

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Linux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，解压缩压缩包，比如你解压到以下这个路径中：

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

然后将需要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，值为0。

```
DEPOT_TOOLS_WIN_TOOLCHAIN = 0
```

这个环境变量的作用是告诉 depnt\_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：

```
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.

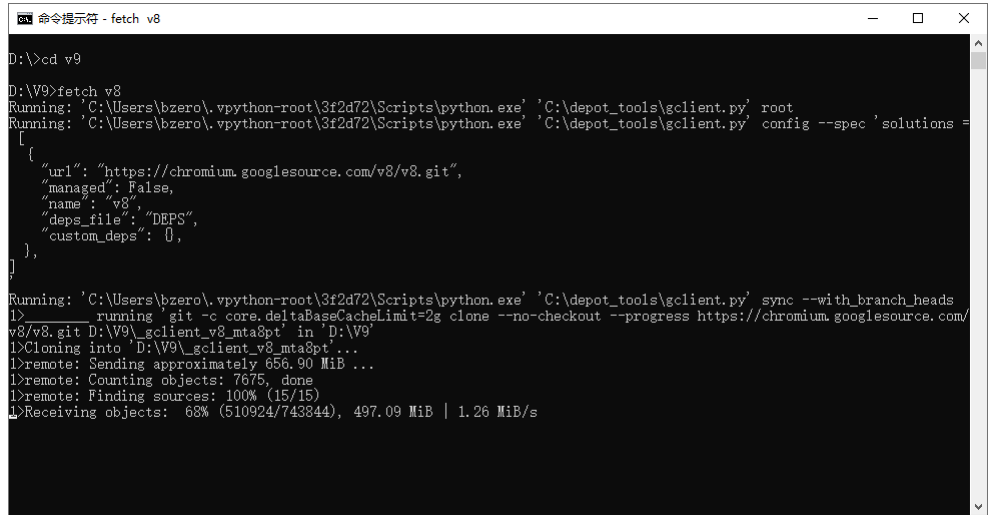
因为编译V8时，使用了这两项所提供的基础开发环境。

### 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：

```
d:
mkdir v8
cd v8
fetch v8
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。



### 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。

```
cd v8
```

```
gn gen out.gn/x64.release --args='is_debug=false target_cpu="x64" v8_target_cpu="arm64" use_goma=true'
```

如果是Mac系统，你可以使用：

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

用它来生成工程。

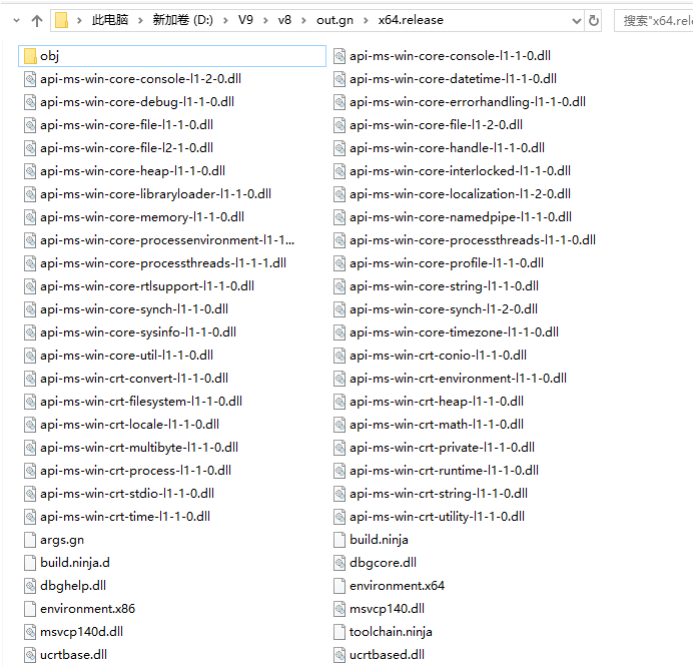
gn是一个跨平台的构建系统，用来构建Ninja工程，Ninja是一个跨平台的编译系统，比如可以通过gn构建Chromium还有V8的工程文件，然后使用Ninja来执行编译，可以使用gn和Ninja来配合使用构建跨平台的工程，这些工程可以在MacOS、Linux、Windows等平台上进行编译。在gn之前，Google使用了gyp来构建，由于gn的效率更高，所以现在都在使用gn。

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：



编译d8

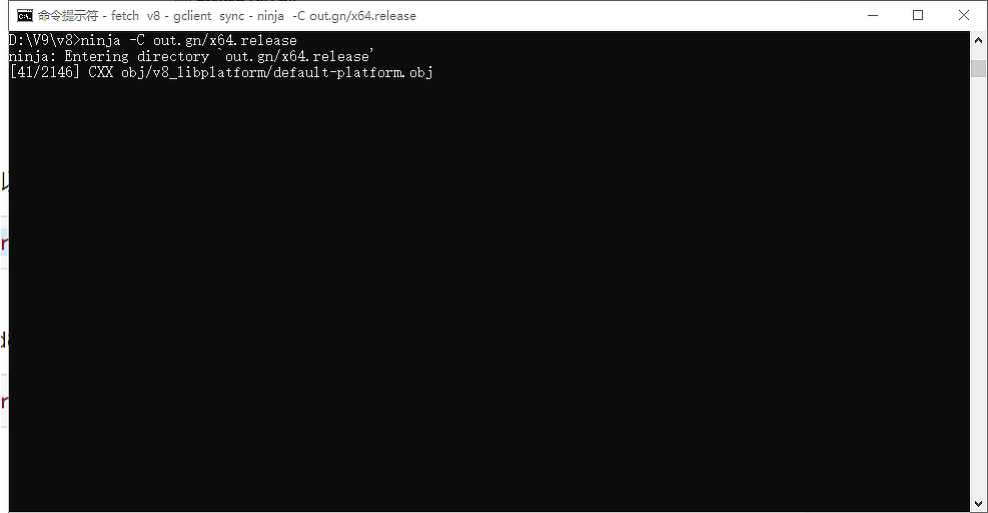
生成了d8的工程配置文件，接下来就可以编译d8了，你可以使用下面的命令：

```
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：

```
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：



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：

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
cctest.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：

```
d8 --help
```

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很多行，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：

```
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：

```
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：

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

function foo () {
  let ret = 0
  for(let i = 1; i < 7049; i++) {
    ret += bar(a)
  }
  return ret
}

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：

```
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：

```
<JSFunction foo (sfi = 0x2c730824fe21)> for optimized recompilation, reason: small function]
```

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 %
```

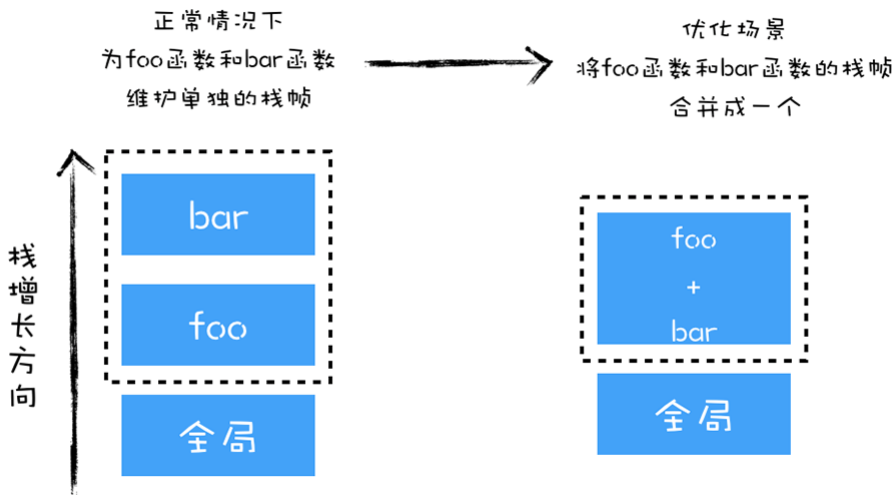
我们看到又出现了一条新的优化信息，新的提示信息如下：

```
<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：

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

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

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：

```
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 /
[97447:0x179700000000] 34 ms: Scavenge 1.2 (3.4) -> 0.3 (3.6) MB, 0.4 /
[97447:0x179700000000] 36 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 37 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 39 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 40 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 42 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 43 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 45 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 46 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 48 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 50 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.2 /
[97447:0x179700000000] 51 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.2 /
[97447:0x179700000000] 53 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 54 ms: Scavenge 0.8 (3.6) -> 0.3 (3.6) MB, 0.2 /
eos@libingdeMacBook-Pro 06 %
```

你会看到一堆提示，如下：

```
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
```

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：

```
function strToArray(str, bufferView) {
  let i = 0
  const len = str.length
  for (; i < len; ++i) {
    bufferView[i] = str.charCodeAt(i);
  }
  return bufferView;
}
function foo() {
  let i = 0
  let str = 'test V8 GC'
  let buffer = new ArrayBuffer(str.length * 2)
  let bufferView = new Uint16Array(buffer);
  while (i++ < 1e5) {
    strToArray(str,bufferView);
  }
}
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：

```
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：

```
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：

```
function Foo(property_num,element_num) {
  //添加可索引属性
  for (let i = 0; i < element_num; i++) {
    this[i] = `element${i}`
  }
  //添加常规属性
  for (let i = 0; i < property_num; i++) {
    let ppt = `property${i}`
    this[ppt] = ppt
  }
}
var bar = new Foo(10,10)
console.log(%HasFastProperties(bar));
delete bar.property2
console.log(%HasFastProperties(bar));
```

我们执行下面这个命令：

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

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。

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Linux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，解压缩包，比如你解压到以下这个路径中：

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

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gcclient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。

```
DEPOT_TOOLS_WIN_TOOLCHAIN = 0
```

这个环境变量的作用是告诉 depot\_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：

```
gcclient 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：

```
d:
mkdir v8
cd v8
fetch v8
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
I> 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'
I>Cloning into 'D:\V9_gclient_v8_mta8pt'...
I>remote: Sending approximately 656.90 MiB ...
I>remote: Counting objects: 7675, done
I>remote: Finding sources: 100% (15/15)
I>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。

```
cd v8

gn gen out.gn/x64.release --args='is_debug=false target_cpu="x64" v8_target_cpu="arm64" use_goma=true'
```

如果是Mac系统，你可以使用：

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

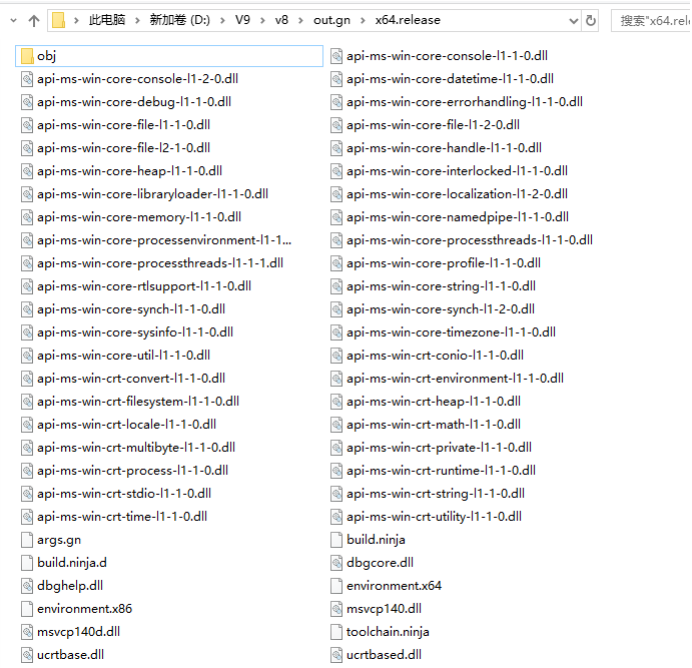
用它来生成工程。

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：



编译d8

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：

```
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：

```
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：



```
命令提示符 - fetch v8 - gclient sync - ninja -C out.gn/x64.release
D:\V9\v8>ninja -C out.gn/x64.release
ninja: Entering directory `out.gn/x64.release'
[41/2146] CXX obj/v8_libplatform/default-platform.obj
```

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：

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
cctest.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：

```
d8 --help
```

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很多行，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：

```
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：

```
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 and 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：

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

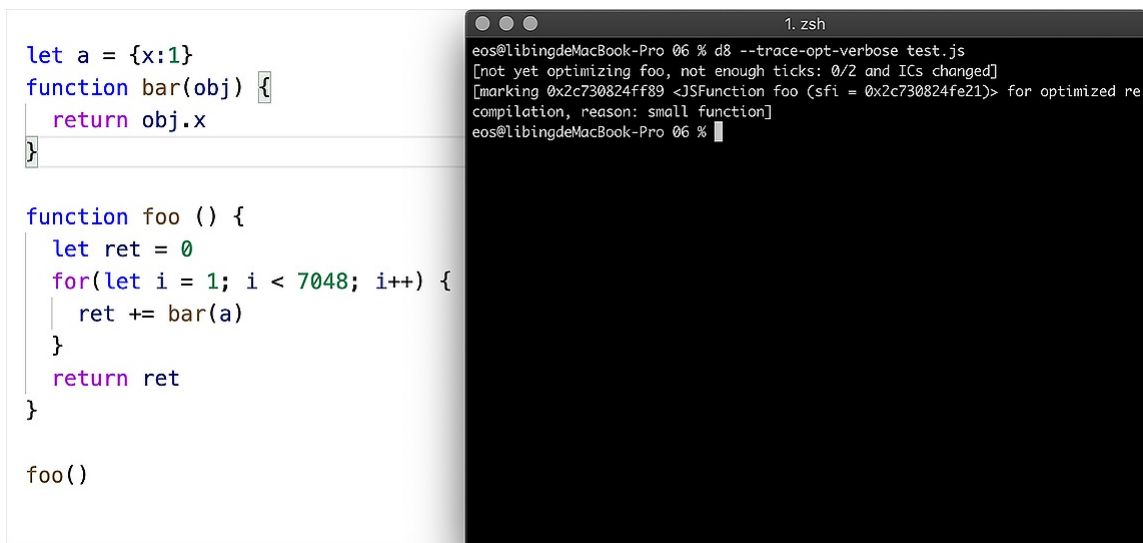
function foo () {
  let ret = 0
  for(let i = 1; i < 7049; i++) {
    ret += bar(a)
  }
  return ret
}

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：

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

执行这段命令之后，提示如下所示：

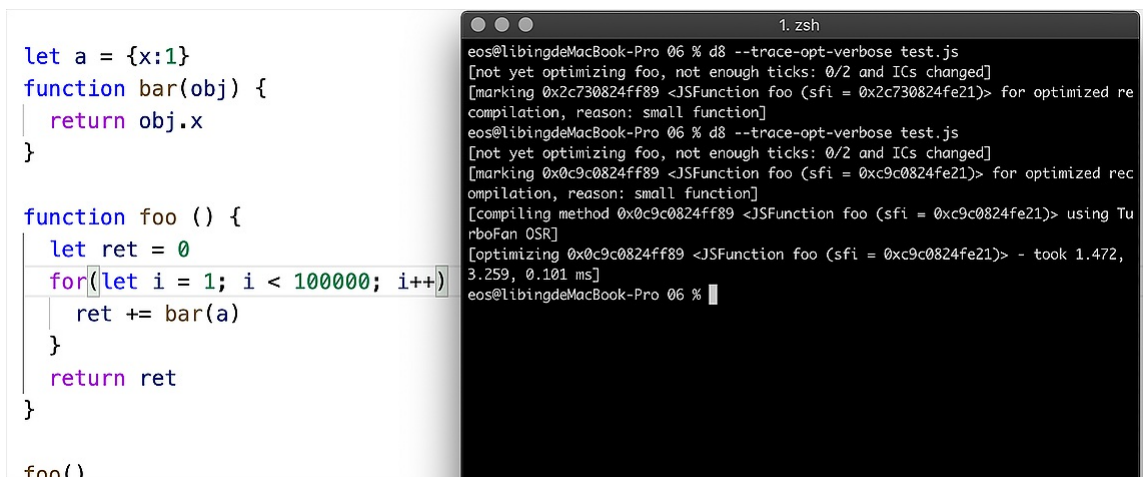


观察上图，我们可以看到终端中出现了一段优化的提示：

```
<JSFunction foo (sfi = 0x2c730824fe21)> for optimized recompilation, reason: small function]
```

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：



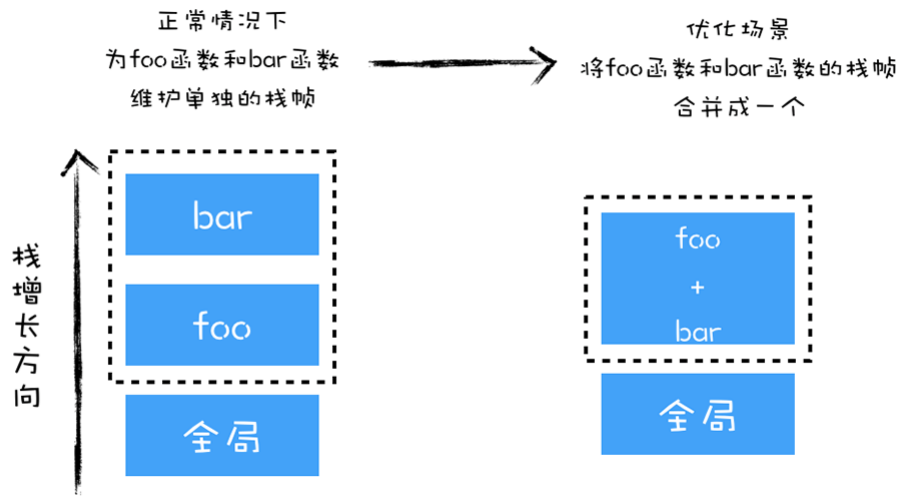
我们看到又出现了一条新的优化信息，新的提示信息如下：

```
<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：

```
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()
```

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：

```
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 /
[97447:0x179700000000] 34 ms: Scavenge 1.2 (3.4) -> 0.3 (3.6) MB, 0.4 /
[97447:0x179700000000] 36 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 37 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 39 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 40 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 42 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 43 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 45 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 46 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 48 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 50 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.2 /
[97447:0x179700000000] 51 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.2 /
[97447:0x179700000000] 53 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 54 ms: Scavenge 0.8 (3.6) -> 0.3 (3.6) MB, 0.2 /
eos@libingdeMacBook-Pro 06 %
```

你会看到一堆提示，如下：

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

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：

```
function strToArray(str, bufferView) {
  let i = 0
  const len = str.length
  for (; i < len; ++i) {
    bufferView[i] = str.charCodeAt(i);
  }
  return bufferView;
}
function foo() {
  let i = 0
  let str = 'test V8 GC'
  let buffer = new ArrayBuffer(str.length * 2)
  let bufferView = new Uint16Array(buffer);
  while (i++ < 1e5) {
    strToArray(str,bufferView);
  }
}
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：

```
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：

```
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：

```
function Foo(property_num,element_num) {
  //添加可索引属性
  for (let i = 0; i < element_num; i++) {
    this[i] = `element${i}`
  }
  //添加常规属性
  for (let i = 0; i < property_num; i++) {
    let ppt = `property${i}`
    this[ppt] = ppt
  }
}
var bar = new Foo(10,10)
console.log(%HasFastProperties(bar));
delete bar.property2
console.log(%HasFastProperties(bar));
```

我们执行下面这个命令：

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

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。

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Linux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，解压缩压缩包，比如你解压到以下这个路径中：

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

然后需要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。

```
DEPOT_TOOLS_WIN_TOOLCHAIN = 0
```

这个环境变量的作用是告诉 depot\_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：

```
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：

```
d:
mkdir v8
cd v8
fetch v8
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。

## 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。

```
cd v8
```

```
gn gen out.gn/x64.release --args='is_debug=false target_cpu="x64" v8_target_cpu="arm64" use_goma=true'
```

如果是Mac系统，你可以使用：

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

用它来生成工程。

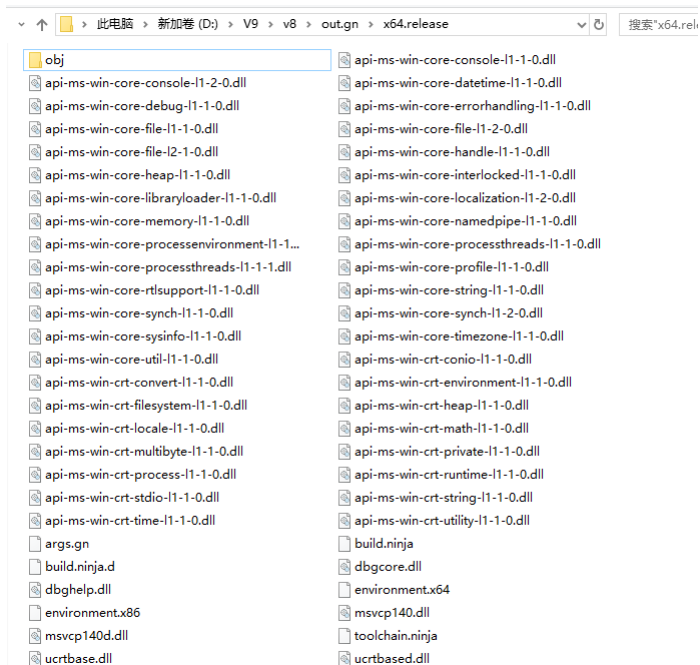
gn是一个跨平台的构建系统，用来构建Ninja工程，Ninja是一个跨平台的编译系统，比如可以通过gn构建Chromium还有V8的工程文件，然后使用Ninja来执行编译，可以使用gn和Ninja来配合使用构建跨平台的工程，这些工程可以在MacOS、Linux、Windows等平台上进行编译。在gn之前，Google使用了gyp来构建，由于gn的效率更高，所以现在都在使用gn。

