92 KB		
gen-regexp-special-case.exe	2020/3/28 9:30	应用程序	33 KB		
inspector-test.exe	2020/3/28 9:39	应用程序	98 KB		
mkgrokdump.exe	2020/3/28 9:39	应用程序	116 KB		
mksnapshot.exe	2020/3/28 9:39	应用程序	29,969 KB		
torque.exe	2020/3/28 9:30	应用程序	1,919 KB		
torque-language-server.exe	2020/3/28 9:30	应用程序	2,049 KB		
unittests.exe	2020/3/28 9:41	应用程序	9,751 KB		
v8_hello_world.exe	2020/3/28 9:39	应用程序	21 KB		
v8_sample_process.exe	2020/3/28 9:39	应用程序	41 KB		
v8_shell.exe	2020/3/28 9:39	应用程序	33 KB		
v8_simple_json_fuzzer.exe	2020/3/28 9:39	应用程序	26 KB		
v8_simple_multi_return_fuzzer.exe	2020/3/28 9:39	应用程序	49 KB		
v8_simple_parser_fuzzer.exe	2020/3/28 9:39	应用程序	28 KB		
v8_simple_regexp_builtins_fuzzer.exe	2020/3/28 9:39	应用程序	54 KB		
v8_simple_regexp_fuzzer.exe	2020/3/28 9:39	应用程序	34 KB		
v8_simple_wasm_async_fuzzer.exe	2020/3/28 9:39	应用程序	69 KB		
v8_simple_wasm_code_fuzzer.exe	2020/3/28 9:39	应用程序	71 KB		
v8_simple_wasm_compile_fuzzer.exe	2020/3/28 9:39	应用程序	202 KB		
v8_simple_wasm_fuzzer.exe	2020/3/28 9:39	应用程序	69 KB		
wasm_api_tests.exe	2020/3/28 9:41	应用程序	26,226 KB		
zlib_bench.exe	2020/3/28 9:30	应用程序	63 KB		

我们可以看到 d8 也在其中。

我将编译出来的 d8 也放到了网上，如果你不想编译，也可以点击 [这里](#) 直接下载使用。

如何使用 d8?

好了，现在我们编译出来了 d8，接下来我们将 d8 所在的目录，v8\out.gn\x64.release 添加到环境变量“PATH”的路径中，这样我们就可以在控制台中使用 d8 了。

我们先来测试下能不能使用 d8，你可以使用下面这个命令，在控制台中执行 d8：



```
1 d8 --help
```


复制代码

最终显示出来了一大堆命令，如下所示：

```
1. zsh
--trace-creation-allocation-sites (trace the creation of allocation sites)
  type: bool  default: false
--print-code (print generated code)
  type: bool  default: false
--print-opt-code (print optimized code)
  type: bool  default: false
--print-opt-code-filter (filter for printing optimized code)
  type: string default: *
--print-code-verbose (print more information for code)
  type: bool  default: false
--print-builtin-code (print generated code for builtins)
  type: bool  default: false
--print-builtin-code-filter (filter for printing builtin code)
  type: string default: *
--print-regexp-code (print generated regexp code)
  type: bool  default: false
--print-regexp-bytecode (print generated regexp bytecode)
  type: bool  default: false
--print-builtin-size (print code size for builtins)
  type: bool  default: false
--sodium (print generated code output suitable for use with the Sodium code viewer)
  type: bool  default: false
--print-all-code (enable all flags related to printing code)
  type: bool  default: false
--predictable (enable predictable mode)
  type: bool  default: false
--predictable-gc-schedule (Predictable garbage collection schedule. Fixes heap growing, idle, and memory reducing behavior.)
  type: bool  default: false
--single-threaded (disable the use of background tasks)
  type: bool  default: false
--single-threaded-gc (disable the use of background gc tasks)
  type: bool  default: false
eos@libingdeMacBook-Pro 06 %
```