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：





## 编译d8

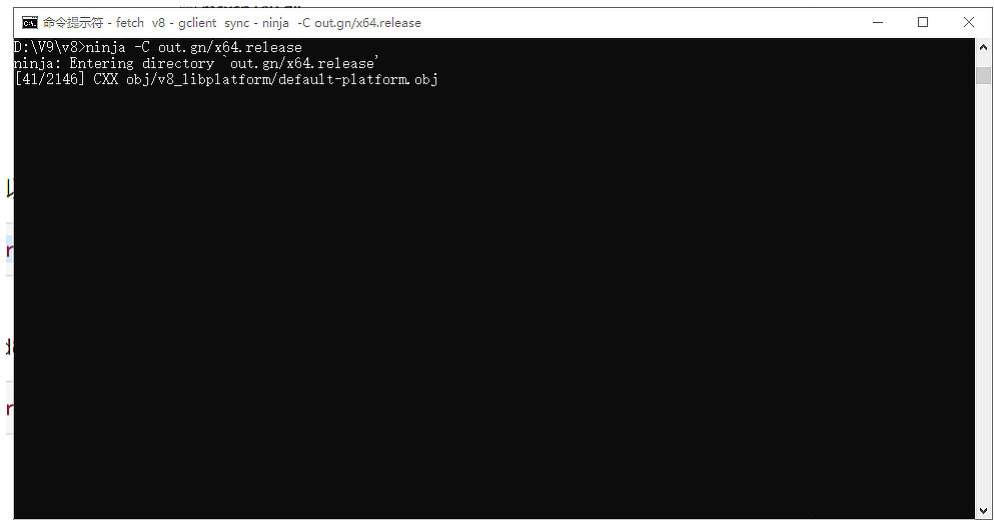
生成了d8的工程配置文件，接下来就可以编译d8了，你可以使用下面的命令：

```
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：

```
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：



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：

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
cctest.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：

```
d8 --help
```

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很多行，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：

```
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：

```
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：

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

function foo () {
  let ret = 0
  for(let i = 1; i < 7049; i++) {
    ret += bar(a)
  }
  return ret
}

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：

```
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：

```
<JSFunction foo (sfi = 0x2c730824fe21)> for optimized recompilation, reason: small function]
```

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 %
```

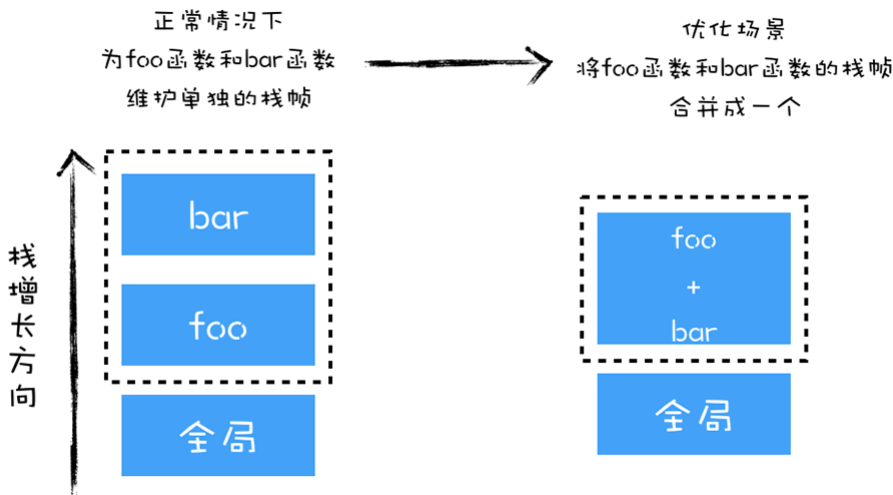
我们看到又出现了一条新的优化信息，新的提示信息如下：

```
<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：

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

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

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：

```
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 /
[97447:0x179700000000] 34 ms: Scavenge 1.2 (3.4) -> 0.3 (3.6) MB, 0.4 /
[97447:0x179700000000] 36 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 37 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 39 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 40 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 42 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 43 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 45 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 46 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 48 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 50 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.2 /
[97447:0x179700000000] 51 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.2 /
[97447:0x179700000000] 53 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 54 ms: Scavenge 0.8 (3.6) -> 0.3 (3.6) MB, 0.2 /
eos@libingdeMacBook-Pro 06 %
```

你会看到一堆提示，如下：

```
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
```

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：

```
function strToArray(str, bufferView) {
  let i = 0
  const len = str.length
  for (; i < len; ++i) {
    bufferView[i] = str.charCodeAt(i);
  }
  return bufferView;
}
function foo() {
  let i = 0
  let str = 'test V8 GC'
  let buffer = new ArrayBuffer(str.length * 2)
  let bufferView = new Uint16Array(buffer);
  while (i++ < 1e5) {
    strToArray(str,bufferView);
  }
}
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：

```
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：

```
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：

```
function Foo(property_num,element_num) {
  //添加可索引属性
  for (let i = 0; i < element_num; i++) {
    this[i] = `element${i}`
  }
  //添加常规属性
  for (let i = 0; i < property_num; i++) {
    let ppt = `property${i}`
    this[ppt] = ppt
  }
}
var bar = new Foo(10,10)
console.log(%HasFastProperties(bar));
delete bar.property2
console.log(%HasFastProperties(bar));
```

我们执行下面这个命令：

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

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。

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Linux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，解压缩压缩包，比如你解压到以下这个路径中：

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

然后需要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。

```
DEPOT_TOOLS_WIN_TOOLCHAIN = 0
```

这个环境变量的作用是告诉 depot\_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：

```
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：

```
d:
mkdir v8
cd v8
fetch v8
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
I> 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
I>Cloning into 'D:\V9\_gclient_v8_mta8pt'...
I>remote: Sending approximately 656.90 MiB ...
I>remote: Counting objects: 7675, done
I>remote: Finding sources: 100% (15/15)
I>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。

```
cd v8

gn gen out.gn/x64.release --args='is_debug=false target_cpu="x64" v8_target_cpu="arm64" use_goma=true'
```

如果是Mac系统，你可以使用：

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

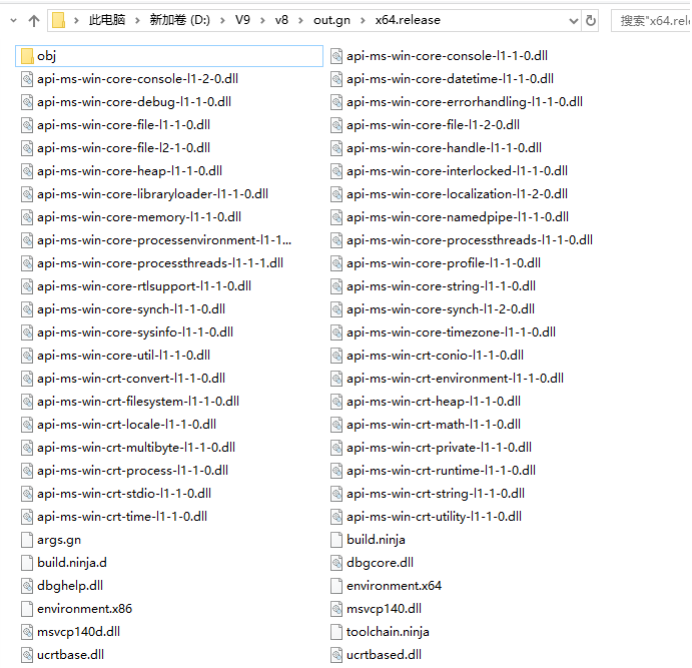
用它来生成工程。

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：



编译d8

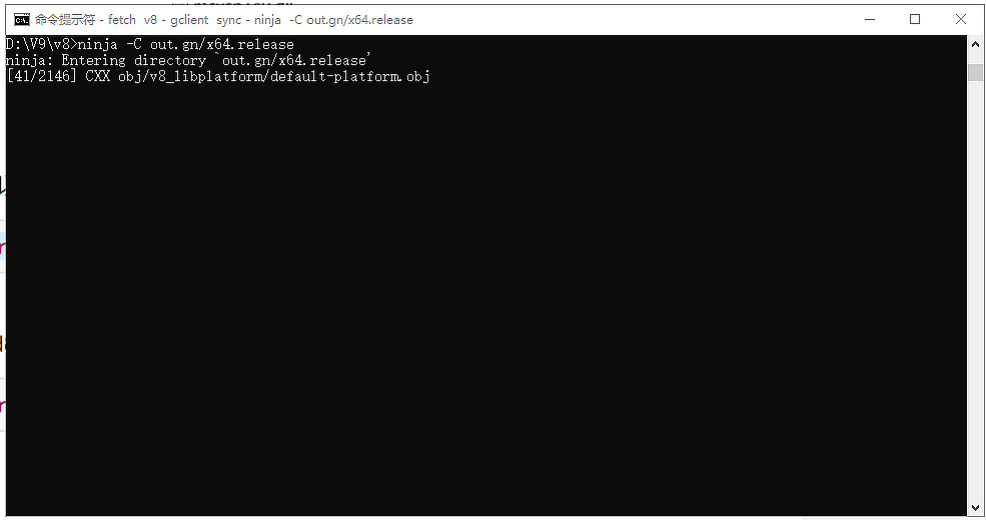
生成了d8的工程配置文件，接下来就可以编译d8了，你可以使用下面的命令：

```
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：

```
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：



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：

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
cctest.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：

```
d8 --help
```

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很多行，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：

```
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：

```
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 and 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：

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

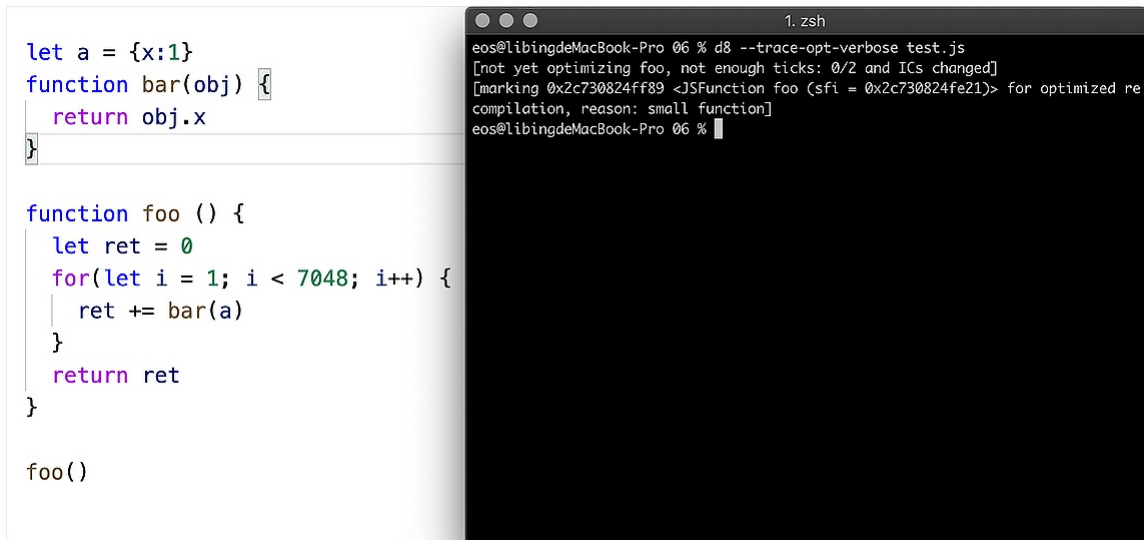
function foo () {
  let ret = 0
  for(let i = 1; i < 7049; i++) {
    ret += bar(a)
  }
  return ret
}

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：

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

执行这段命令之后，提示如下所示：

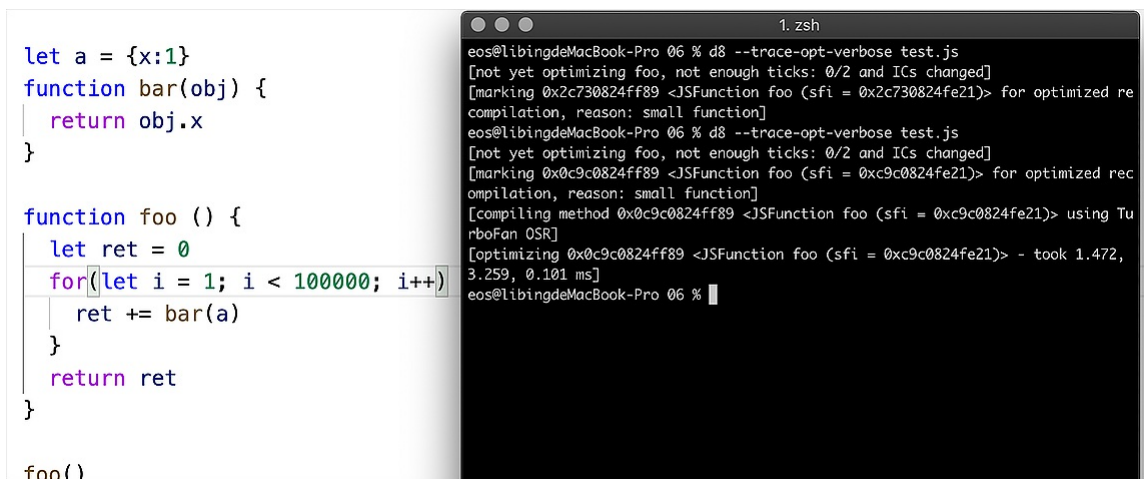


观察上图，我们可以看到终端中出现了一段优化的提示：

```
<JSFunction foo (sfi = 0x2c730824fe21)> for optimized recompilation, reason: small function]
```

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：



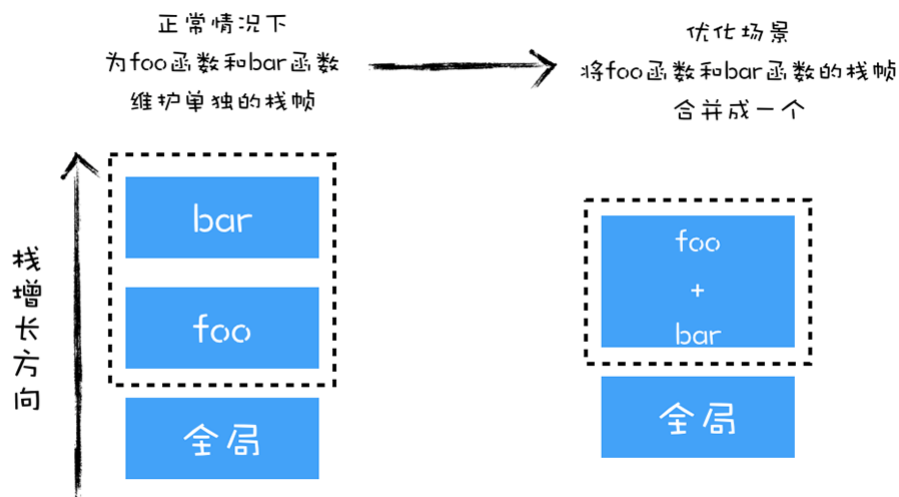
我们看到又出现了一条新的优化信息，新的提示信息如下：

```
<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：

```
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()

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：

```
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 /
[97447:0x179700000000] 34 ms: Scavenge 1.2 (3.4) -> 0.3 (3.6) MB, 0.4 /
[97447:0x179700000000] 36 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 37 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 39 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 40 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 42 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 43 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 45 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 46 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 48 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 50 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.2 /
[97447:0x179700000000] 51 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.2 /
[97447:0x179700000000] 53 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 54 ms: Scavenge 0.8 (3.6) -> 0.3 (3.6) MB, 0.2 /
eos@libingdeMacBook-Pro 06 %
```

你会看到一堆提示，如下：

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

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：

```
function strToArray(str, bufferView) {
  let i = 0
  const len = str.length
  for (; i < len; ++i) {
    bufferView[i] = str.charCodeAt(i);
  }
  return bufferView;
}
function foo() {
  let i = 0
  let str = 'test V8 GC'
  let buffer = new ArrayBuffer(str.length * 2)
  let bufferView = new Uint16Array(buffer);
  while (i++ < 1e5) {
    strToArray(str,bufferView);
  }
}
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：

```
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：

```
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：

```
function Foo(property_num,element_num) {
  //添加可索引属性
  for (let i = 0; i < element_num; i++) {
    this[i] = `element${i}`
  }
  //添加常规属性
  for (let i = 0; i < property_num; i++) {
    let ppt = `property${i}`
    this[ppt] = ppt
  }
}
var bar = new Foo(10,10)
console.log(%HasFastProperties(bar));
delete bar.property2
console.log(%HasFastProperties(bar));
```

我们执行下面这个命令：

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

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。

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Linux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，解压缩压缩包，比如你解压到以下这个路径中：

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

然后需要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，值为 0。

```
DEPOT_TOOLS_WIN_TOOLCHAIN = 0
```

这个环境变量的作用是告诉 depot\_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：

```
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。

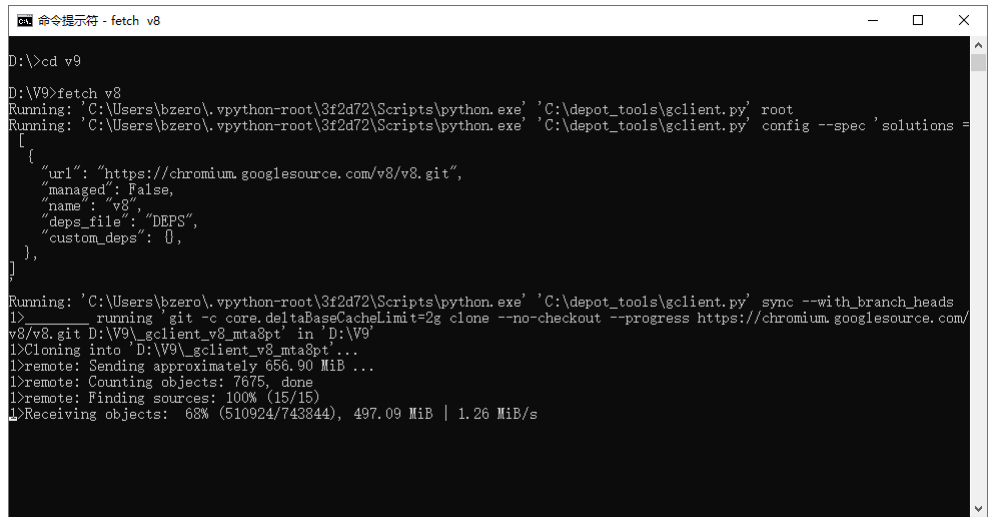
因为编译V8时，使用了这两项所提供的基础开发环境。

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：

```
d:
mkdir v8
cd v8
fetch v8
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。



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。

```
cd v8
```

```
gn gen out.gn/x64.release --args='is_debug=false target_cpu="x64" v8_target_cpu="arm64" use_goma=true'
```

如果是Mac系统，你可以使用：

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

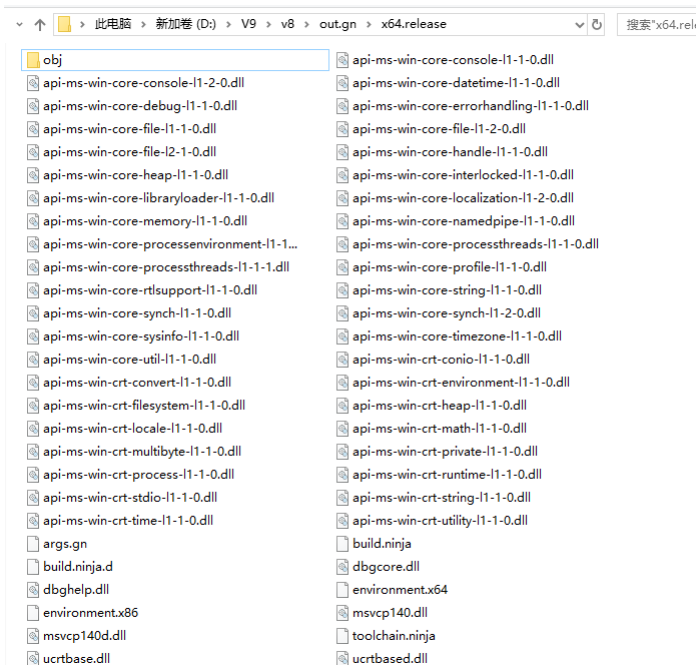
用它来生成工程。

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：



## 编译d8

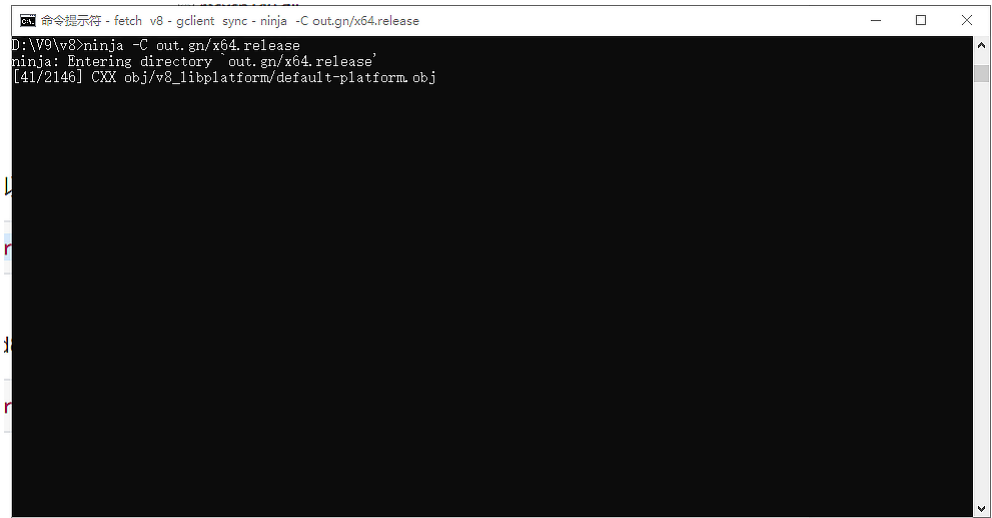
生成了d8的工程配置文件，接下来就可以编译d8了，你可以使用下面的命令：

```
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：

```
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：



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：

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
cctest.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：

```
d8 --help
```

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很多行，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：

```
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：

```
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：

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

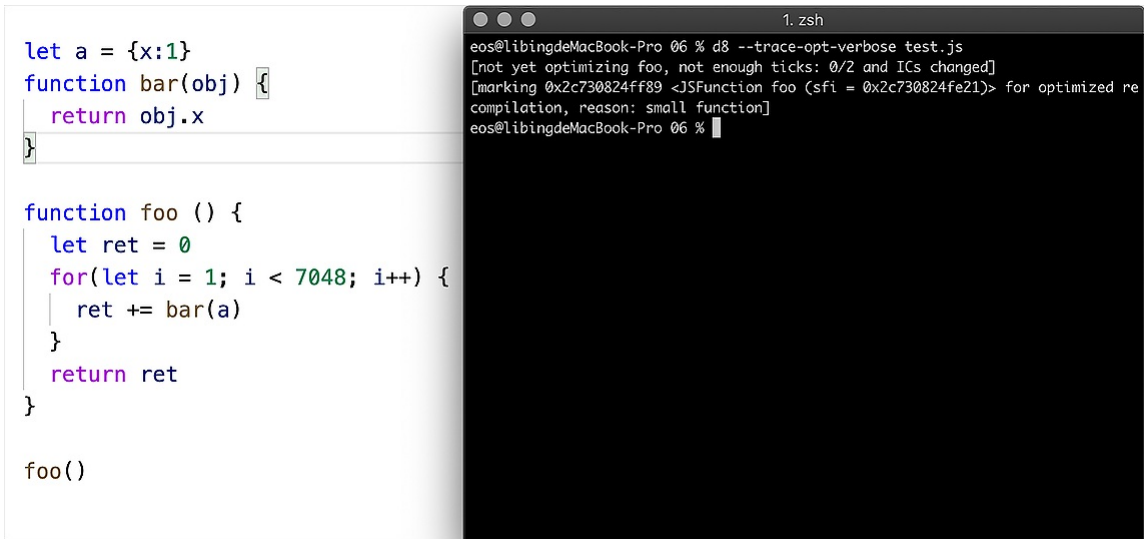
function foo () {
  let ret = 0
  for(let i = 1; i < 7049; i++) {
    ret += bar(a)
  }
  return ret
}

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：

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

执行这段命令之后，提示如下所示：



观察上图，我们可以看到终端中出现了一段优化的提示：

```
<JSFunction foo (sfi = 0x2c730824fe21)> for optimized recompilation, reason: small function]
```

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 %
```

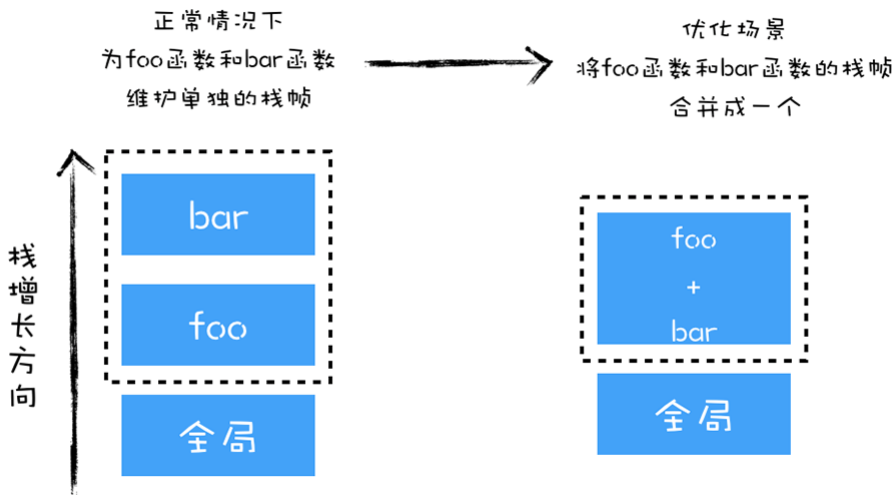
我们看到又出现了一条新的优化信息，新的提示信息如下：

```
<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：

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

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

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：

```
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 /
[97447:0x179700000000] 34 ms: Scavenge 1.2 (3.4) -> 0.3 (3.6) MB, 0.4 /
[97447:0x179700000000] 36 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 37 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 39 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 40 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 42 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 43 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 45 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 46 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 48 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 50 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.2 /
[97447:0x179700000000] 51 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.2 /
[97447:0x179700000000] 53 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 54 ms: Scavenge 0.8 (3.6) -> 0.3 (3.6) MB, 0.2 /
eos@libingdeMacBook-Pro 06 %
```

你会看到一堆提示，如下：

```
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
```

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：

```
function strToArray(str, bufferView) {
  let i = 0
  const len = str.length
  for (; i < len; ++i) {
    bufferView[i] = str.charCodeAt(i);
  }
  return bufferView;
}
function foo() {
  let i = 0
  let str = 'test V8 GC'
  let buffer = new ArrayBuffer(str.length * 2)
  let bufferView = new Uint16Array(buffer);
  while (i++ < 1e5) {
    strToArray(str,bufferView);
  }
}
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：

```
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：

```
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：

```
function Foo(property_num,element_num) {
  //添加可索引属性
  for (let i = 0; i < element_num; i++) {
    this[i] = `element${i}`
  }
  //添加常规属性
  for (let i = 0; i < property_num; i++) {
    let ppt = `property${i}`
    this[ppt] = ppt
  }
}
var bar = new Foo(10,10)
console.log(%HasFastProperties(bar));
delete bar.property2
console.log(%HasFastProperties(bar));
```

我们执行下面这个命令：

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

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。

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Linux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，解压缩压缩包，比如你解压到以下这个路径中：

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

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gcclient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。

```
DEPOT_TOOLS_WIN_TOOLCHAIN = 0
```

这个环境变量的作用是告诉 depot\_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：

```
gcclient 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：

```
d:
mkdir v8
cd v8
fetch v8
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
I> 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'
I>Cloning into 'D:\V9_gclient_v8_mta8pt'...
I>remote: Sending approximately 656.90 MiB ...
I>remote: Counting objects: 7675, done
I>remote: Finding sources: 100% (15/15)
I>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。

```
cd v8

gn gen out.gn/x64.release --args='is_debug=false target_cpu="x64" v8_target_cpu="arm64" use_goma=true'
```

如果是Mac系统，你可以使用：

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

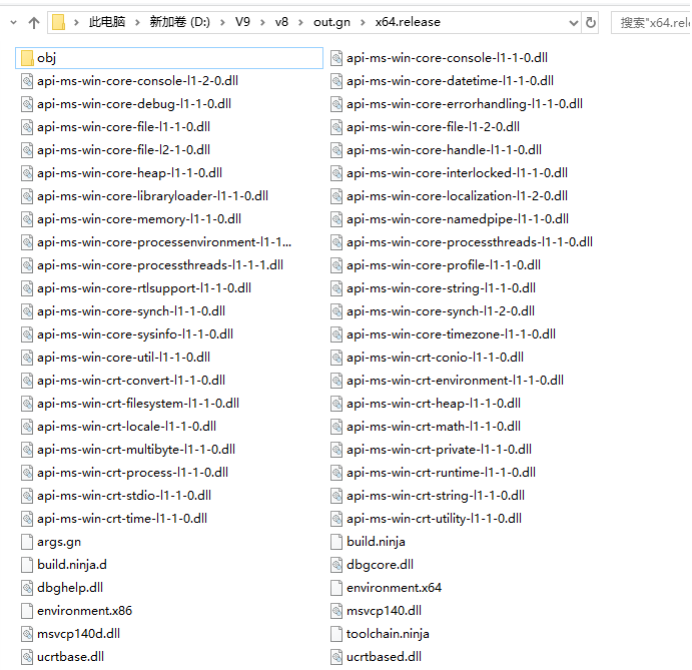
用它来生成工程。

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：



编译d8

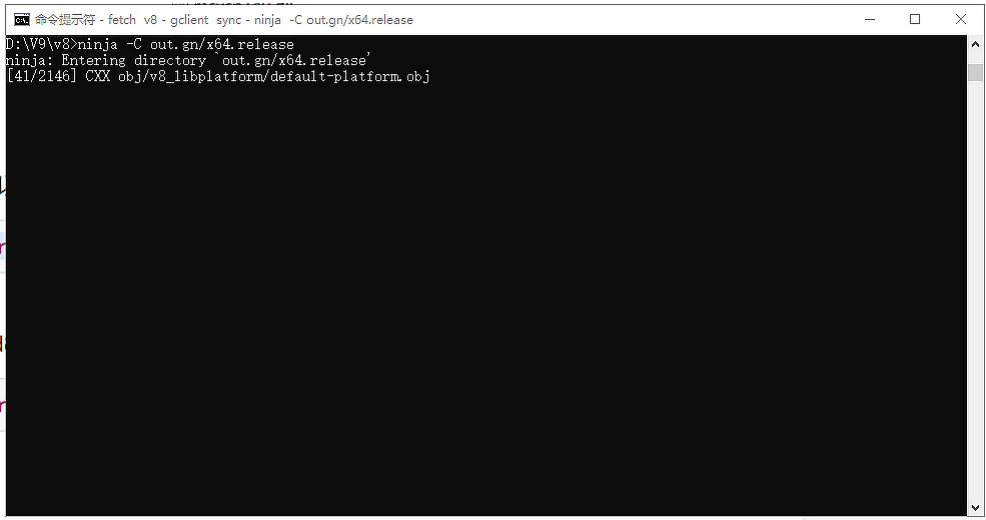
生成了d8的工程配置文件，接下来就可以编译d8了，你可以使用下面的命令：

```
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：

```
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：



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：

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
cctest.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：

```
d8 --help
```

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很多行，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：

```
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：

```
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 and 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：

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

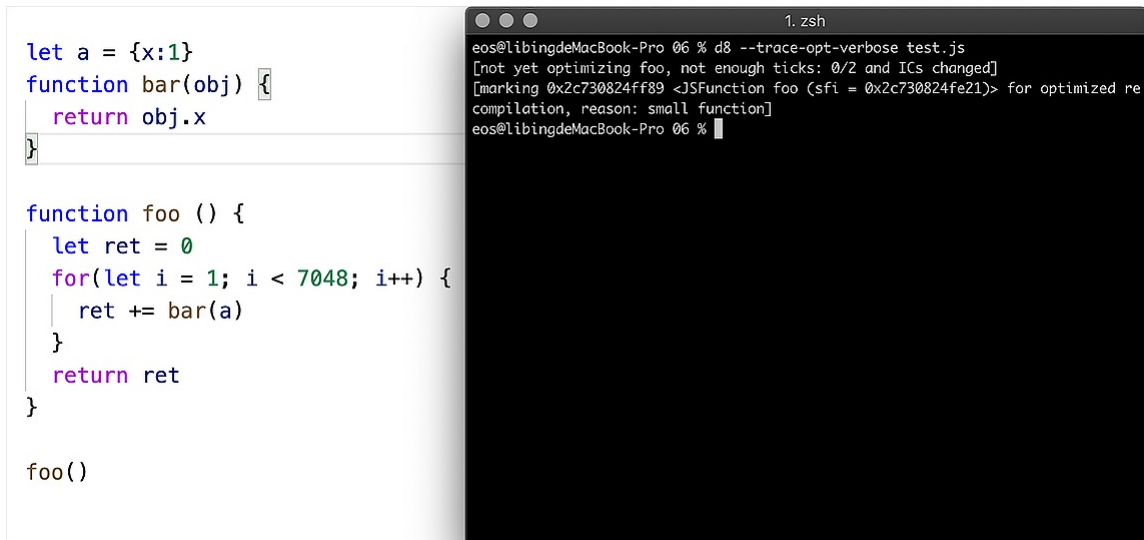
function foo () {
  let ret = 0
  for(let i = 1; i < 7049; i++) {
    ret += bar(a)
  }
  return ret
}

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：

```
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()

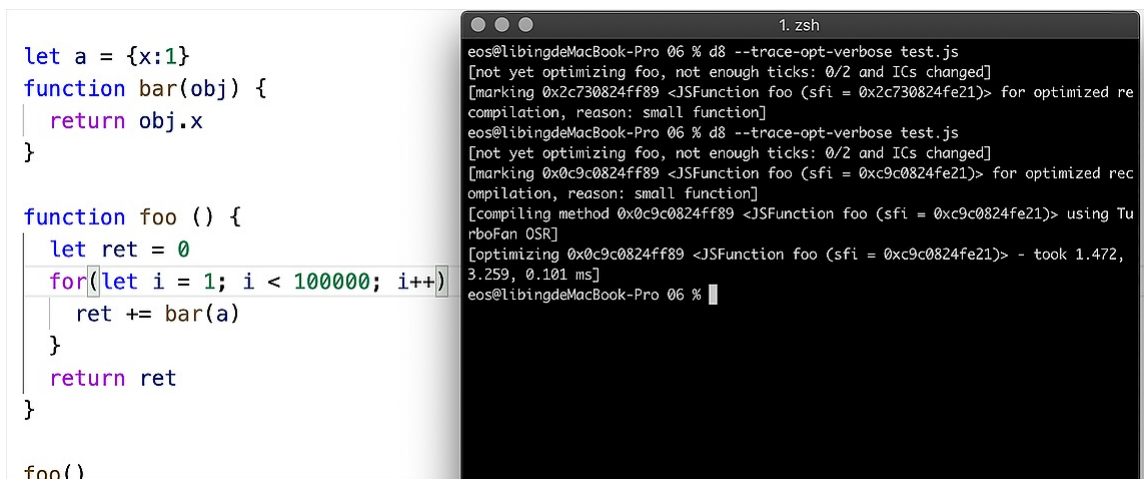
eos@libingdeMacBook-Pro 06 % d8 --trace-opt-verbose test.js
[not yet optimizing foo, not enough ticks: 0/2 and ICs changed]
[marking 0x2c730824ff89 <JSFunction foo (sfi = 0x2c730824fe21)> for optimized re
compilation, reason: small function]
eos@libingdeMacBook-Pro 06 %
```

观察上图，我们可以看到终端中出现了一段优化的提示：

```
<JSFunction foo (sfi = 0x2c730824fe21)> for optimized recompilation, reason: small function]
```

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()

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 %
```

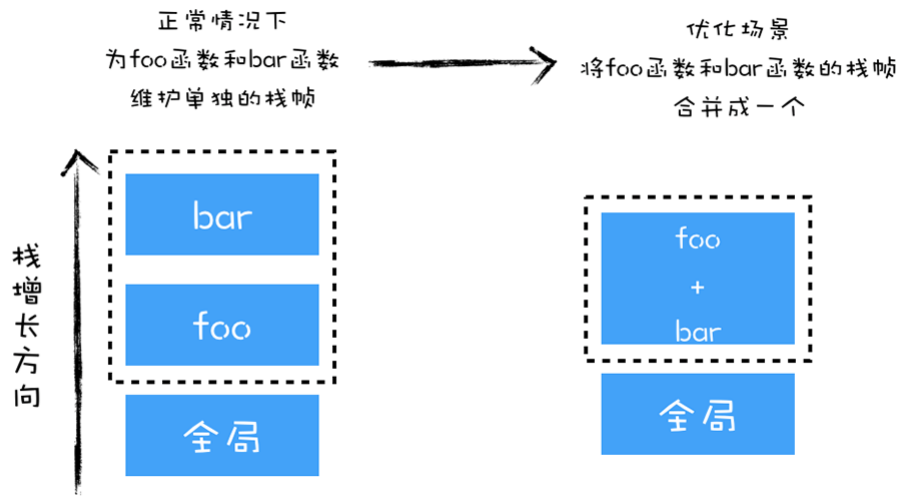
我们看到又出现了一条新的优化信息，新的提示信息如下：

```
<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：

```
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()
```

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：

```
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 /  
[97447:0x179700000000] 34 ms: Scavenge 1.2 (3.4) -> 0.3 (3.6) MB, 0.4 /  
[97447:0x179700000000] 36 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /  
[97447:0x179700000000] 37 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /  
[97447:0x179700000000] 39 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /  
[97447:0x179700000000] 40 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /  
[97447:0x179700000000] 42 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /  
[97447:0x179700000000] 43 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /  
[97447:0x179700000000] 45 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /  
[97447:0x179700000000] 46 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /  
[97447:0x179700000000] 48 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /  
[97447:0x179700000000] 50 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.2 /  
[97447:0x179700000000] 51 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.2 /  
[97447:0x179700000000] 53 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /  
[97447:0x179700000000] 54 ms: Scavenge 0.8 (3.6) -> 0.3 (3.6) MB, 0.2 /  
eos@libingdeMacBook-Pro 06 %
```

你会看到一堆提示，如下：



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

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，比如我们可以用换种方式来实现上述代码，如下所示：

```
function strToArray(str, bufferView) {
  let i = 0
  const len = str.length
  for (; i < len; ++i) {
    bufferView[i] = str.charCodeAt(i);
  }
  return bufferView;
}
function foo() {
  let i = 0
  let str = 'test V8 GC'
  let buffer = new ArrayBuffer(str.length * 2)
  let bufferView = new Uint16Array(buffer);
  while (i++ < 1e5) {
    strToArray(str,bufferView);
  }
}
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：

```
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：

```
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：

```
function Foo(property_num,element_num) {
  //添加可索引属性
  for (let i = 0; i < element_num; i++) {
    this[i] = `element${i}`
  }
  //添加常规属性
  for (let i = 0; i < property_num; i++) {
    let ppt = `property${i}`
    this[ppt] = ppt
  }
}
var bar = new Foo(10,10)
console.log(%HasFastProperties(bar));
delete bar.property2
console.log(%HasFastProperties(bar));
```

我们执行下面这个命令：

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

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。

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Linux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，解压缩压缩包，比如你解压到以下这个路径中：

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

然后需要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。

```
DEPOT_TOOLS_WIN_TOOLCHAIN = 0
```

这个环境变量的作用是告诉 depot\_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：