我们可以看到上图中打印出来了 many 行，每一行其实都对应着一个命令，比如 `print-bytecode` 就是查看生成的字节码，`print-opt-code` 是要查看优化后的代码，`turbofan-stats` 是打印出来优化编译器的一些统计数据的命令，每个命令后面都有一个括号，括号里面是介绍这个命令的具体用途。

使用 d8 进行调试方式如下：


 复制代码

```
1 d8 test.js --print-bytecode
```

d8 后面跟上文件名和要执行的命令，如果执行上面这行命令，就会打印出 `test.js` 文件所生成的字节码。



不过，通过 `d8 --help` 打印出来的列表非常长，如果过滤特定的命令，你可以使用下面的命令来查看：

 复制代码

```
1 d8 --help |grep print
```

这样我们就能查看 d8 有多少关于 print 的命令，如果你使用了 Windows 系统，可能缺少 grep 程序，你可以去 [这里](#) 下载。

安装完成之后，记得手动将 grep 程序所在的目录添加到环境变量 PATH 中，这样才能在控制台使用 grep 命令。

最终打印出来带有 print 字样的命令，包含以下内容：

```
命令提示符
Microsoft Windows [版本 10.0.18362.720]
(c) 2019 Microsoft Corporation。保留所有权利。

C:\Users\bzero>d8 --help |grep print
--print-bytecode (print bytecode generated by ignition interpreter)
--print-bytecode-filter (filter for selecting which functions to print bytecode)
--print-deopt-stress (print number of possible deopt points)
--turbo-stats (print TurboFan statistics)
--turbo-stats-nvp (print TurboFan statistics in machine-readable format)
--turbo-stats-wasm (print TurboFan statistics of wasm compilations)
--trace-wasm-memory (print all memory updates performed in wasm code)
--print-wasm-code (Print WebAssembly code)
--print-wasm-stub-code (Print WebAssembly stub code)
--trace-gc (print one trace line following each garbage collection)
--trace-gc-nvp (print one detailed trace line in name=value format after each garbage collection)
--trace-gc-ignore-scavenger (do not print trace line after scavenger collection)
--trace-idle-notification (print one trace line following each idle notification)
--trace-idle-notification-verbose (prints the heap state used by the idle notification)
--trace-gc-verbose (print more details following each garbage collection)
--trace-gc-freelists (prints details of each freelist before and after each major garbage collection)
--trace-gc-freelists-verbose (prints details of freelists of each page before and after each major garbage collection)
--trace-allocation-stack-interval (print stack trace after <n> free-list allocations)
--trace-duplicate-threshold-kb (print duplicate objects in the heap if their size is more than given threshold)
--trace-mutator-utilization (print mutator utilization, allocation speed, gc speed)
--fuzzer-gc-analysis (prints number of allocations and enables analysis mode for gc fuzz testing, e.g. --stress-marking, --stress-scavenge)
--trace-serializer (print code serializer trace)
--external-reference-stats (print statistics on external references used during serialization)
--trace-side-effect-free-debug-evaluate (print debug messages for side-effect-free debug-evaluate for testing)
--max-stack-trace-source-length (maximum length of function source code printed in a stack trace.)
--use-idle-notification (Use idle notification to reduce memory footprint.)
--use-verbose-printer (allows verbose printing)
--stack-trace-on-illegal (print stack trace when an illegal exception is thrown)
--print-all-exceptions (print exception object and stack trace on each thrown exception)
--print-ast (print source AST)
--print-scopes (print scopes)
--gc-verbose (print stuff during garbage collection)
--print-handles (report handles after GC)
--print-global-handles (report global handles after GC)
--trace-normalization (prints when objects are turned into dictionaries.)
--print-break-location (print source location on debug break)
--print-opt-source (print source code of optimized and inlined functions)
--print-code (print generated code)
--print-opt-code (print optimized code)
--print-opt-code-filter (filter for printing optimized code)
--print-code-verbose (print more information for code)
--print-builtin-code (print generated code for builtins)
--print-builtin-code-filter (filter for printing builtin code)
--print-regexp-code (print generated regexp code)
--print-regexp-bytecode (print generated regexp bytecode)
--print-builtin-size (print code size for builtins)
--sodium (print generated code output suitable for use with the Sodium code viewer)
--print-all-code (enable all flags related to printing code)

C:\Users\bzero>
```

d8 的命令很多，如果有时间，你可以逐一试下。接下来下面我们挑其中一些重点的命令来介绍下，比如`trace-gc`，`trace-opt-verbose`。这些命令涉及到了编译流水线的中间数据，垃圾回收器执行状态等，熟悉使用这些命令可以帮助我们更加深刻理解编译流水线和垃圾回收器的执行状态。

在使用 d8 执行一段代码之前，你需要将你的 JavaScript 源码保存到一个 js 文件中，我们把所需要需要观察的代码都存放到 `test.js` 这个文件中。


打印优化数据

你可以使用`--print-ast`来查看中间生成的 AST，使用`---print-scope`来查看中间生成的作用域，`--print-bytecode`来查看中间生成的字节码。除了这些数据之外，**我们还可以使用 d8 来打印一些优化的数据**，比如下面这样一段代码：

 复制代码

```
1 let a = {x:1}
2 function bar(obj) {
3   return obj.x
4 }
5
6
7 function foo () {
8   let ret = 0
9   for(let i = 1; i < 7049; i++) {
10     ret += bar(a)
11   }
12   return ret
13 }
14
15
16 foo()
```

当 V8 先执行到这段代码的时候，监控到 while 循环会一直被执行，于是判断这是一块热点代码，于是，V8 就会将热点代码编译为优化后的二进制代码，你可以通过下面的命令来查看：

 复制代码

```
1 d8 --trace-opt-verbose test.js
```



执行这段命令之后，提示如下所示：

```
let a = {x:1}
function bar(obj) {
  return obj.x
}

function foo () {
  let ret = 0
  for(let i = 1; i < 7048; i++) {
    ret += bar(a)
  }
  return ret
}

foo()
```

```
1. zsh
eos@libingdeMacBook-Pro 06 % d8 --trace-opt-verbose test.js
[not yet optimizing foo, not enough ticks: 0/2 and ICs changed]
[marking 0x2c730824ff89 <JSFunction foo (sfi = 0x2c730824fe21)> for optimized re
compilation, reason: small function]
eos@libingdeMacBook-Pro 06 %
```

观察上图，我们可以看到终端中出现了一段优化的提示：

复制代码

```
1 <JSFunction foo (sfi = 0x2c730824fe21)> for optimized recompilation, reason: smal
```

这就是告诉我们，已经使用 TurboFan 优化编译器将函数 foo 优化成了二进制代码，执行 foo 时，实际上是执行优化过的二进制代码。

现在我们把 foo 函数中的循环加到 10 万，再来查看优化信息，最终效果如下图所示：

```
let a = {x:1}
function bar(obj) {
  return obj.x
}

function foo () {
  let ret = 0
  for(let i = 1; i < 100000; i++) {
    ret += bar(a)
  }
  return ret
}

foo()
```

```
1. zsh
eos@libingdeMacBook-Pro 06 % d8 --trace-opt-verbose test.js
[not yet optimizing foo, not enough ticks: 0/2 and ICs changed]
[marking 0x2c730824ff89 <JSFunction foo (sfi = 0x2c730824fe21)> for optimized re
compilation, reason: small function]
eos@libingdeMacBook-Pro 06 % d8 --trace-opt-verbose test.js
[not yet optimizing foo, not enough ticks: 0/2 and ICs changed]
[marking 0x0c9c0824ff89 <JSFunction foo (sfi = 0xc9c0824fe21)> for optimized rec
ompilation, reason: small function]
[compiling method 0x0c9c0824ff89 <JSFunction foo (sfi = 0xc9c0824fe21)> using Tu
rboFan OSR]
[optimizing 0x0c9c0824ff89 <JSFunction foo (sfi = 0xc9c0824fe21)> - took 1.472,
3.259, 0.101 ms]
eos@libingdeMacBook-Pro 06 %
```