```
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。

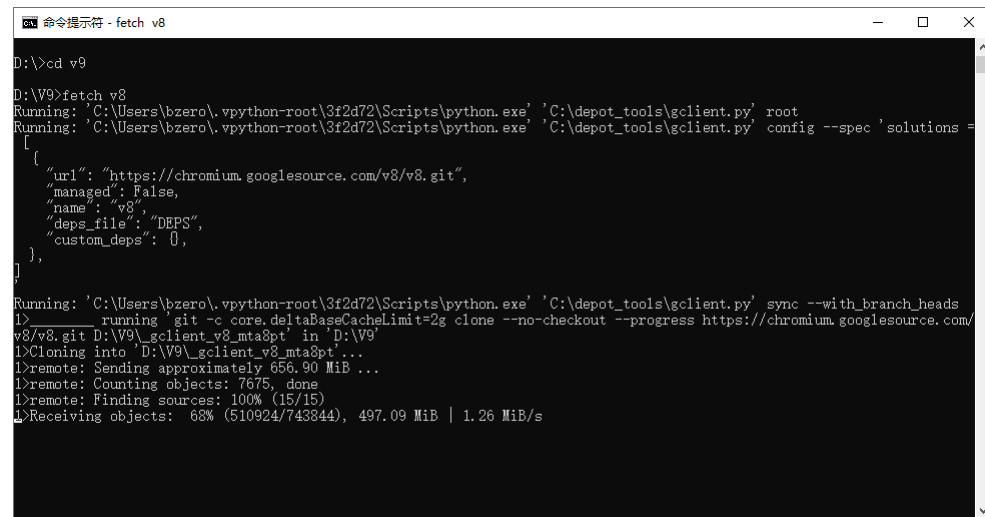
因为编译V8时，使用了这两项所提供的基础开发环境。

## 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：

```
d:
mkdir v8
cd v8
fetch v8
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
I> running 'git -c core.deltaBaseCacheLimit=2g clone --no-checkout --progress https://chromium.googlesource.com/
v8/v8.git D:\V9\gclient_v8_mta3pt' in 'D:\V9'
I>Cloning into 'D:\V9\gclient_v8_mta3pt'...
I>remote: Sending approximately 656.00 MiB ...
I>remote: Counting objects: 7675, done
I>remote: Finding sources: 100% (15/15)
I>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。

```
cd v8
```

```
gn gen out.gn/x64.release --args='is_debug=false target_cpu="x64" v8_target_cpu="arm64" use_goma=true'
```

如果是Mac系统，你可以使用：

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

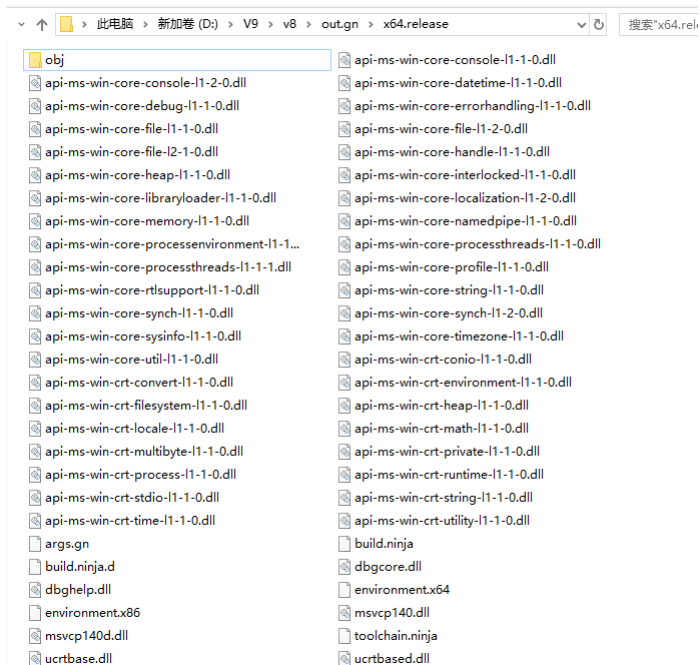
用它来生成工程。

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：



## 编译d8

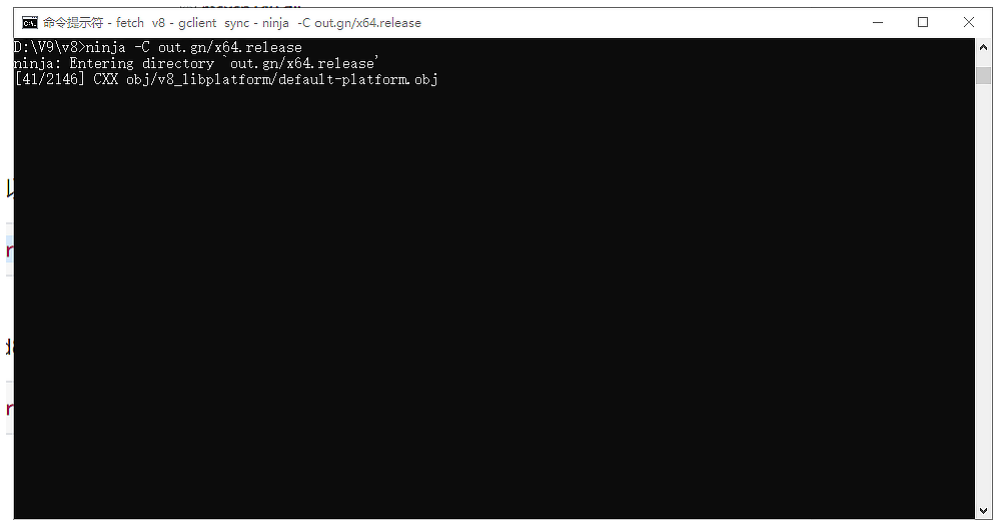
生成了d8的工程配置文件，接下来就可以编译d8了，你可以使用下面的命令：

```
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：

```
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：



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：

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
cctest.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：

```
d8 --help
```

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很多行，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：

```
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：

```
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：

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

function foo () {
  let ret = 0
  for(let i = 1; i < 7049; i++) {
    ret += bar(a)
  }
  return ret
}

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：

```
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：

```
<JSFunction foo (sfi = 0x2c730824fe21)> for optimized recompilation, reason: small function]
```

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 %
```

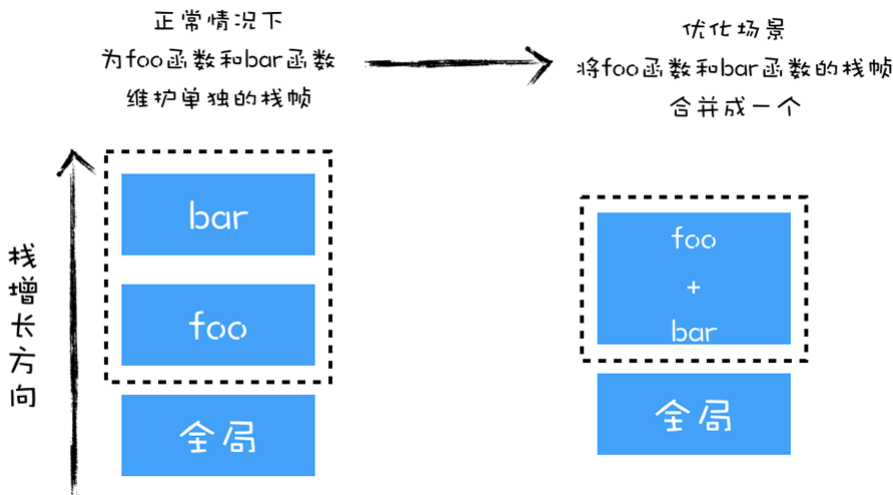
我们看到又出现了一条新的优化信息，新的提示信息如下：

```
<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：

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

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

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：

```
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 /
[97447:0x179700000000] 34 ms: Scavenge 1.2 (3.4) -> 0.3 (3.6) MB, 0.4 /
[97447:0x179700000000] 36 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 37 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 39 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 40 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 42 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 43 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 45 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 46 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 48 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 50 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.2 /
[97447:0x179700000000] 51 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.2 /
[97447:0x179700000000] 53 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 54 ms: Scavenge 0.8 (3.6) -> 0.3 (3.6) MB, 0.2 /
eos@libingdeMacBook-Pro 06 %
```

你会看到一堆提示，如下：

```
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
```

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：

```
function strToArray(str, bufferView) {
  let i = 0
  const len = str.length
  for (; i < len; ++i) {
    bufferView[i] = str.charCodeAt(i);
  }
  return bufferView;
}
function foo() {
  let i = 0
  let str = 'test V8 GC'
  let buffer = new ArrayBuffer(str.length * 2)
  let bufferView = new Uint16Array(buffer);
  while (i++ < 1e5) {
    strToArray(str,bufferView);
  }
}
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：

```
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：

```
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：

```
function Foo(property_num,element_num) {
  //添加可索引属性
  for (let i = 0; i < element_num; i++) {
    this[i] = `element${i}`
  }
  //添加常规属性
  for (let i = 0; i < property_num; i++) {
    let ppt = `property${i}`
    this[ppt] = ppt
  }
}
var bar = new Foo(10,10)
console.log(%HasFastProperties(bar));
delete bar.property2
console.log(%HasFastProperties(bar));
```

我们执行下面这个命令：

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

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。

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Linux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，解压缩包，比如你解压到以下这个路径中：

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

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gcclient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。

```
DEPOT_TOOLS_WIN_TOOLCHAIN = 0
```

这个环境变量的作用是告诉 depot\_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：

```
gcclient 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：

```
d:
mkdir v8
cd v8
fetch v8
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
I> 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
I>Cloning into 'D:\V9_gclient_v8_mta8pt'...
I>remote: Sending approximately 656.90 MiB ...
I>remote: Counting objects: 7675, done
I>remote: Finding sources: 100% (15/15)
I>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。

```
cd v8

gn gen out.gn/x64.release --args='is_debug=false target_cpu="x64" v8_target_cpu="arm64" use_goma=true'
```

如果是Mac系统，你可以使用：

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

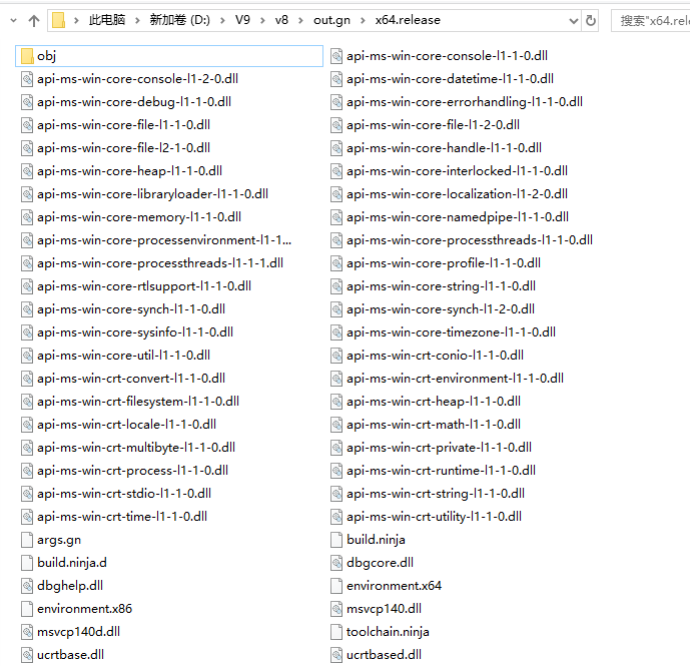
用它来生成工程。

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：



编译d8

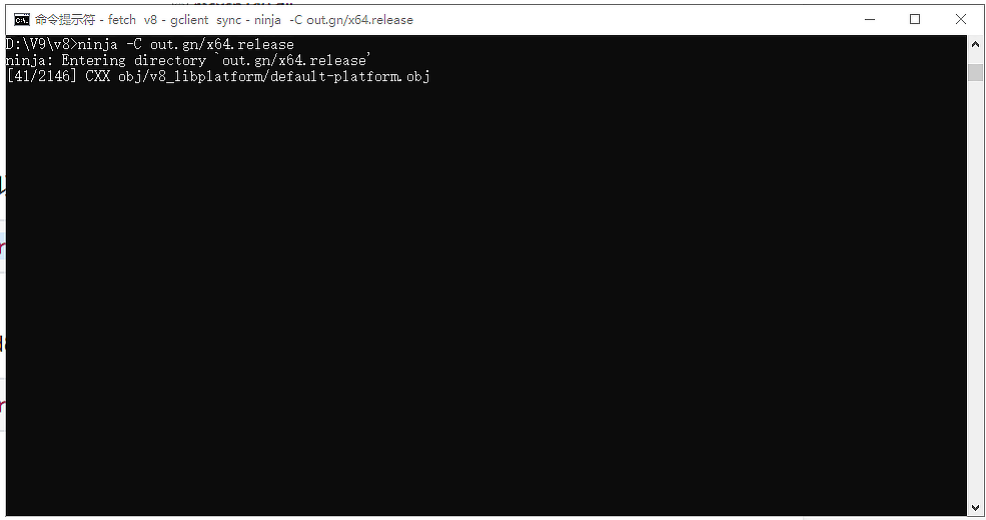
生成了d8的工程配置文件，接下来就可以编译d8了，你可以使用下面的命令：

```
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：

```
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：



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：

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
cctest.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：

```
d8 --help
```

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很多行，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：

```
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：

```
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 and 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：

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

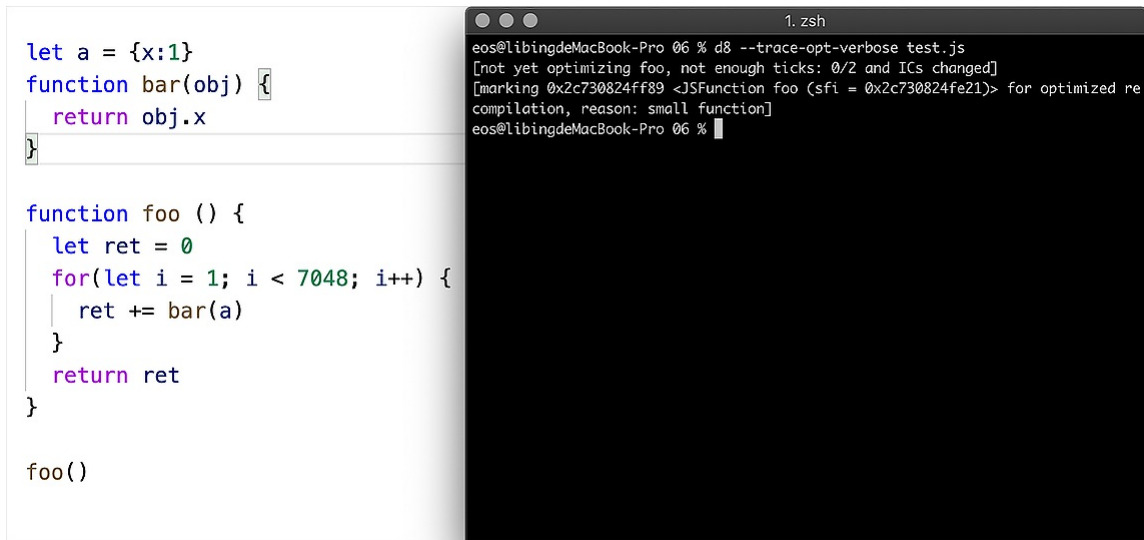
function foo () {
  let ret = 0
  for(let i = 1; i < 7049; i++) {
    ret += bar(a)
  }
  return ret
}

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：

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

执行这段命令之后，提示如下所示：

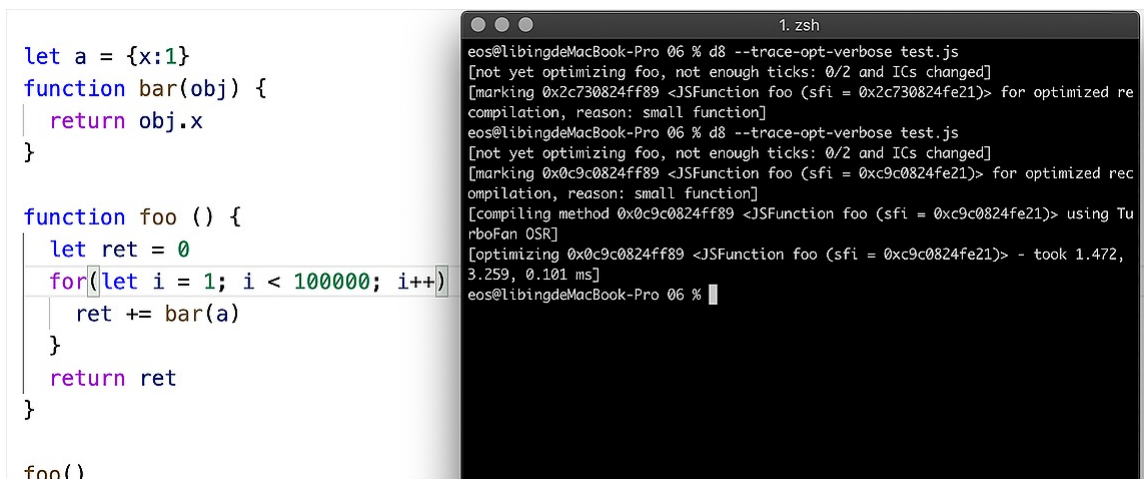


观察上图，我们可以看到终端中出现了一段优化的提示：

```
<JSFunction foo (sfi = 0x2c730824fe21)> for optimized recompilation, reason: small function]
```

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：



我们看到又出现了一条新的优化信息，新的提示信息如下：

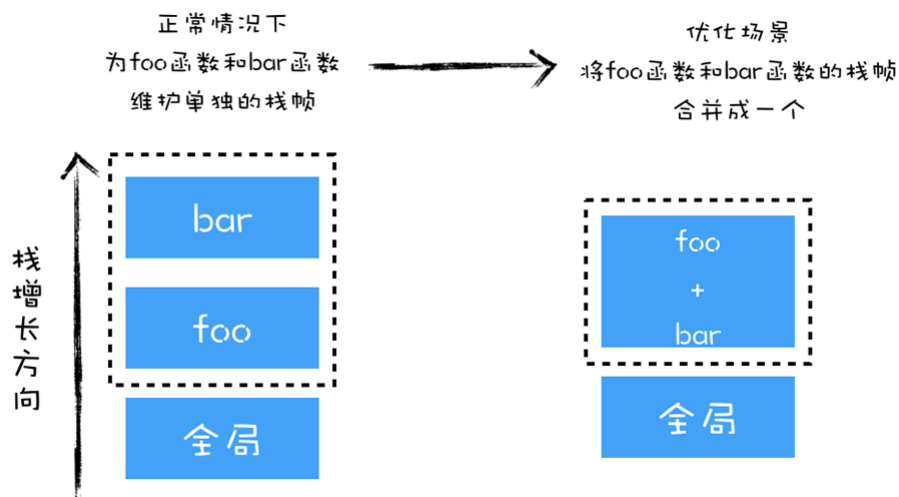
```
<JSFunction foo (sfi = 0xc9c0824fe21)> using TurboFan OSR]
```

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：

```
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()

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：

```
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 /
[97447:0x179700000000] 34 ms: Scavenge 1.2 (3.4) -> 0.3 (3.6) MB, 0.4 /
[97447:0x179700000000] 36 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 37 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 39 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 40 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 42 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 43 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 45 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 46 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 48 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 50 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.2 /
[97447:0x179700000000] 51 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.2 /
[97447:0x179700000000] 53 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 54 ms: Scavenge 0.8 (3.6) -> 0.3 (3.6) MB, 0.2 /
eos@libingdeMacBook-Pro 06 %
```

你会看到一堆提示，如下：

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

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，比如我们可以用换种方式来实现上述代码，如下所示：

```
function strToArray(str, bufferView) {
  let i = 0
  const len = str.length
  for (; i < len; ++i) {
    bufferView[i] = str.charCodeAt(i);
  }
  return bufferView;
}
function foo() {
  let i = 0
  let str = 'test V8 GC'
  let buffer = new ArrayBuffer(str.length * 2)
  let bufferView = new Uint16Array(buffer);
  while (i++ < 1e5) {
    strToArray(str,bufferView);
  }
}
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：

```
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：

```
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：

```
function Foo(property_num,element_num) {
  //添加可索引属性
  for (let i = 0; i < element_num; i++) {
    this[i] = `element${i}`
  }
  //添加常规属性
  for (let i = 0; i < property_num; i++) {
    let ppt = `property${i}`
    this[ppt] = ppt
  }
}
var bar = new Foo(10,10)
console.log(%HasFastProperties(bar));
delete bar.property2
console.log(%HasFastProperties(bar));
```

我们执行下面这个命令：

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

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。

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是一个非常实用的调试工具，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Linux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，解压缩压缩包，比如你解压到以下这个路径中：

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

然后需要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。

```
DEPOT_TOOLS_WIN_TOOLCHAIN = 0
```

这个环境变量的作用是告诉 depot\_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：