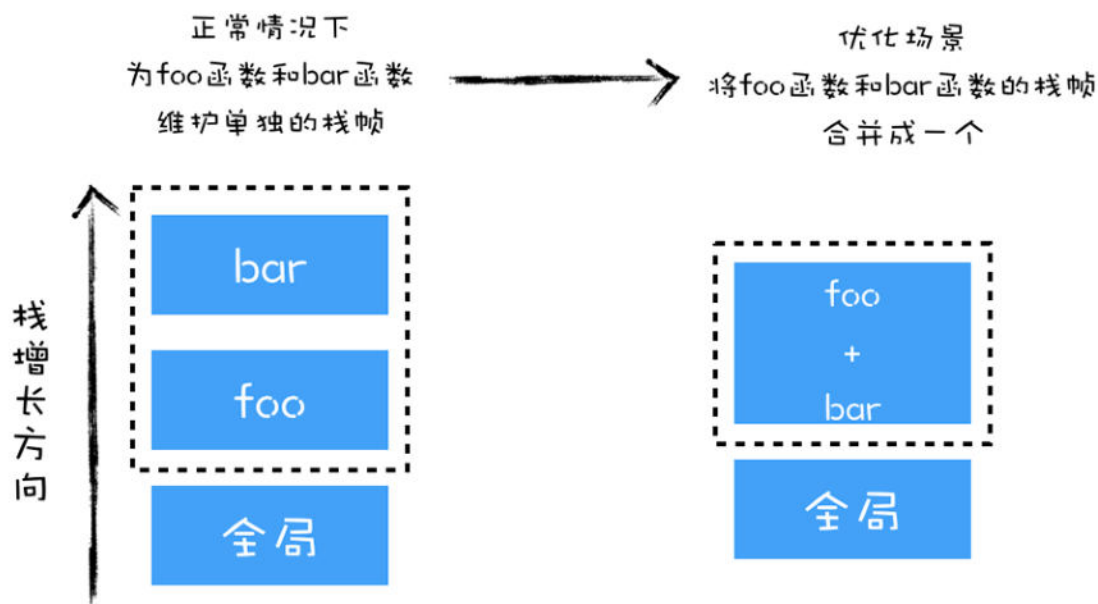
我们看到又出现了一条新的优化信息，新的提示信息如下：

```
1 <JSFunction foo (sfi = 0xc9c0824fe21)> using TurboFan OSR]
```

这段提示是说，由于循环次数过多，V8 采取了 TurboFan 的 OSR 优化，OSR 全称是 **On-Stack Replacement**，它是一种在运行时替换正在运行的函数的栈帧的技术，如果在 foo 函数中，每次调用 bar 函数时，都要创建 bar 函数的栈帧，等 bar 函数执行结束之后，又要销毁 bar 函数的栈帧。

通常情况下，这没有问题，但是在 foo 函数中，采用了大量的循环来重复调用 bar 函数，这就意味着 V8 需要不断为 bar 函数创建栈帧，销毁栈帧，那么这样势必会影响到 foo 函数的执行效率。

于是，V8 采用了 OSR 技术，将 bar 函数和 foo 函数合并成一个新的函数，具体你可以参考下图：



如果我在 foo 函数里面执行了 10 万次循环，在循环体内调用了 10 万次 bar 函数，那么 V8 会实现两次优化，第一次是将 foo 函数编译成优化的二进制代码，第二次是将 foo 函数和 bar 函数合成为一个新的函数。

网上有一篇介绍 OSR 的文章也不错，叫 [on-stack replacement in v8](#)，如果你感兴趣可以查看下。




查看垃圾回收

我们还可以通过 d8 来查看垃圾回收的状态，你可以参看下面这段代码：

 复制代码

```
1 function strToArray(str) {
2   let i = 0
3   const len = str.length
4   let arr = new Uint16Array(str.length)
5   for (; i < len; ++i) {
6     arr[i] = str.charCodeAt(i)
7   }
8   return arr;
9 }
10
11
12 function foo() {
13   let i = 0
14   let str = 'test V8 GC'
15   while (i++ < 1e5) {
16     strToArray(str);
17   }
18 }
19
20
21 foo()
```

上面这段代码，我们重复将一段字符串转换为数组，并重复在堆中申请内存，将转换后的数组存放在内存中。我们可以通过trace-gc来查看这段代码的内存回收状态，执行下面这段命令：

 复制代码

```
1 d8 --trace-gc test.js
```

最终打印出来的结果如下图所示：




```
function strToArray(str) {
  let i = 0
  const len = str.length
  let arr = new Uint16Array(str.length)
  for (; i < len; ++i) {
    arr[i] = str.charCodeAt(i)
  }
  return arr;
}

function foo() {
  let i = 0
  let str = 'test V8 GC'
  while (i++ < 1e5) {
    strToArray(str);
  }
}

foo()
```

```
1. zsh
eos@libingdeMacBook-Pro 06 % d8 --trace-gc test.js
[97447:0x179700000000] 32 ms: Scavenge 1.2 (2.4) -> 0.3 (3.4) MB, 0.9 / 0.0 ms (average mu = 1.000, current mu = 1.000) allocation failure
[97447:0x179700000000] 34 ms: Scavenge 1.2 (3.4) -> 0.3 (3.6) MB, 0.4 / 0.0 ms (average mu = 1.000, current mu = 1.000) allocation failure
[97447:0x179700000000] 36 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 / 0.0 ms (average mu = 1.000, current mu = 1.000) allocation failure
[97447:0x179700000000] 37 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 / 0.0 ms (average mu = 1.000, current mu = 1.000) allocation failure
[97447:0x179700000000] 39 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 / 0.0 ms (average mu = 1.000, current mu = 1.000) allocation failure
[97447:0x179700000000] 40 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 / 0.0 ms (average mu = 1.000, current mu = 1.000) allocation failure
[97447:0x179700000000] 42 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 / 0.0 ms (average mu = 1.000, current mu = 1.000) allocation failure
[97447:0x179700000000] 43 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 / 0.0 ms (average mu = 1.000, current mu = 1.000) allocation failure
[97447:0x179700000000] 45 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 / 0.0 ms (average mu = 1.000, current mu = 1.000) allocation failure
[97447:0x179700000000] 46 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 / 0.0 ms (average mu = 1.000, current mu = 1.000) allocation failure
[97447:0x179700000000] 48 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 / 0.0 ms (average mu = 1.000, current mu = 1.000) allocation failure
[97447:0x179700000000] 50 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.2 / 0.0 ms (average mu = 1.000, current mu = 1.000) allocation failure
[97447:0x179700000000] 51 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.2 / 0.0 ms (average mu = 1.000, current mu = 1.000) allocation failure
[97447:0x179700000000] 53 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 / 0.0 ms (average mu = 1.000, current mu = 1.000) allocation failure
[97447:0x179700000000] 54 ms: Scavenge 0.8 (3.6) -> 0.3 (3.6) MB, 0.2 / 0.0 ms (average mu = 1.000, current mu = 1.000) idle task
eos@libingdeMacBook-Pro 06 %
```

你会看到一堆提示，如下：

复制代码

```
1 Scavenge 1.2 (2.4) -> 0.3 (3.4) MB, 0.9 / 0.0 ms (average mu = 1.000, current mu
```

这句话的意思是提示“Scavenge ... 分配失败”，是因为垃圾回收器 **Scavenge** 所负责的空间已经满了，Scavenge 主要回收 V8 中“**新生代**”中的内存，大多数对象都是分配在新生代内存中，内存分配到新生代中是非常快速的，但是新生代的空间却非常小，通常在 1~8 MB 之间，一旦空间被填满，Scavenge 就会进行“清理”操作。