```
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。

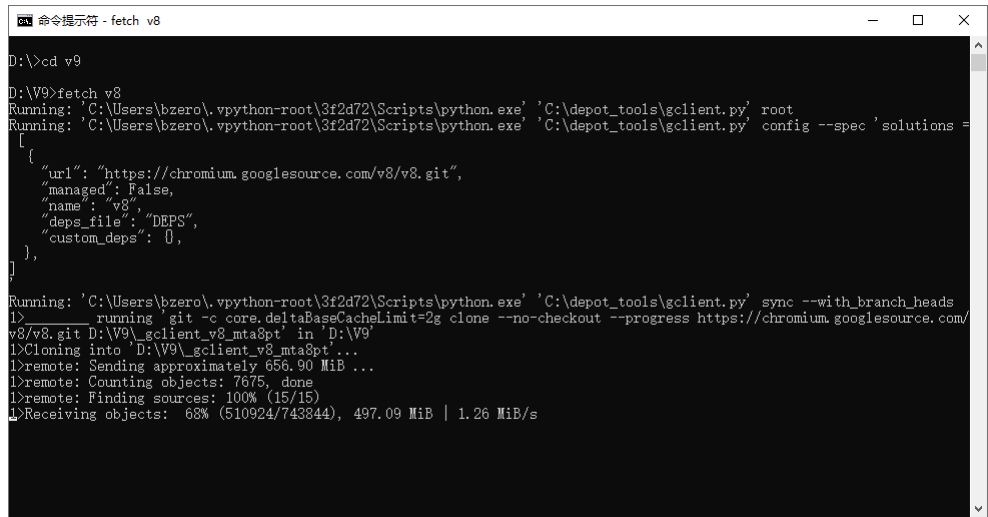
因为编译V8时，使用了这两项所提供的基础开发环境。

### 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：

```
d:
mkdir v8
cd v8
fetch v8
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。



### 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。

```
cd v8
```

```
gn gen out.gn/x64.release --args='is_debug=false target_cpu="x64" v8_target_cpu="arm64" use_goma=true'
```

如果是Mac系统，你可以使用：

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

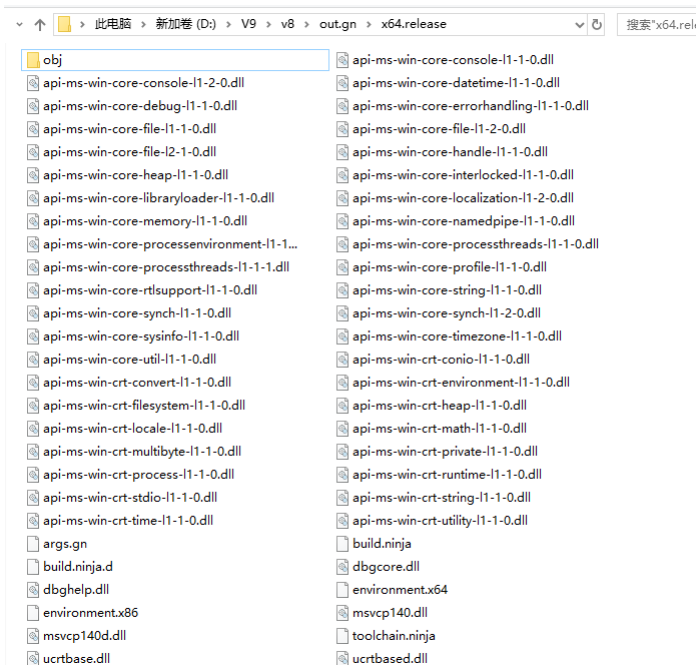
用它来生成工程。

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：



### 编译d8

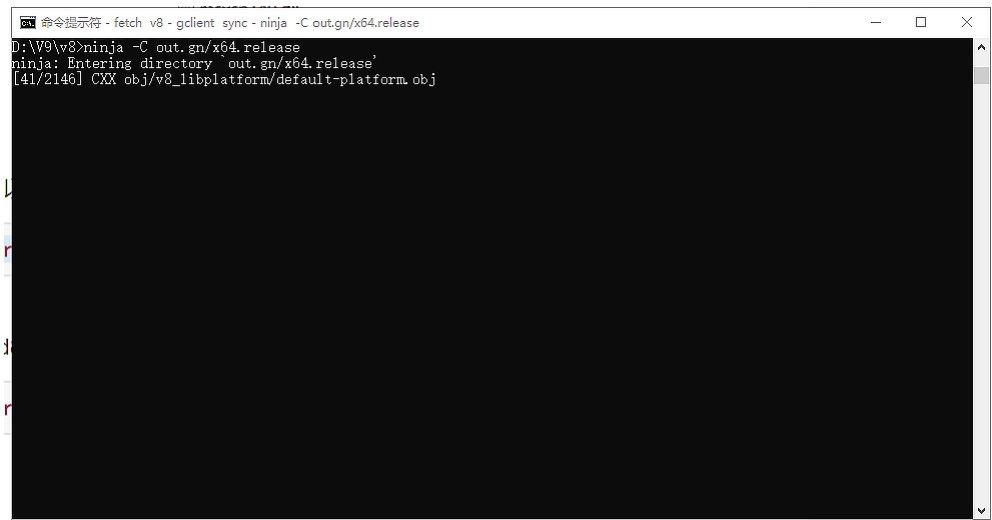
生成了d8的工程配置文件，接下来就可以编译d8了，你可以使用下面的命令：

```
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：

```
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：



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：

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
cctest.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:

```
d8 --help
```

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很多行，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：

```
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：

```
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：

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

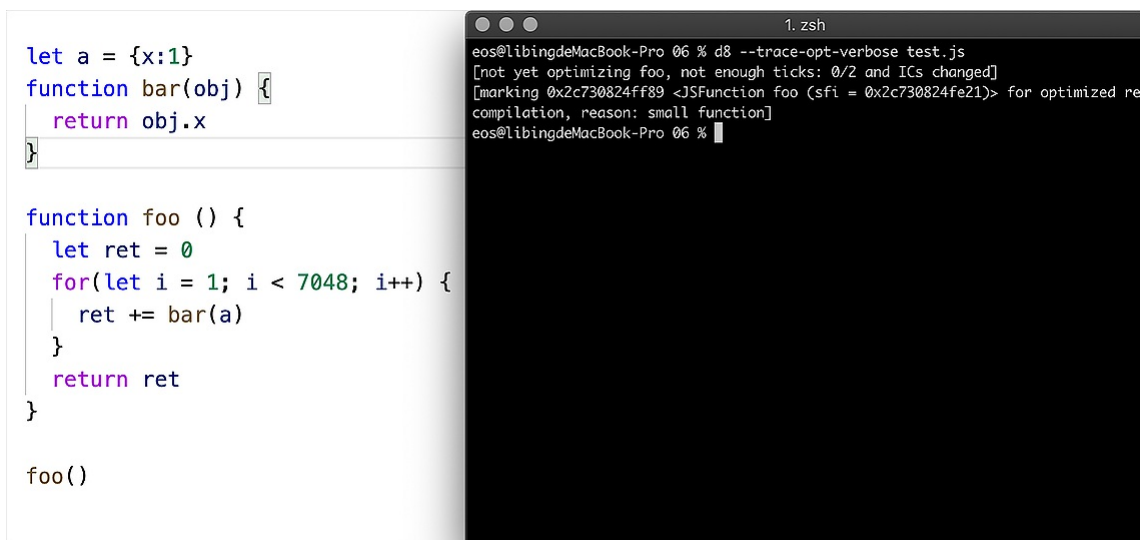
function foo () {
  let ret = 0
  for(let i = 1; i < 7049; i++) {
    ret += bar(a)
  }
  return ret
}

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：

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

执行这段命令之后，提示如下所示：



观察上图，我们可以看到终端中出现了一段优化的提示：

```
<JSFunction foo (sfi = 0x2c730824fe21)> for optimized recompilation, reason: small function]
```



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 %
```

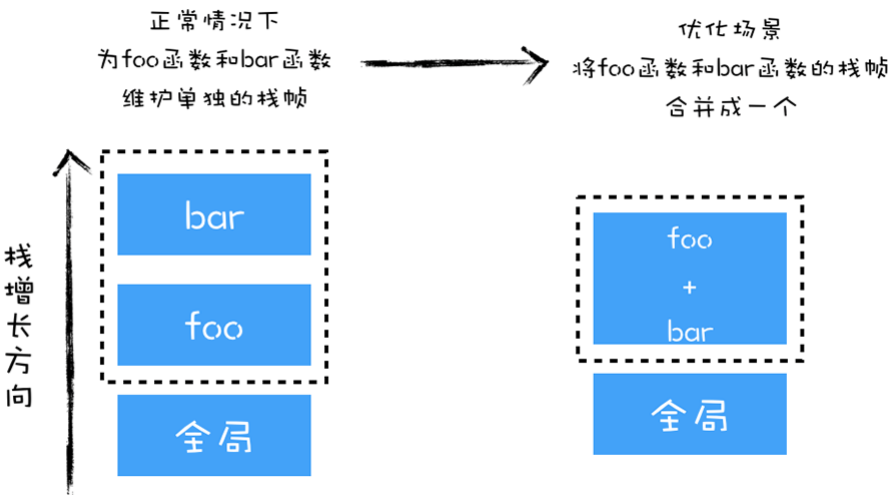
我们看到又出现了一条新的优化信息，新的提示信息如下：

```
<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：

```
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()
```

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：

```
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 /
[97447:0x179700000000] 34 ms: Scavenge 1.2 (3.4) -> 0.3 (3.6) MB, 0.4 /
[97447:0x179700000000] 36 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 37 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 39 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 40 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 42 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 43 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 45 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 46 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 48 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 50 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.2 /
[97447:0x179700000000] 51 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.2 /
[97447:0x179700000000] 53 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 54 ms: Scavenge 0.8 (3.6) -> 0.3 (3.6) MB, 0.2 /
eos@libingdeMacBook-Pro 06 %
```

你会看到一堆提示，如下：

```
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
```

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：

```
function strToArray(str, bufferView) {
  let i = 0
  const len = str.length
  for (; i < len; ++i) {
    bufferView[i] = str.charCodeAt(i);
  }
  return bufferView;
}
function foo() {
  let i = 0
  let str = 'test V8 GC'
  let buffer = new ArrayBuffer(str.length * 2)
  let bufferView = new Uint16Array(buffer);
  while (i++ < 1e5) {
    strToArray(str,bufferView);
  }
}
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：

```
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：

```
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：

```
function Foo(property_num,element_num) {
  //添加可索引属性
  for (let i = 0; i < element_num; i++) {
    this[i] = `element${i}`
  }
  //添加常规属性
  for (let i = 0; i < property_num; i++) {
    let ppt = `property${i}`
    this[ppt] = ppt
  }
}
var bar = new Foo(10,10)
console.log(%HasFastProperties(bar));
delete bar.property2
console.log(%HasFastProperties(bar));
```

我们执行下面这个命令：

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

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。

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Linux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，解压缩压缩包，比如你解压到以下这个路径中：

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

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gcclient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。

```
DEPOT_TOOLS_WIN_TOOLCHAIN = 0
```

这个环境变量的作用是告诉 depot\_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：

```
gcclient 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：

```
d:
mkdir v8
cd v8
fetch v8
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
I> 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'
I>Cloning into 'D:\V9_gclient_v8_mta8pt'...
I>remote: Sending approximately 656.90 MiB ...
I>remote: Counting objects: 7675, done
I>remote: Finding sources: 100% (15/15)
I>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。

```
cd v8

gn gen out.gn/x64.release --args='is_debug=false target_cpu="x64" v8_target_cpu="arm64" use_goma=true'
```

如果是Mac系统，你可以使用：

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

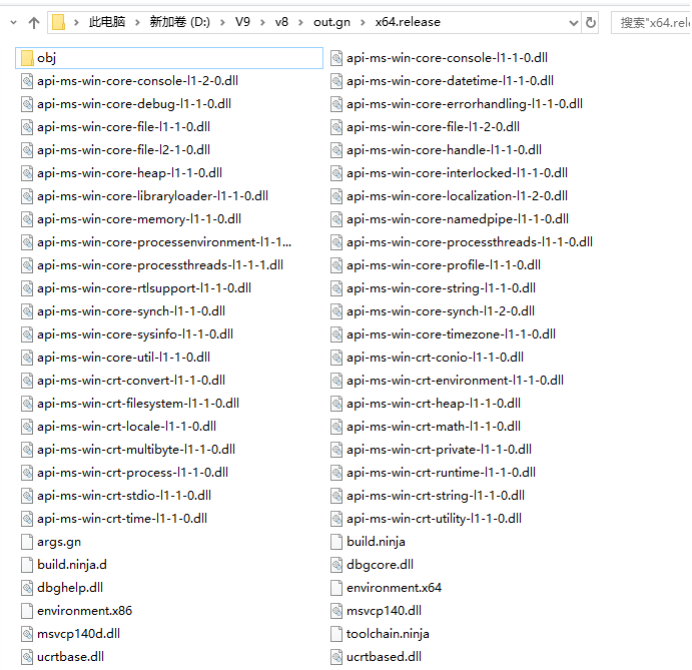
用它来生成工程。

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：



编译d8

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：

```
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：

```
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：

```
命令提示符 - fetch v8 - gclient sync - ninja -C out.gn/x64.release
D:\V9\v8>ninja -C out.gn/x64.release
ninja: Entering directory 'out.gn/x64.release'
[41/2146] CXX obj/v8_libplatform/default-platform.obj
```

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		
cctest.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：

```
d8 --help
```

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很多行，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：

```
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：

```
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 and 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：

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

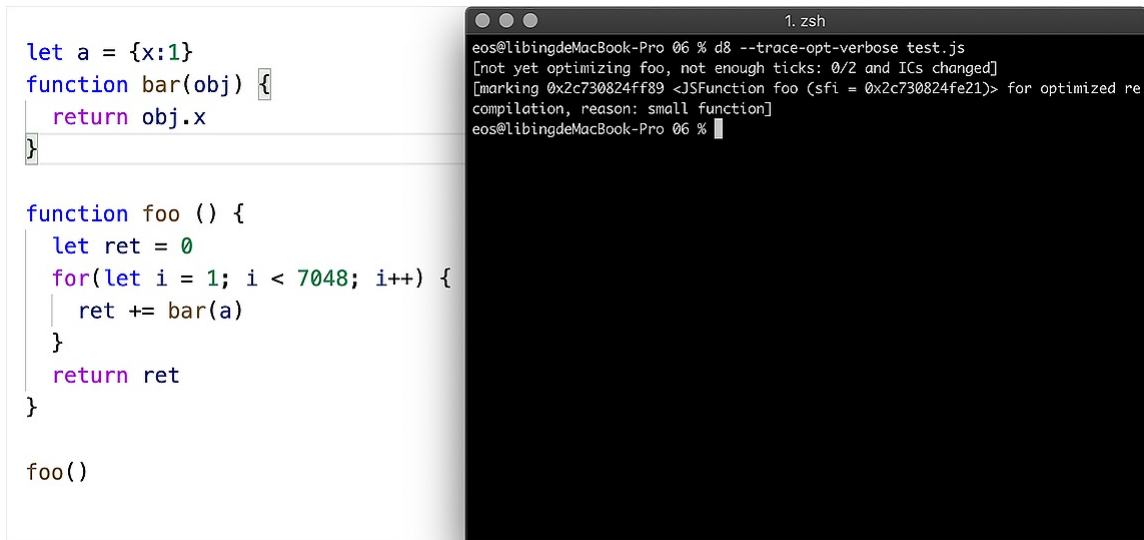
function foo () {
  let ret = 0
  for(let i = 1; i < 7049; i++) {
    ret += bar(a)
  }
  return ret
}

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：

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

执行这段命令之后，提示如下所示：

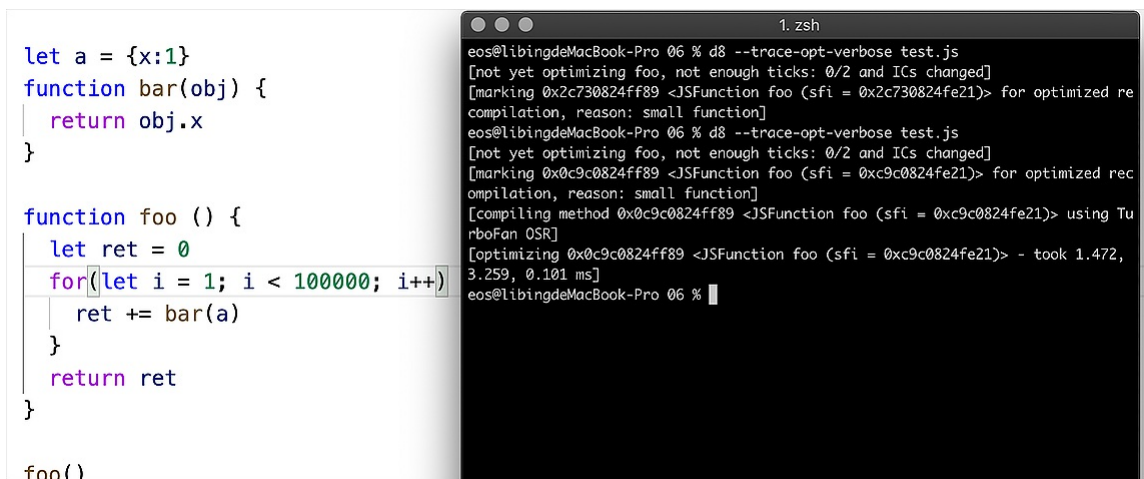


观察上图，我们可以看到终端中出现了一段优化的提示：

```
<JSFunction foo (sfi = 0x2c730824fe21)> for optimized recompilation, reason: small function]
```

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：



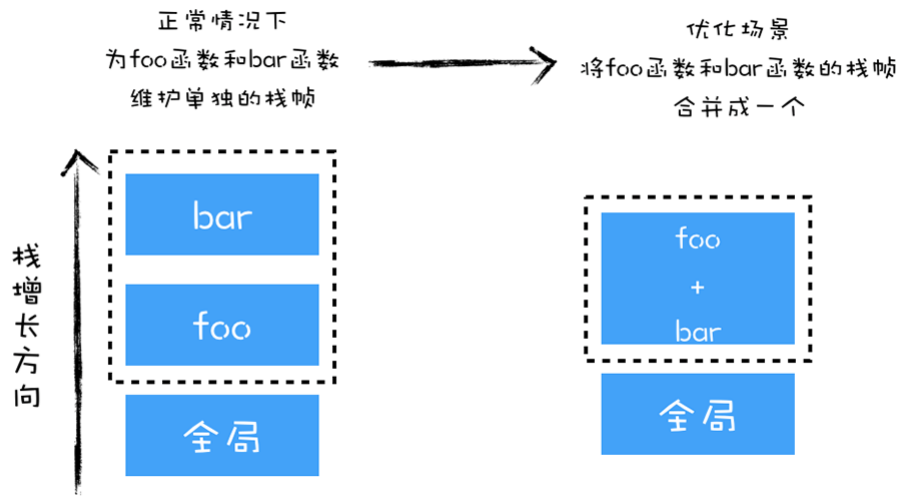
我们看到又出现了一条新的优化信息，新的提示信息如下：

```
<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：

```
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()
```

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：

```
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 /
[97447:0x179700000000] 34 ms: Scavenge 1.2 (3.4) -> 0.3 (3.6) MB, 0.4 /
[97447:0x179700000000] 36 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 37 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 39 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 40 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 42 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 43 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 45 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 46 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 48 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 50 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.2 /
[97447:0x179700000000] 51 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.2 /
[97447:0x179700000000] 53 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 54 ms: Scavenge 0.8 (3.6) -> 0.3 (3.6) MB, 0.2 /
eos@libingdeMacBook-Pro 06 %
```

你会看到一堆提示，如下：

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

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：

```
function strToArray(str, bufferView) {
  let i = 0
  const len = str.length
  for (; i < len; ++i) {
    bufferView[i] = str.charCodeAt(i);
  }
  return bufferView;
}
function foo() {
  let i = 0
  let str = 'test V8 GC'
  let buffer = new ArrayBuffer(str.length * 2)
  let bufferView = new Uint16Array(buffer);
  while (i++ < 1e5) {
    strToArray(str,bufferView);
  }
}
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：

```
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：

```
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：

```
function Foo(property_num,element_num) {
  //添加可索引属性
  for (let i = 0; i < element_num; i++) {
    this[i] = `element${i}`
  }
  //添加常规属性
  for (let i = 0; i < property_num; i++) {
    let ppt = `property${i}`
    this[ppt] = ppt
  }
}
var bar = new Foo(10,10)
console.log(%HasFastProperties(bar));
delete bar.property2
console.log(%HasFastProperties(bar));
```

我们执行下面这个命令：

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

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。

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**Linux**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，解压缩压缩包，比如你解压到以下这个路径中：

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

然后需要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。

```
DEPOT_TOOLS_WIN_TOOLCHAIN = 0
```