上面这段代码之所以能频繁触发新生代的垃圾回收，是因为它频繁地去申请内存，而申请内存之后，这块内存就立马变得无效了，为了减少垃圾回收的频率，我们尽量避免申请不必要的内存，比如我们可以换种方式来实现上述代码，如下所示：

复制代码

```
1 function strToArray(str, bufferView) {
2   let i = 0
3   const len = str.length
4   for (; i < len; ++i) {
5     bufferView[i] = str.charCodeAt(i);
6   }
7   return bufferView;
8 }
9 function foo() {
10  let i = 0
11  let str = 'test V8 GC'
12  let buffer = new ArrayBuffer(str.length * 2)
13  let bufferView = new Uint16Array(buffer);
14  while (i++ < 1e5) {
```



```
15     strToArray(str,bufferView);
16   }
17 }
18 foo()
```

我们将 strToArray 中分配的内存块，提前到了 foo 函数中分配，这样我们就不需要每次在 strToArray 函数分配内存了，再次执行trace-gc的命令：

 复制代码

```
1 d8 --trace-gc test.js
```

我们就会看到，这时候没有任何垃圾回收的提示了，这也意味着这时没有任何垃圾分配的操作了。

内部方法

另外，你还可以使用 V8 所提供的一些内部方法，只需要在启动 V8 时传入allow-natives-syntax命令，具体使用方式如下所示：

 复制代码

```
1 d8 --allow-natives-syntax test.js
```

还记得我们在《[03 | 快属性和慢属性：V8 采用了哪些策略提升了对象属性的访问速度？](#)》讲到的快属性和慢属性吗？

我们可以通过内部方法 HasFastProperties 来检查一个对象是否拥有快属性，比如下面这段代码：

 复制代码

```
1 function Foo(property_num,element_num) {
2   //添加可索引属性
3   for (let i = 0; i < element_num; i++) {
4     this[i] = `element${i}`
5   }
6   //添加常规属性
7   for (let i = 0; i < property_num; i++) {
8     let ppt = `property${i}`
```



```
9      this[ppt] = ppt
10    }
11  }
12  var bar = new Foo(10,10)
13  console.log(%HasFastProperties(bar));
14  delete bar.property2
15  console.log(%HasFastProperties(bar)).
```

我们执行下面这个命令：

```
1 d8 test.js --allow-natives-syntax
```

 复制代码

通过传入allow-natives-syntax命令，就能使用HasFastProperties这一类内部接口，默认情况下，我们知道 V8 中的对象都提供了快属性，不过使用了delete bar.property2之后，就没有快属性了，我们可以通过HasFastProperties来判断。

所以可以得出，使用 delete 时候，我们查找属性的速度就会变慢，这也是我们尽量不要使用 delete 的原因。

除了HasFastProperties方法之外，V8 提供的内部方法还有很多，比如你可以使用GetHeapUsage来查看堆的使用状态，可以使用CollectGarbage来主动触发垃圾回收，诸如HaveSameMap、HasDoubleElements等，具体命令细节你可以参考[这里](#)。

总结

好了，今天的内容就介绍到这里，我们来简单总结一下。

d8 是个非常有用的调试工具，能够帮助我们发现我们的代码是否可以被 V8 高效地执行，比如通过 d8 查看代码有没有被 JIT 编译器优化，还可以通过 d8 内置的一些接口查看更多的代码内部信息，而且通过使用 d8，我们会接触各种实际的优化策略，学习这些策略并结合 V8 的工作原理，可以让我们更加接地气地了解 V8 的工作机制。

通过源码来构建 d8 的流程比较简单，首先下载 V8 的编译工具链：depot_tools，然后再利用 depot_tools 下载源码、生成工程、编译工程，这就实现了通过源码编译 d8。这个过程不难，但涉及到了许多工具，在配置过程中可能会遇到一些坑，不过按照流程操作应该能顺利编译出来 d8。



接下来我们重点讨论了如何使用 d8，我们可以通过传入不同的命令，让 d8 来分析 V8 在执行 JavaScript 过程中的一些中间数据。你应该熟练掌握 d8 的使用方式，在后续课程中我们应该还会反复引用本节的一些内容。