这个环境变量的作用是告诉 depot\_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：

```
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。

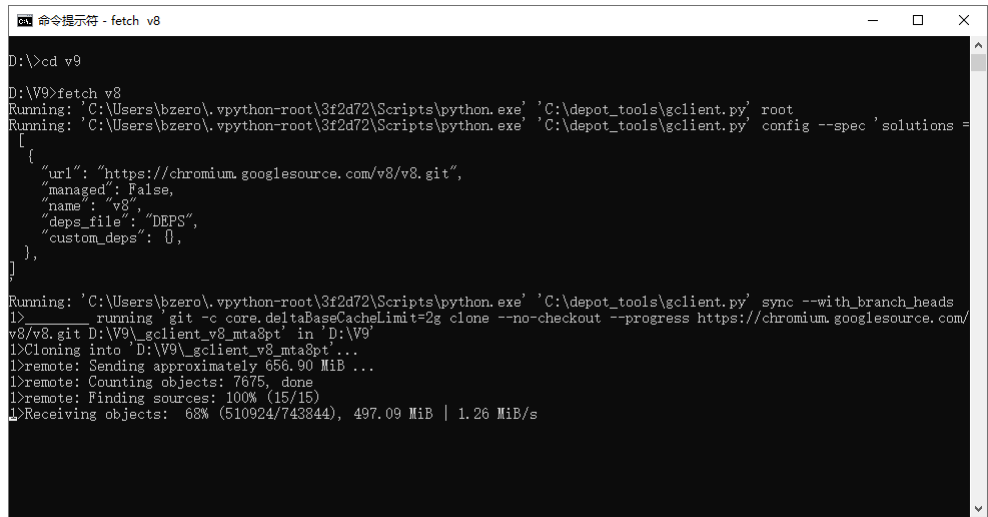
因为编译V8时，使用了这两项所提供的基础开发环境。

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：

```
d:
mkdir v8
cd v8
fetch v8
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。



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。

```
cd v8
```

```
gn gen out.gn/x64.release --args='is_debug=false target_cpu="x64" v8_target_cpu="arm64" use_goma=true'
```

如果是Mac系统，你可以使用：

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

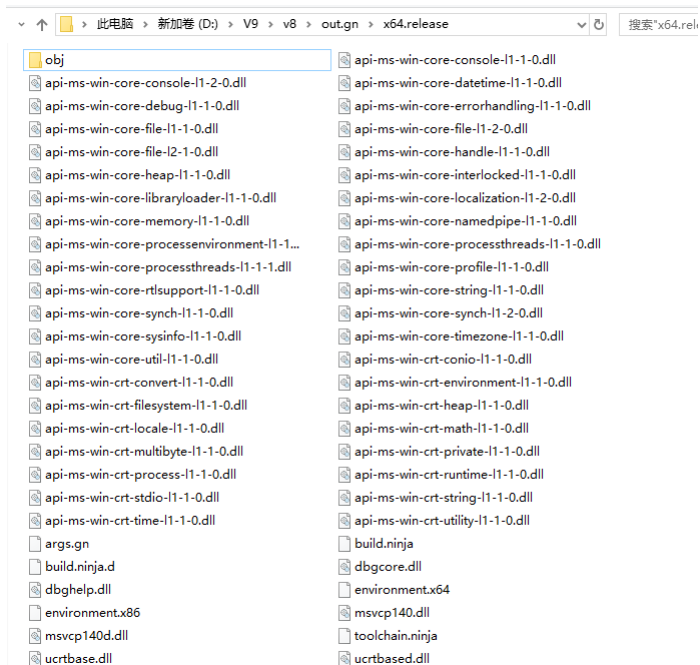
用它来生成工程。

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：



## 编译d8

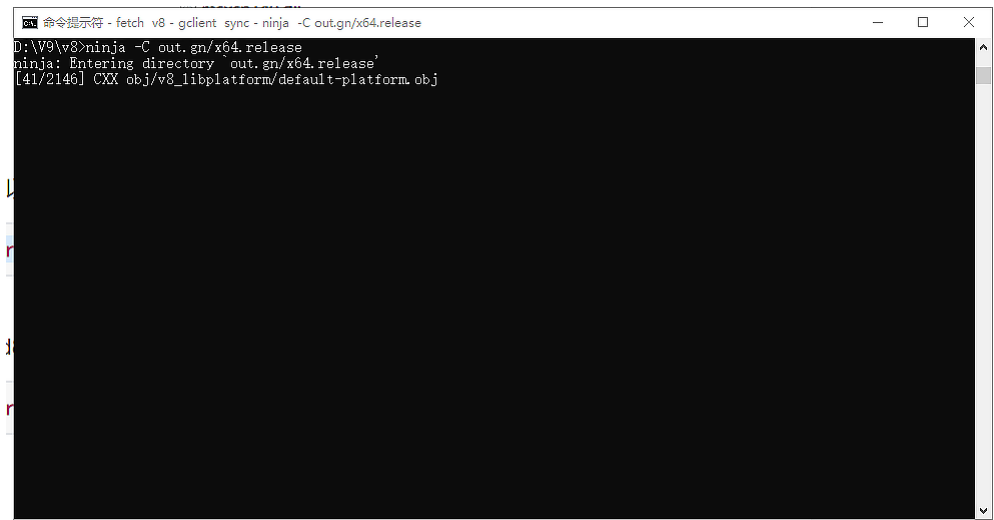
生成了d8的工程配置文件，接下来就可以编译d8了，你可以使用下面的命令：

```
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：

```
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：



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：

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
cctest.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：

```
d8 --help
```

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很多行，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：

```
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：

```
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：

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

function foo () {
  let ret = 0
  for(let i = 1; i < 7049; i++) {
    ret += bar(a)
  }
  return ret
}

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：

```
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：

```
<JSFunction foo (sfi = 0x2c730824fe21)> for optimized recompilation, reason: small function]
```

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 %
```

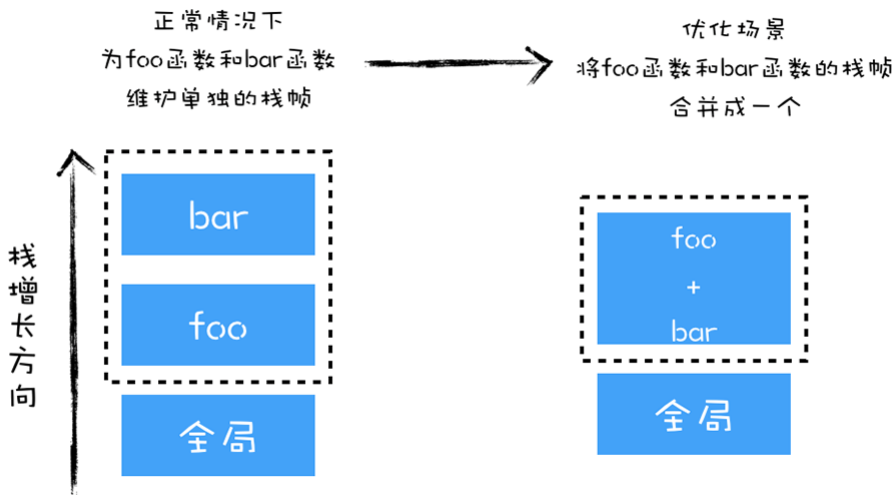
我们看到又出现了一条新的优化信息，新的提示信息如下：

```
<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：

```
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()
```

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：

```
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 /
[97447:0x179700000000] 34 ms: Scavenge 1.2 (3.4) -> 0.3 (3.6) MB, 0.4 /
[97447:0x179700000000] 36 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 37 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 39 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 40 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 42 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 43 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 45 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 46 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 48 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 50 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.2 /
[97447:0x179700000000] 51 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.2 /
[97447:0x179700000000] 53 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 54 ms: Scavenge 0.8 (3.6) -> 0.3 (3.6) MB, 0.2 /
eos@libingdeMacBook-Pro 06 %
```

你会看到一堆提示，如下：

```
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
```

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：

```
function strToArray(str, bufferView) {
  let i = 0
  const len = str.length
  for (; i < len; ++i) {
    bufferView[i] = str.charCodeAt(i);
  }
  return bufferView;
}
function foo() {
  let i = 0
  let str = 'test V8 GC'
  let buffer = new ArrayBuffer(str.length * 2)
  let bufferView = new Uint16Array(buffer);
  while (i++ < 1e5) {
    strToArray(str,bufferView);
  }
}
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：

```
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：

```
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：

```
function Foo(property_num,element_num) {
  //添加可索引属性
  for (let i = 0; i < element_num; i++) {
    this[i] = `element${i}`
  }
  //添加常规属性
  for (let i = 0; i < property_num; i++) {
    let ppt = `property${i}`
    this[ppt] = ppt
  }
}
var bar = new Foo(10,10)
console.log(%HasFastProperties(bar));
delete bar.property2
console.log(%HasFastProperties(bar));
```

我们执行下面这个命令：

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

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。

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Linux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，解压缩包，比如你解压到以下这个路径中：

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

然后需要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gcclient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。

```
DEPOT_TOOLS_WIN_TOOLCHAIN = 0
```

这个环境变量的作用是告诉 depot\_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：

```
gcclient 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：

```
d:
mkdir v8
cd v8
fetch v8
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
I> 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'
I>Cloning into 'D:\V9_gclient_v8_mta8pt'...
I>remote: Sending approximately 656.90 MiB ...
I>remote: Counting objects: 7675, done
I>remote: Finding sources: 100% (15/15)
I>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。

```
cd v8

gn gen out.gn/x64.release --args='is_debug=false target_cpu="x64" v8_target_cpu="arm64" use_goma=true'
```

如果是Mac系统，你可以使用：

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

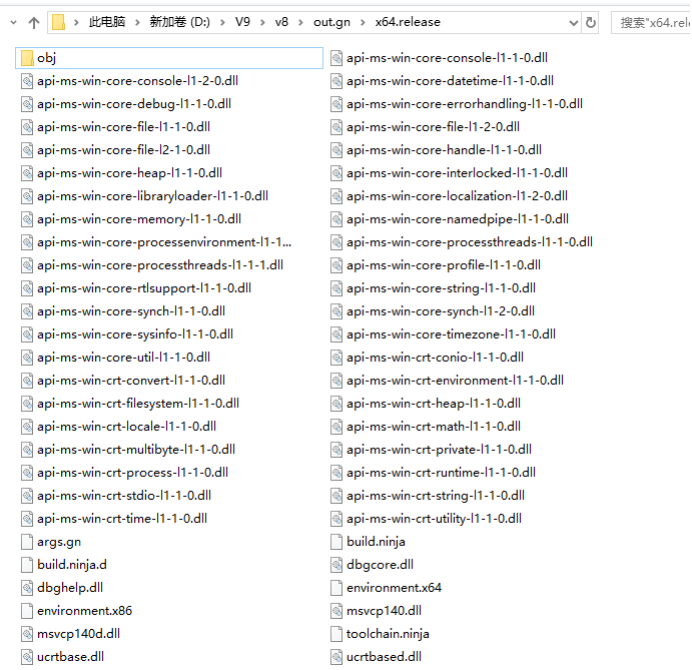
用它来生成工程。

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：



编译d8

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：

```
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：

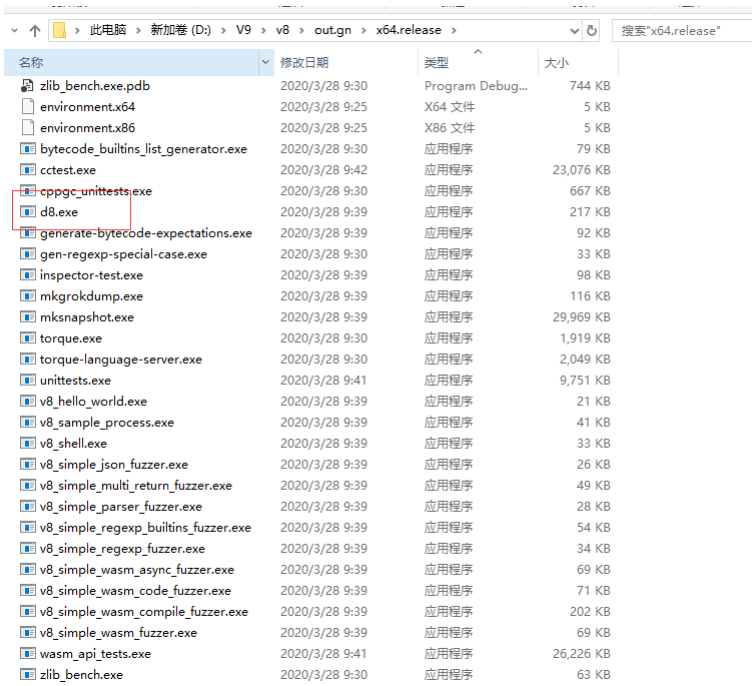
```
ninja -C out.gn/x64.release d8
```

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：

```
命令提示符 - fetch v8 - gclient sync - ninja -C out.gn/x64.release
D:\V9\v8>ninja -C out.gn/x64.release
ninja: Entering directory `out.gn/x64.release'
[41/2146] CXX obj/v8_libplatform/default-platform.obj
```

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：



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
cctest.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：

```
d8 --help
```

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很多行，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：

```
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：

```
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 and 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：

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

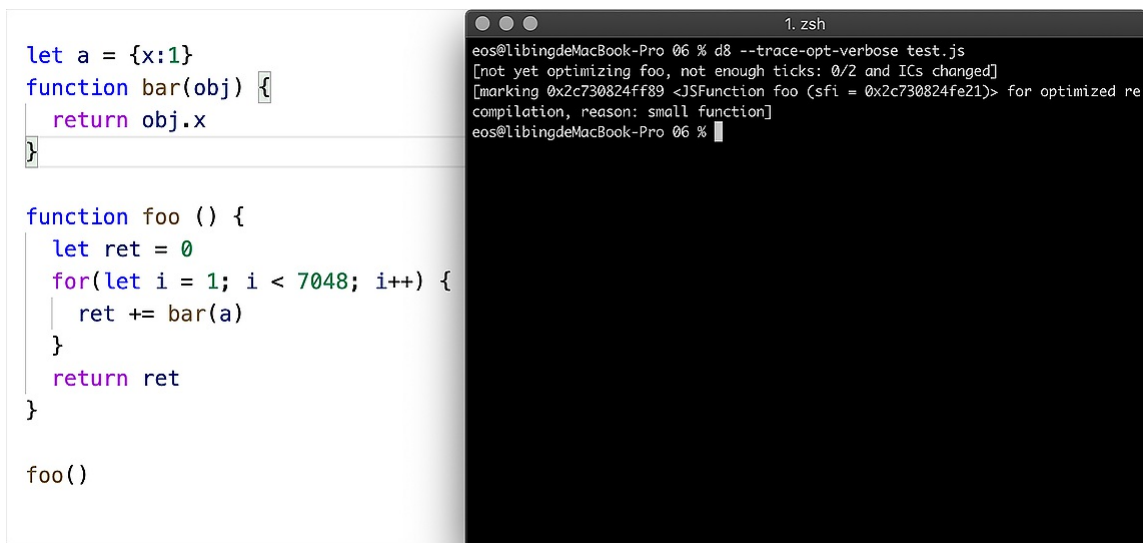
function foo () {
  let ret = 0
  for(let i = 1; i < 7049; i++) {
    ret += bar(a)
  }
  return ret
}

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：

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

执行这段命令之后，提示如下所示：

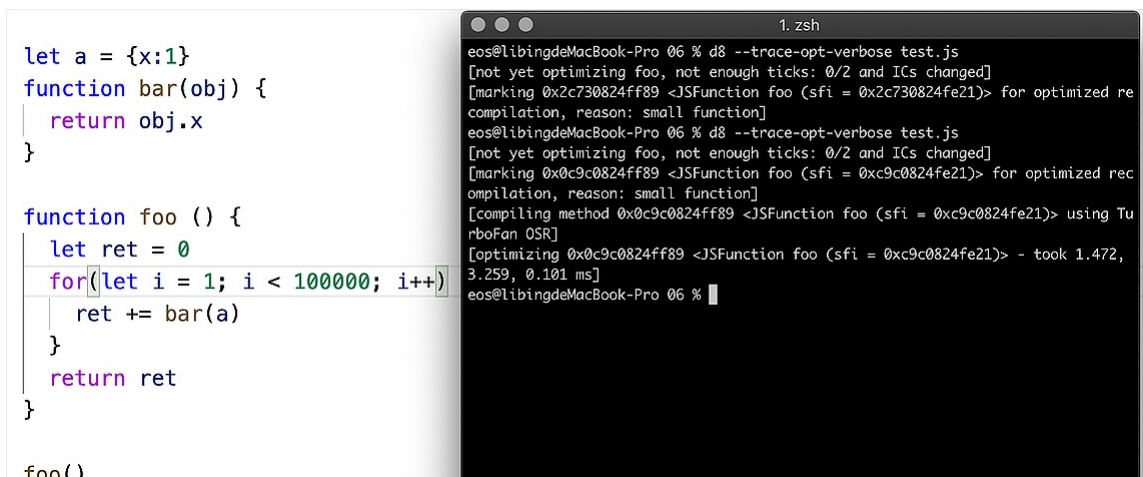


观察上图，我们可以看到终端中出现了一段优化的提示：

```
<JSFunction foo (sfi = 0x2c730824fe21)> for optimized recompilation, reason: small function]
```

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：



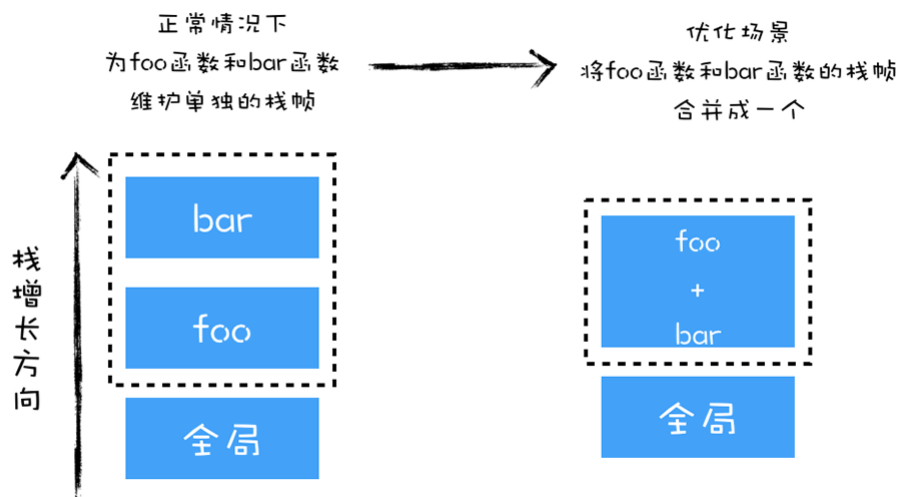
我们看到又出现了一条新的优化信息，新的提示信息如下：

```
<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：

```
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()
```

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：

```
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 /
[97447:0x179700000000] 34 ms: Scavenge 1.2 (3.4) -> 0.3 (3.6) MB, 0.4 /
[97447:0x179700000000] 36 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 37 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 39 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 40 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 42 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 43 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 45 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 46 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 48 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 50 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.2 /
[97447:0x179700000000] 51 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.2 /
[97447:0x179700000000] 53 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 54 ms: Scavenge 0.8 (3.6) -> 0.3 (3.6) MB, 0.2 /
eos@libingdeMacBook-Pro 06 %
```

你会看到一堆提示，如下：

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

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，比如我们可以以换种方式来实现上述代码，如下所示：

```
function strToArray(str, bufferView) {
  let i = 0
  const len = str.length
  for (; i < len; ++i) {
    bufferView[i] = str.charCodeAt(i);
  }
  return bufferView;
}
function foo() {
  let i = 0
  let str = 'test V8 GC'
  let buffer = new ArrayBuffer(str.length * 2)
  let bufferView = new Uint16Array(buffer);
  while (i++ < 1e5) {
    strToArray(str,bufferView);
  }
}
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：

```
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：

```
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：

```
function Foo(property_num,element_num) {
  //添加可索引属性
  for (let i = 0; i < element_num; i++) {
    this[i] = `element${i}`
  }
  //添加常规属性
  for (let i = 0; i < property_num; i++) {
    let ppt = `property${i}`
    this[ppt] = ppt
  }
}
var bar = new Foo(10,10)
console.log(%HasFastProperties(bar));
delete bar.property2
console.log(%HasFastProperties(bar));
```

我们执行下面这个命令：

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

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。

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Linux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，解压缩压缩包，比如你解压到以下这个路径中：

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

然后需要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。

```
DEPOT_TOOLS_WIN_TOOLCHAIN = 0
```

这个环境变量的作用是告诉 depot\_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：

```
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。

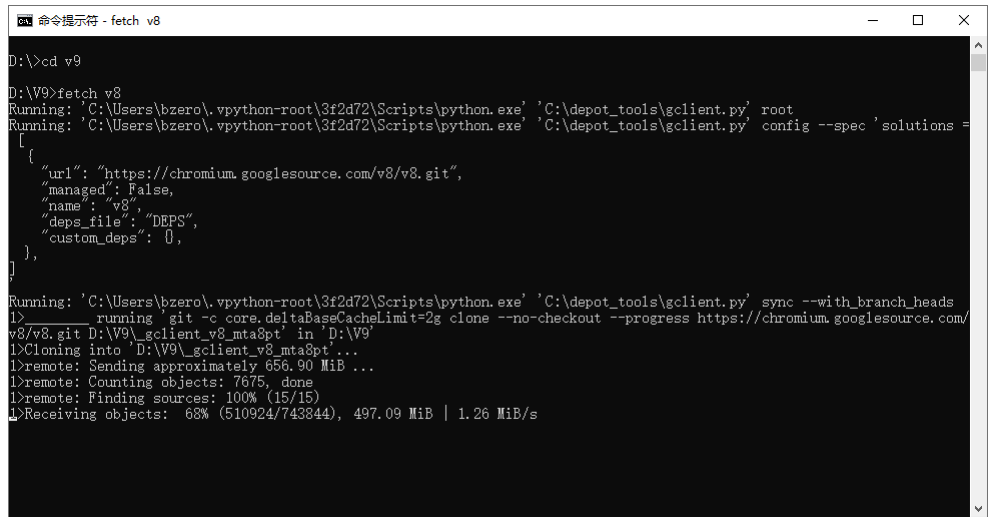
因为编译V8时，使用了这两项所提供的基础开发环境。

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：

```
d:
mkdir v8
cd v8
fetch v8
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。



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。

```
cd v8
```

```
gn gen out.gn/x64.release --args='is_debug=false target_cpu="x64" v8_target_cpu="arm64" use_goma=true'
```

如果是Mac系统，你可以使用：

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

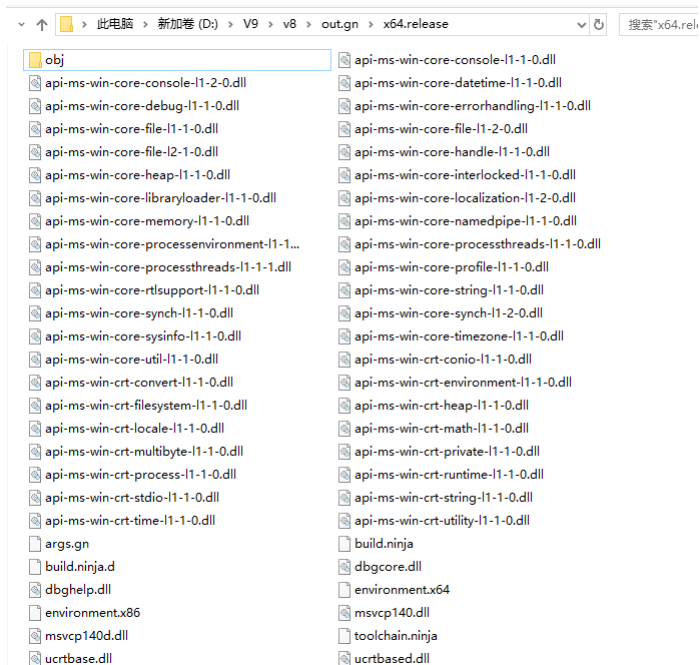
用它来生成工程。

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：



## 编译d8

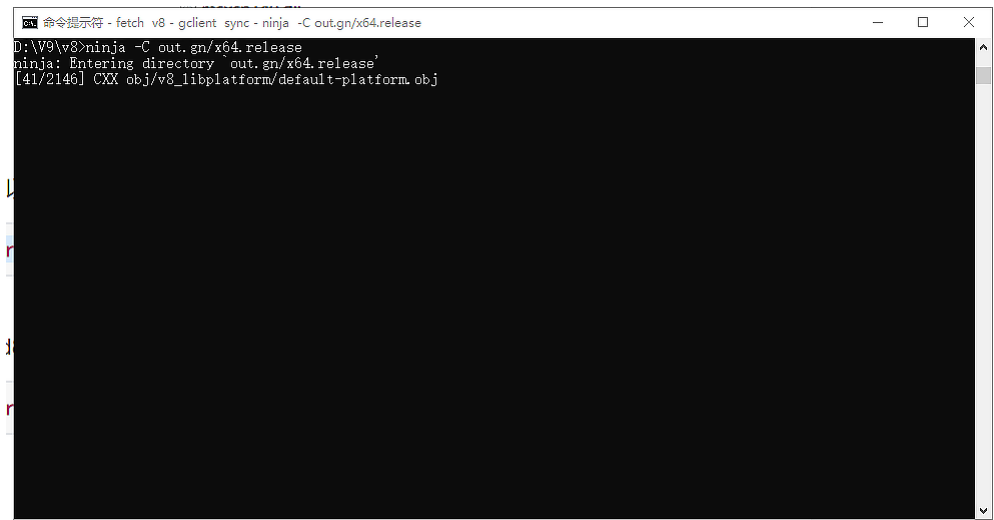
生成了d8的工程配置文件，接下来就可以编译d8了，你可以使用下面的命令：

```
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：

```
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：



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：



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
cctest.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:

```
d8 --help
```

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很多行，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：

```
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：

```
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：

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

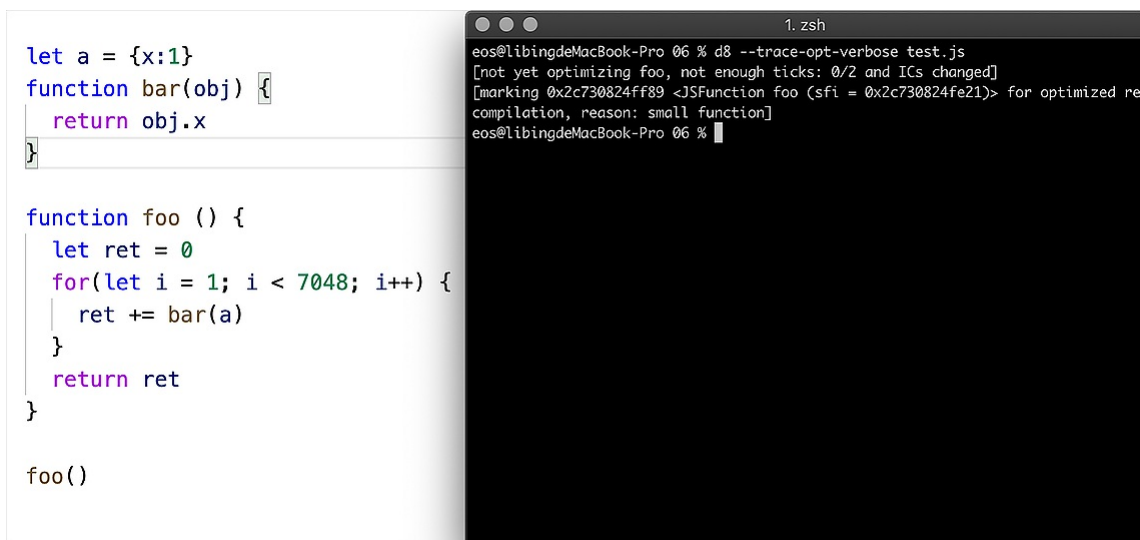
function foo () {
  let ret = 0
  for(let i = 1; i < 7049; i++) {
    ret += bar(a)
  }
  return ret
}

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：

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

执行这段命令之后，提示如下所示：



```
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：

```
<JSFunction foo (sfi = 0x2c730824fe21)> for optimized recompilation, reason: small function]
```

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 %
```

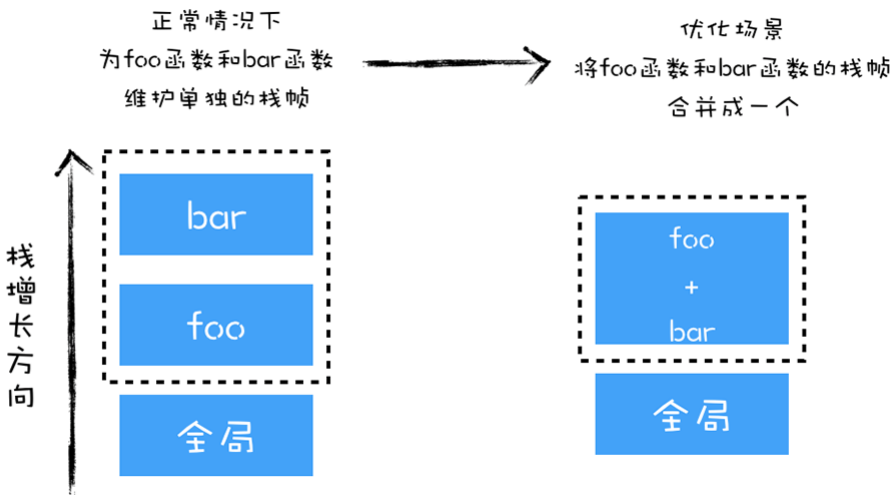
我们看到又出现了一条新的优化信息，新的提示信息如下：

```
<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：

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

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

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：

```
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()
```

你会看到一堆提示，如下：

```
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
```

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：

```
function strToArray(str, bufferView) {
  let i = 0
  const len = str.length
  for (; i < len; ++i) {
    bufferView[i] = str.charCodeAt(i);
  }
  return bufferView;
}
function foo() {
  let i = 0
  let str = 'test V8 GC'
  let buffer = new ArrayBuffer(str.length * 2)
  let bufferView = new Uint16Array(buffer);
  while (i++ < 1e5) {
    strToArray(str,bufferView);
  }
}
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：

```
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：

```
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：

```
function Foo(property_num,element_num) {
  //添加可索引属性
  for (let i = 0; i < element_num; i++) {
    this[i] = `element${i}`
  }
  //添加常规属性
  for (let i = 0; i < property_num; i++) {
    let ppt = `property${i}`
    this[ppt] = ppt
  }
}
var bar = new Foo(10,10)
console.log(%HasFastProperties(bar));
delete bar.property2
console.log(%HasFastProperties(bar));
```

我们执行下面这个命令：

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

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。

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Linux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，解压缩压缩包，比如你解压到以下这个路径中：

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

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gcclient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。

```
DEPOT_TOOLS_WIN_TOOLCHAIN = 0
```

这个环境变量的作用是告诉 depot\_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：

```
gcclient 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：

```
d:
mkdir v8
cd v8
fetch v8
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
I> 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'
I>Cloning into 'D:\V9_gclient_v8_mta8pt'...
I>remote: Sending approximately 656.90 MiB ...
I>remote: Counting objects: 7675, done
I>remote: Finding sources: 100% (15/15)
I>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。

```
cd v8

gn gen out.gn/x64.release --args='is_debug=false target_cpu="x64" v8_target_cpu="arm64" use_goma=true'
```

如果是Mac系统，你可以使用：

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

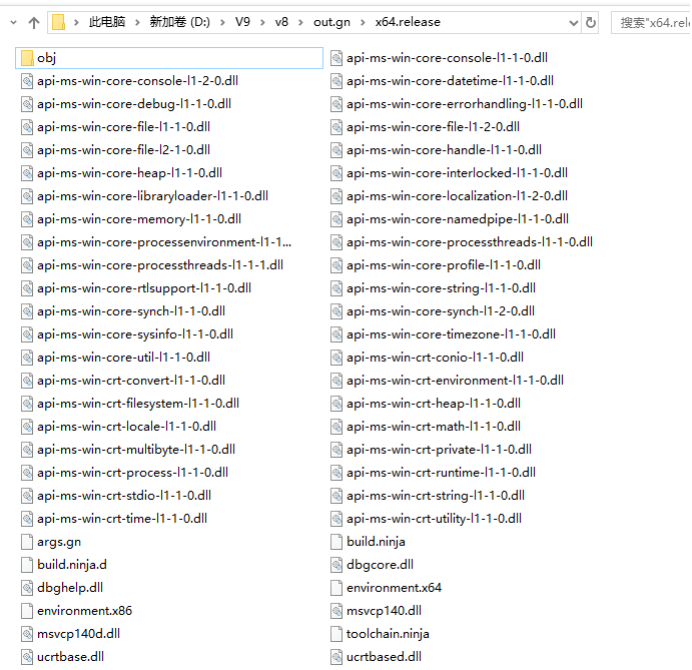
用它来生成工程。

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：



编译d8

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：

```
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：

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

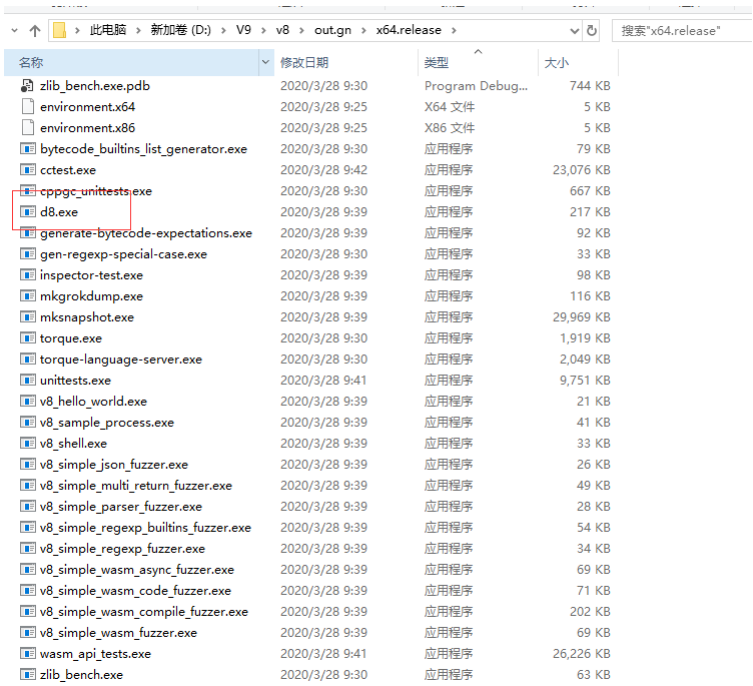
这个命令只会编译和d8所依赖的工程，然后就开始执行编译流程了。如下图所示：



```
命令提示符 - fetch v8 - gclient sync - ninja -C out.gn/x64.release
D:\V9\v8>ninja -C out.gn/x64.release
ninja: Entering directory `out.gn/x64.release'
[41/2146] CXX obj/v8_libplatform/default-platform.obj
```

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：



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
cctest.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：

```
d8 --help
```

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很多行，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：

```
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：

```
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 and 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：

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

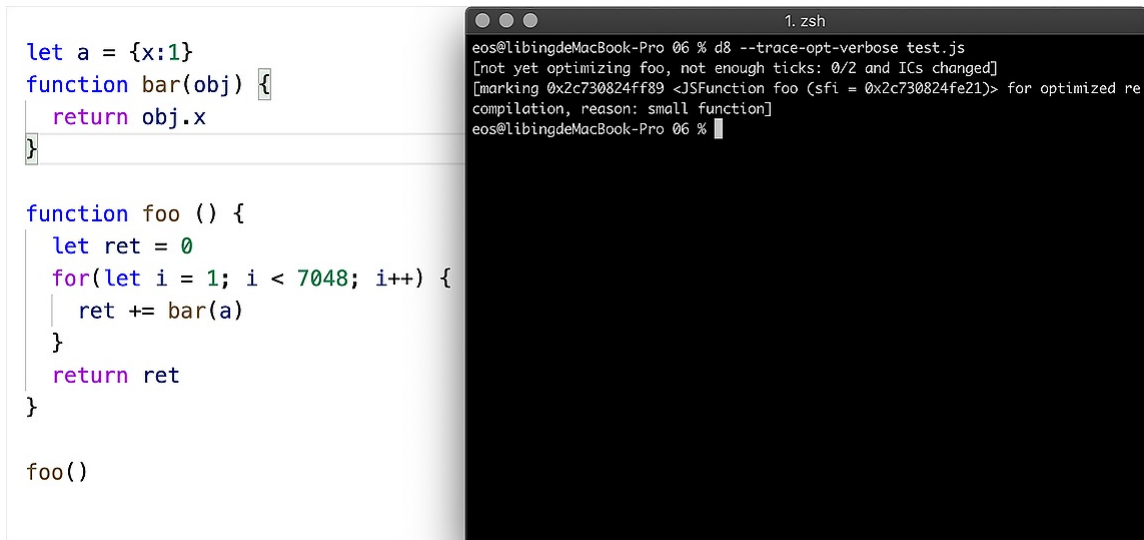
function foo () {
  let ret = 0
  for(let i = 1; i < 7049; i++) {
    ret += bar(a)
  }
  return ret
}

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：

```
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()
```

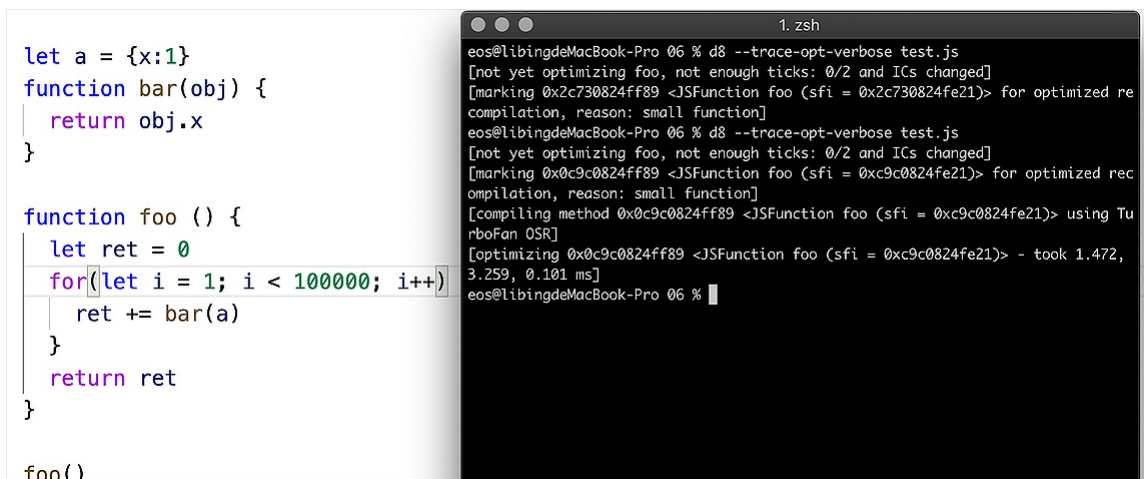
```
1. zsh
eos@libingdeMacBook-Pro 06 % d8 --trace-opt-verbose test.js
[not yet optimizing foo, not enough ticks: 0/2 and ICs changed]
[marking 0x2c730824ff89 <JSFunction foo (sfi = 0x2c730824fe21)> for optimized re
compilation, reason: small function]
eos@libingdeMacBook-Pro 06 %
```

观察上图，我们可以看到终端中出现了一段优化的提示：

```
<JSFunction foo (sfi = 0x2c730824fe21)> for optimized recompilation, reason: small function]
```

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 %
```

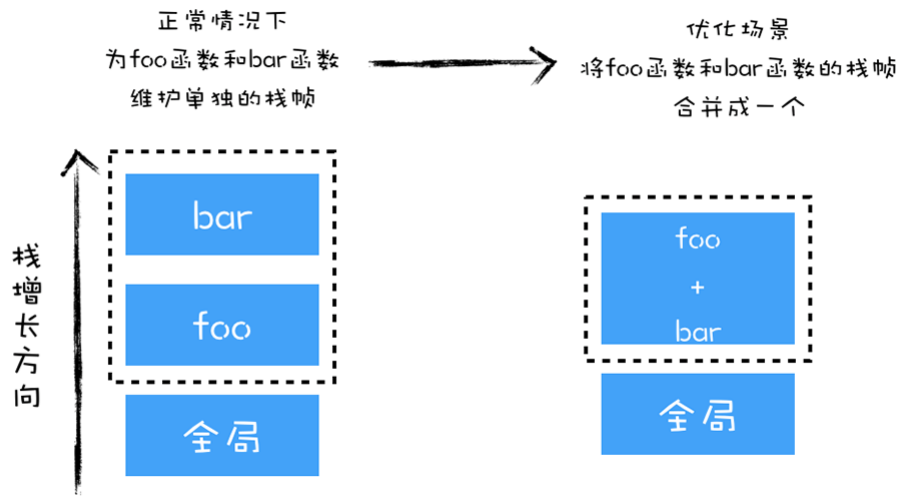
我们看到又出现了一条新的优化信息，新的提示信息如下：

```
<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：

```
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()
```

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：

```
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 /  
[97447:0x179700000000] 34 ms: Scavenge 1.2 (3.4) -> 0.3 (3.6) MB, 0.4 /  
[97447:0x179700000000] 36 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /  
[97447:0x179700000000] 37 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /  
[97447:0x179700000000] 39 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /  
[97447:0x179700000000] 40 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /  
[97447:0x179700000000] 42 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /  
[97447:0x179700000000] 43 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /  
[97447:0x179700000000] 45 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /  
[97447:0x179700000000] 46 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /  
[97447:0x179700000000] 48 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /  
[97447:0x179700000000] 50 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.2 /  
[97447:0x179700000000] 51 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.2 /  
[97447:0x179700000000] 53 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /  
[97447:0x179700000000] 54 ms: Scavenge 0.8 (3.6) -> 0.3 (3.6) MB, 0.2 /  
eos@libingdeMacBook-Pro 06 %
```