思考题

c/c++ 中有内联 (Inline) 函数，和我们文中分析的 OSR 类似，内联函数和 V8 中所采用的 OSR 优化手段类似，都是在执行过程中将两个函数合并成一个，这样在执行代码的过程中，就减少了栈帧切换操作，增加了执行效率，那么今天留给你的思考题是，你认为在什么情况下，V8 会将多个函数合成一个函数？

你可以列举一个实际的例子，并使用 d8 来分析 V8 是否对你的例子进行了优化操作。欢迎你在留言区与我分享讨论。

感谢你的阅读，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

分享给需要的人，Ta 订阅超级会员，你将得 50 元

Ta 单独购买本课程，你将得 20 元

 生成海报并分享

 赞 8  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

[上一篇](#) 07 | 类型转换：V8 是怎么实现 1+“2”的？

[下一篇](#) 09 | 运行时环境：运行 JavaScript 代码的基石



JVM + NIO + Spring

各大厂面试题及知识点详解

限时免费



精选留言 (21)

写留言



h.g. 置顶

2020-05-24

mac brew install v8 就可以直接使用 d8 了

作者回复: 赞

共 5 条评论 >

25



champ可口可乐了

2020-04-17

找到了编译好的d8工具:

mac平台:

<https://storage.googleapis.com/chromium-v8/official/canary/v8-mac64-dbg-8.4.109.zip>

linux32平台:

<https://storage.googleapis.com/chromium-v8/official/canary/v8-linux32-dbg-8.4.109.zip>

linux64平台:

<https://storage.googleapis.com/chromium-v8/official/canary/v8-linux64-dbg-8.4.109.zip>

win32平台:



<https://storage.googleapis.com/chromium-v8/official/canary/v8-win32-dbg-8.4.109.zip>
win64平台:
<https://storage.googleapis.com/chromium-v8/official/canary/v8-win64-dbg-8.4.109.zip>

作者回复: 不错

共 4 条评论 >

👍 19



sugar

2020-04-02

这篇太棒了，终于有这样的手把手带着调试v8的课节了...真的希望这个系列课程中能多加餐一些面向v8底层 c++调试的一些内容，市面上这类资料良莠不齐 且很多资料时效性已经非常差了不具有参考性。几年前，那时我第一次编译chromium，一台顶配的macbookpro生生跑了一下午，然后要是想自己试着改一些地方的代码 反复编译出来看效果就更费劲了。当时一直苦于没人带走了很多弯路，如今看到老师您的这篇专栏喜出望外，真的很希望能多聊些深入的东西，照顾一下各个阶段的学员哈

作者回复: chromium是个庞然大物，下载代码和配置工程和编译代码都是非常费时费力的，调试也是非常麻烦，各种跳转，各种跨进程通信



👍 18



文茵

2020-04-19

经过一番猛折腾 最后发现了jsvu 这个工具才是安装 d8 最简单的方式。<https://github.com/GoogleChromeLabs/jsvu>

作者回复: 赞

共 2 条评论 >

👍 8



踢车牛

2020-04-11

搞了两天，终于在 mac 上把 v8 编译成功了，记录下踩过的坑

1. v8 编译依赖于 xcode, 首先安装相应的 xcode, 然后 `sudo xcode-select -s /Users/videojj/test/Xcode.app`。
2. 代理最好开成全局模式，否则，可能某些地址会访问不到，我通过 fetch v8 是就经常失败，然后全局模式后就成功了。
3. xcode 大约 7G, v8 7.48G, depot_tools 400M, 因此要想编译成功，首先硬盘空间要够。



**子云**

2020-05-05

我有几点疑问呀。

- 一：test.js 里似乎不能出现 require 和 import，d8 直接会报错连语法树都编译不了，。
- 二：垃圾回收那个例子

...