你会看到一堆提示，如下：

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

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，比如我们可以以换种方式来实现上述代码，如下所示：

```
function strToArray(str, bufferView) {
  let i = 0
  const len = str.length
  for (; i < len; ++i) {
    bufferView[i] = str.charCodeAt(i);
  }
  return bufferView;
}
function foo() {
  let i = 0
  let str = 'test V8 GC'
  let buffer = new ArrayBuffer(str.length * 2)
  let bufferView = new Uint16Array(buffer);
  while (i++ < 1e5) {
    strToArray(str,bufferView);
  }
}
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：

```
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：

```
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：

```
function Foo(property_num,element_num) {
  //添加可索引属性
  for (let i = 0; i < element_num; i++) {
    this[i] = `element${i}`
  }
  //添加常规属性
  for (let i = 0; i < property_num; i++) {
    let ppt = `property${i}`
    this[ppt] = ppt
  }
}
var bar = new Foo(10,10)
console.log(%HasFastProperties(bar));
delete bar.property2
console.log(%HasFastProperties(bar));
```

我们执行下面这个命令：

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

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。

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Linux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，解压缩压缩包，比如你解压到以下这个路径中：

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

之后需要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。

```
DEPOT_TOOLS_WIN_TOOLCHAIN = 0
```

这个环境变量的作用是告诉 depot\_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：

```
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。

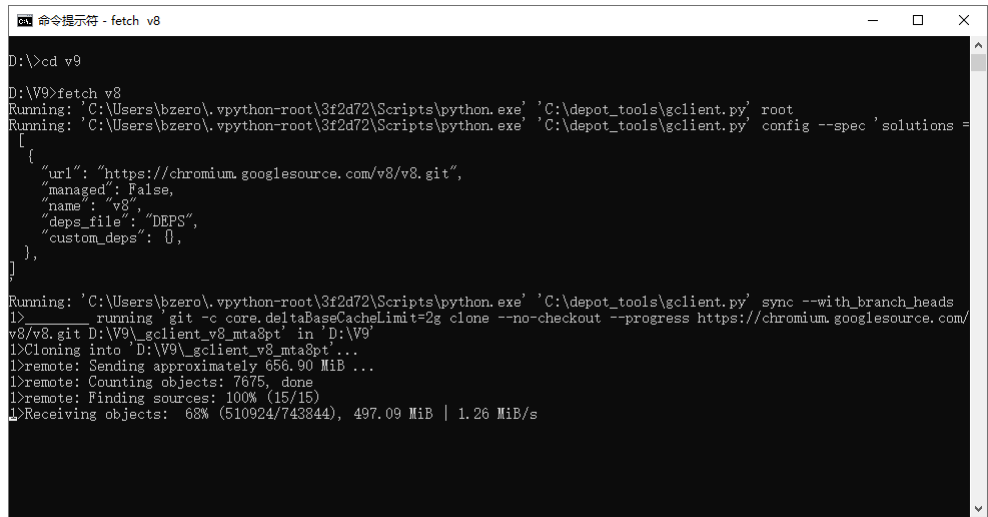
因为编译V8时，使用了这两项所提供的基础开发环境。

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：

```
d:
mkdir v8
cd v8
fetch v8
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。



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。

```
cd v8
```

```
gn gen out.gn/x64.release --args='is_debug=false target_cpu="x64" v8_target_cpu="arm64" use_goma=true'
```

如果是Mac系统，你可以使用：

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

用它来生成工程。

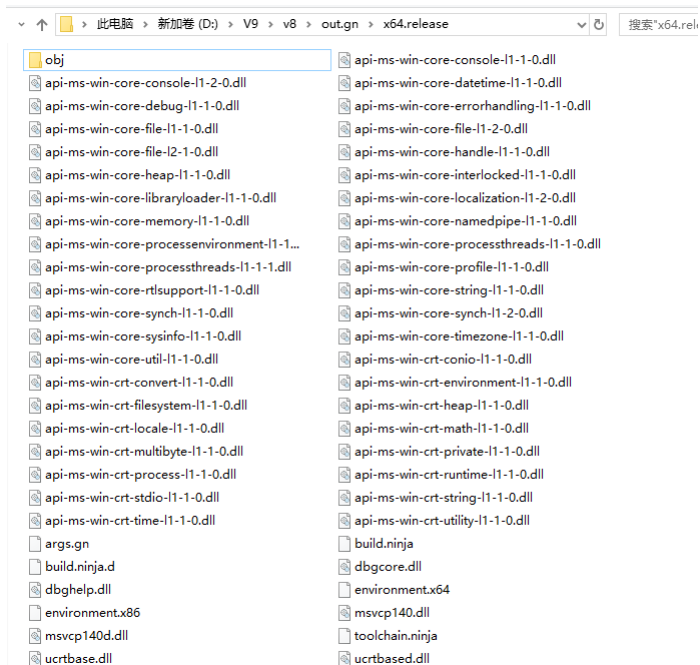
gn是一个跨平台的构建系统，用来构建Ninja工程，Ninja是一个跨平台的编译系统，比如可以通过gn构建Chromium还有V8的工程文件，然后使用Ninja来执行编译，可以使用gn和Ninja来配合使用构建跨平台的工程，这些工程可以在MacOS、Linux、Windows等平台上进行编译。在gn之前，Google使用了gyp来构建，由于gn的效率更高，所以现在都在使用gn。

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：





## 编译d8

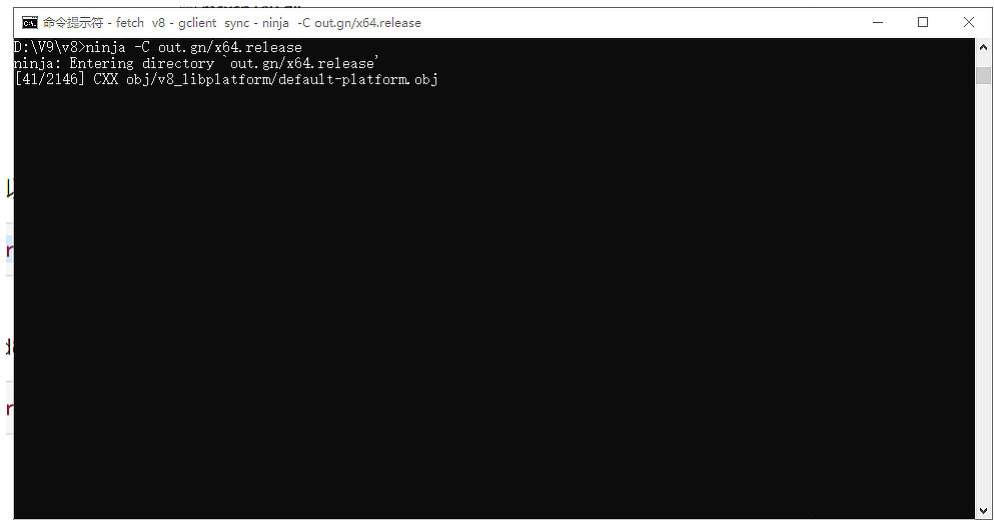
生成了d8的工程配置文件，接下来就可以编译d8了，你可以使用下面的命令：

```
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：

```
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：



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：

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
cctest.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：

```
d8 --help
```

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很多行，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：

```
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：

```
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：

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

function foo () {
  let ret = 0
  for(let i = 1; i < 7049; i++) {
    ret += bar(a)
  }
  return ret
}

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：

```
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：

```
<JSFunction foo (sfi = 0x2c730824fe21)> for optimized recompilation, reason: small function]
```

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 %
```

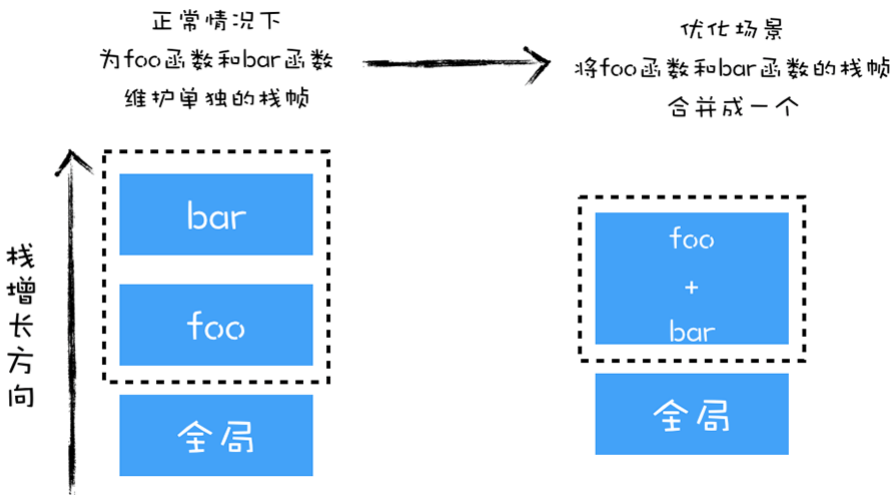
我们看到又出现了一条新的优化信息，新的提示信息如下：

```
<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：

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

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

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：

```
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 /
[97447:0x179700000000] 34 ms: Scavenge 1.2 (3.4) -> 0.3 (3.6) MB, 0.4 /
[97447:0x179700000000] 36 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 37 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 39 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 40 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 42 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 43 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 45 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 46 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 48 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 50 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.2 /
[97447:0x179700000000] 51 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.2 /
[97447:0x179700000000] 53 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /
[97447:0x179700000000] 54 ms: Scavenge 0.8 (3.6) -> 0.3 (3.6) MB, 0.2 /
eos@libingdeMacBook-Pro 06 %
```

你会看到一堆提示，如下：

```
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
```

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：

```
function strToArray(str, bufferView) {
  let i = 0
  const len = str.length
  for (; i < len; ++i) {
    bufferView[i] = str.charCodeAt(i);
  }
  return bufferView;
}
function foo() {
  let i = 0
  let str = 'test V8 GC'
  let buffer = new ArrayBuffer(str.length * 2)
  let bufferView = new Uint16Array(buffer);
  while (i++ < 1e5) {
    strToArray(str,bufferView);
  }
}
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：

```
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：

```
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：

```
function Foo(property_num,element_num) {
  //添加可索引属性
  for (let i = 0; i < element_num; i++) {
    this[i] = `element${i}`
  }
  //添加常规属性
  for (let i = 0; i < property_num; i++) {
    let ppt = `property${i}`
    this[ppt] = ppt
  }
}
var bar = new Foo(10,10)
console.log(%HasFastProperties(bar));
delete bar.property2
console.log(%HasFastProperties(bar));
```

我们执行下面这个命令：

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

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。

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Linux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，解压缩压缩包，比如你解压到以下这个路径中：

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

然后需要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gcclient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。

```
DEPOT_TOOLS_WIN_TOOLCHAIN = 0
```

这个环境变量的作用是告诉 depot\_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：

```
gcclient 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：

```
d:
mkdir v8
cd v8
fetch v8
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
I> 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'
I>Cloning into 'D:\V9_gclient_v8_mta8pt'...
I>remote: Sending approximately 656.90 MiB ...
I>remote: Counting objects: 7675, done
I>remote: Finding sources: 100% (15/15)
I>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。

```
cd v8

gn gen out.gn/x64.release --args='is_debug=false target_cpu="x64" v8_target_cpu="arm64" use_goma=true'
```

如果是Mac系统，你可以使用：

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

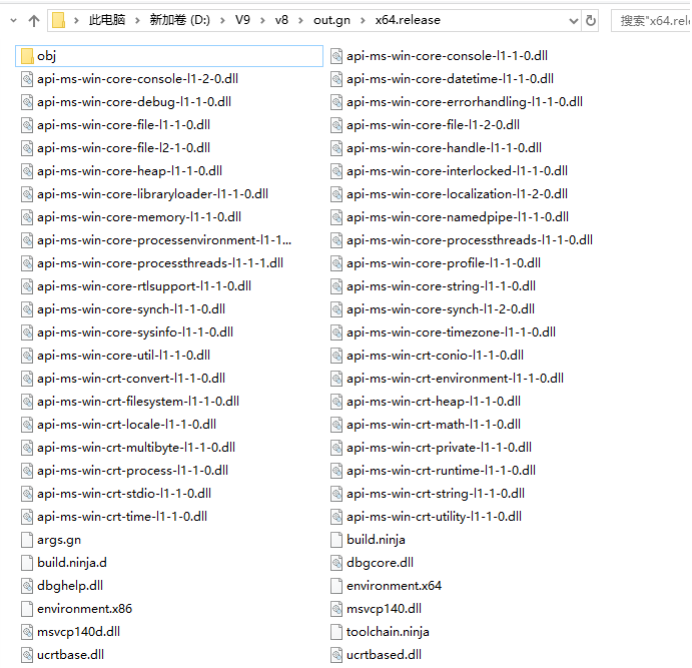
用它来生成工程。

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：



编译d8

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：

```
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：

```
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：

```
命令提示符 - fetch v8 - gclient sync - ninja -C out.gn/x64.release
D:\V9\v8>ninja -C out.gn/x64.release
ninja: Entering directory `out.gn/x64.release'
[41/2146] CXX obj/v8_libplatform/default-platform.obj
```

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		
cctest.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：

```
d8 --help
```

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很多行，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：

```
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：

```
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 and 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：

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

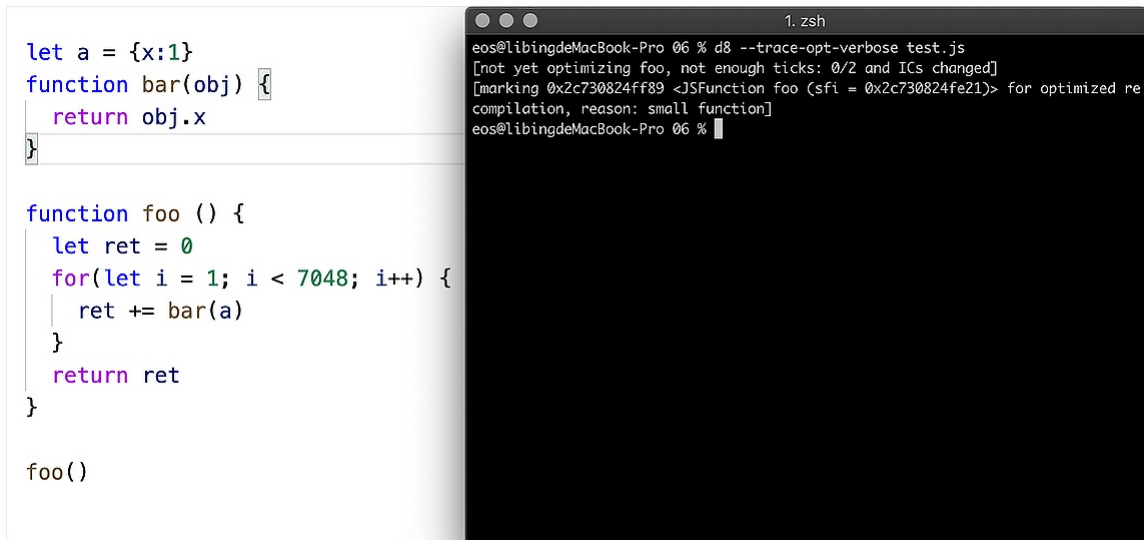
function foo () {
  let ret = 0
  for(let i = 1; i < 7049; i++) {
    ret += bar(a)
  }
  return ret
}

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：

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

执行这段命令之后，提示如下所示：

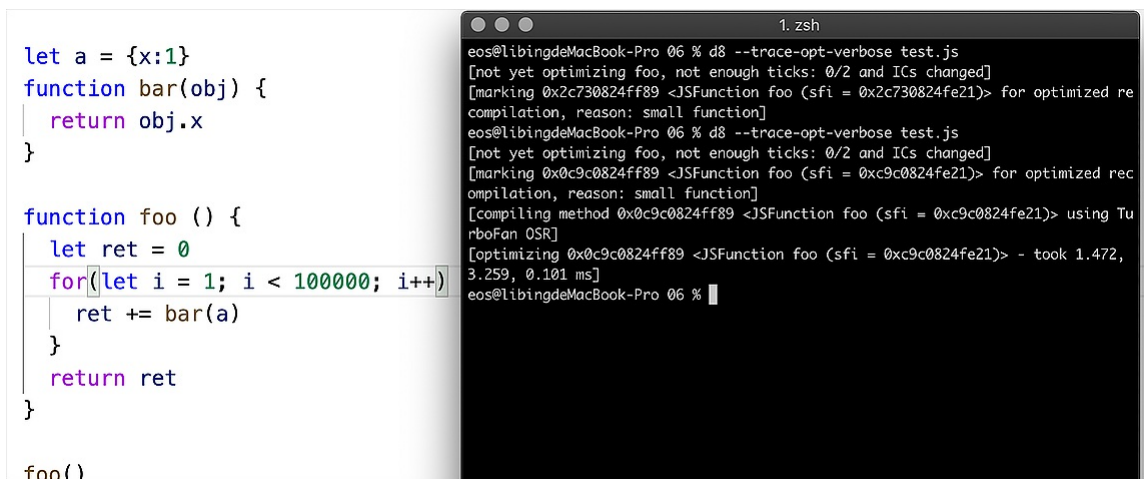


观察上图，我们可以看到终端中出现了一段优化的提示：

```
<JSFunction foo (sfi = 0x2c730824fe21)> for optimized recompilation, reason: small function]
```

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：



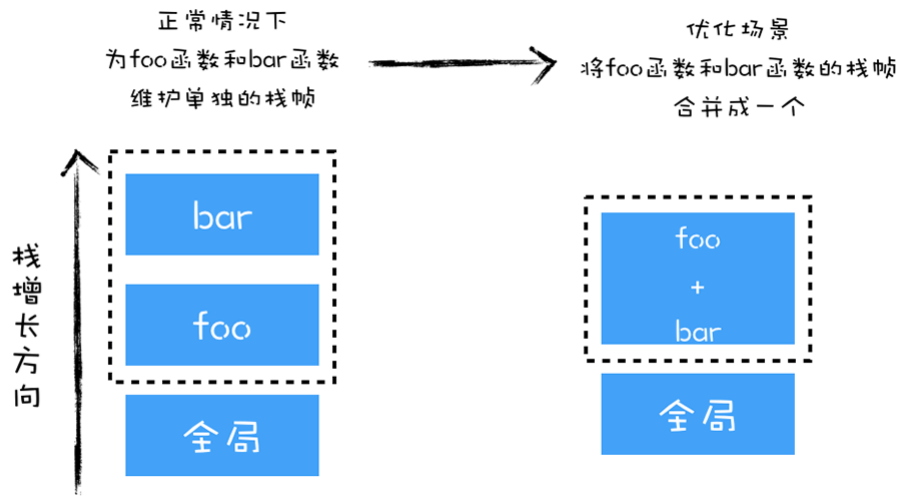
我们看到又出现了一条新的优化信息，新的提示信息如下：

```
<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：

```
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()
```

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：

```
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 /  
[97447:0x179700000000] 34 ms: Scavenge 1.2 (3.4) -> 0.3 (3.6) MB, 0.4 /  
[97447:0x179700000000] 36 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /  
[97447:0x179700000000] 37 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /  
[97447:0x179700000000] 39 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /  
[97447:0x179700000000] 40 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /  
[97447:0x179700000000] 42 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /  
[97447:0x179700000000] 43 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /  
[97447:0x179700000000] 45 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /  
[97447:0x179700000000] 46 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /  
[97447:0x179700000000] 48 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /  
[97447:0x179700000000] 50 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.2 /  
[97447:0x179700000000] 51 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.2 /  
[97447:0x179700000000] 53 ms: Scavenge 1.3 (3.6) -> 0.3 (3.6) MB, 0.1 /  
[97447:0x179700000000] 54 ms: Scavenge 0.8 (3.6) -> 0.3 (3.6) MB, 0.2 /  
eos@libingdeMacBook-Pro 06 %
```

你会看到一堆提示，如下：

```
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
```

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，比如我们可以以换种方式来实现上述代码，如下所示：

```
function strToArray(str, bufferView) {
  let i = 0
  const len = str.length
  for (; i < len; ++i) {
    bufferView[i] = str.charCodeAt(i);
  }
  return bufferView;
}
function foo() {
  let i = 0
  let str = 'test V8 GC'
  let buffer = new ArrayBuffer(str.length * 2)
  let bufferView = new Uint16Array(buffer);
  while (i++ < 1e5) {
    strToArray(str,bufferView);
  }
}
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：

```
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：

```
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：

```
function Foo(property_num,element_num) {
  //添加可索引属性
  for (let i = 0; i < element_num; i++) {
    this[i] = `element${i}`
  }
  //添加常规属性
  for (let i = 0; i < property_num; i++) {
    let ppt = `property${i}`
    this[ppt] = ppt
  }
}
var bar = new Foo(10,10)
console.log(%HasFastProperties(bar));
delete bar.property2
console.log(%HasFastProperties(bar));
```

我们执行下面这个命令：

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

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。