// 本节里的这个例子，我数了数一共打印了 14 次 Scavenge....

```
function strToArray(str) {  
  let i = 0;  
  const len = str.length;  
  let arr = new Uint16Array(str.length);  
  for (; i < len; ++i) {  
    arr[i] = str.charCodeAt(i);  
  }  
  return arr;  
}
```

// 换成下面这个，之打印一次 Scavenge.....，这是为什么？

```
function strToArray(str) {  
  let i = 0;  
  const len = str.length;  
  let arr = str.split(',');  
  return arr;  
}  
...
```

作者回复: d8比较简单，不支持太复杂的功能，最好是单一文件

第二个strtoarray中没有分配新的大数据，也就不会很快沾满新生代，当然就不会频繁触发垃圾回收器了



👍 2

**高亮~**

2020-04-04

glcient sync 命令应该是错的
应该是gclient sync



作者回复: 嗯, 我改下



👍 2



—_—|||

2020-04-16

文中”不过使用了 delete bar.property2 之后, 就没有快属性了”为什么一删就没快属性了, 内存不连续了?

共 4 条评论 >

👍 1



成楠Peter

2020-04-02

思考题

- 1、如文章所说, 需要频繁切换栈帧操作的, 在函数中的for循环操作。
- 2、函数代码行数太少, V8会合并, 减少不必要的开销
- 3、在关键代码(频繁调用)中, 函数频繁调用会增加开销, V8会优化这一部分代码, 合并代码。



👍 1



Geek_1dbd15

2022-01-03

老师, mac中怎么构建和使用v8, 希望出个教程。



👍



大兔叽

2021-11-12

老师, 请问文中提到新生代的容量大概在1-8M, 但是我在查询其他资料的时候, 有提到在32位系统中是18M, 在64位系统中是32M, 所以是我哪里理解有问题吗?



👍



Bleolvlea

2021-01-29

老师, 直接下载您提供的编译好的d8, 将环境变量添加好了后, 可以执行d8 --help命令, 但是别的任何命令如: d8 test.js --print-bytecode, 都会报一个相同的错误: Failed to open startup resource 'snapshot_blob.bin'.

共 2 条评论 >

👍



BlingBling

2020-12-15

老是您好，我mac环境使用源码构建出来的d8，和您的相比似乎少了一些参数选项，比如--print-ast就没有，可能会是什么原因呢？

我V8的版本：V8 version 8.0.426.27

共 2 条评论 >



神仙朱

2020-08-10

为什么 那个例子 会提示scavenge 分配失败，不是说新生代内存满了之后就马上清理吗，这样不就会有内存空闲出来了吗，为什么还会分配失败



Aaaaaaaaaayou

2020-04-22

老师，代码中有 require 引用模块的语句，使用 d8 调试的时候会报 require is not defined，个人理解 require 应该是 nodejs 这个运行环境才能识别。我现在的目标是想看下有模块引入的时候 AST 是什么样子的，请问这个有其他方式吗

作者回复: d8比较简单，不支持太复杂的功能

共 2 条评论 >



踢车牛

2020-04-11

老师，我从 mac 下编译的确实没有 d8 print-ast选项，第一节的 d8 --print-ast test.js 你是如何编译的？

共 4 条评论 >



杨越

2020-04-02

es6开始js开始从基于对象变成了面向对象，因为有了extends关键词，这种说法对吗老师？

共 7 条评论 >



leaf

2020-04-02

在jvm中，针对存在大循环的方法，osr优化是将该方法的栈帧从解释栈帧在线替换成编译栈帧，从而让该方法进入编译执行模式。请问老师确定v8的osr是您文中的意义吗？



一步

2020-04-02

nodejs 有个参数 也是打印有关 v8 的信息的 `node --v8-options` , 只不过这个 能打印的信息 比 d8 少了一部分



Bazinga

2020-04-02

mac 安装了 depot_tools 怎么配置, 直接运行 `glcient sync` 运行不出啊

作者回复: 需要将depot_tools的目录设置到环境变量中

共 5 条评论 >

