## 关于本档

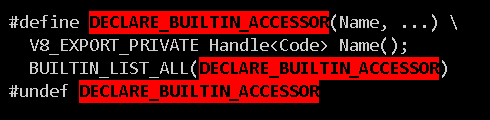
**Author: MG193.7**. 记录本档的初始意图是希望能够在V8引擎内实现字节码层面的调试, 而非局限于JS层的调试. 本档基于在野的缩减Chromium内核miniblink, 即5.7版本的V8引擎.

本档记录于20240221

## Builtin如何将架构下汇编存入Code对象

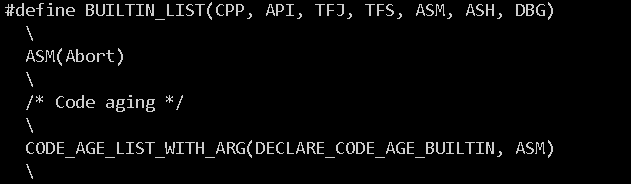
在前一份V8引擎的草稿档中, 我分析过Code对象结构下kHeaderSize偏移处(0x3f)存储着成员function\_data的指针, 此指针指向Builtin在初始化解释器相关内容与结构时提前存储至内存内的一系列汇编函数(指令集).

此处以解释器入口汇编函数InterpreterEntryTrampoline为例. 跟踪此汇编函数, 并分析指令集被写入Code对象的过程.



在builtins.h源文件内, 宏函数DECLARE\_BUILTIN\_ACCESSOR在代码内初始化了一系列类型为Handle<Code>的申明函数, 且该宏函数被交接给另一个宏BUILTIN\_LIST\_ALL.

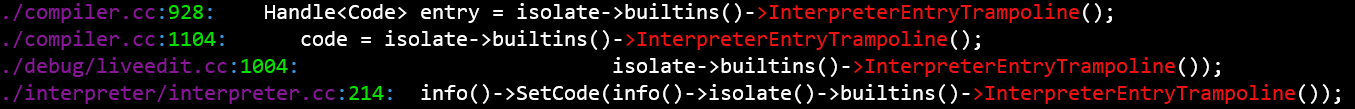




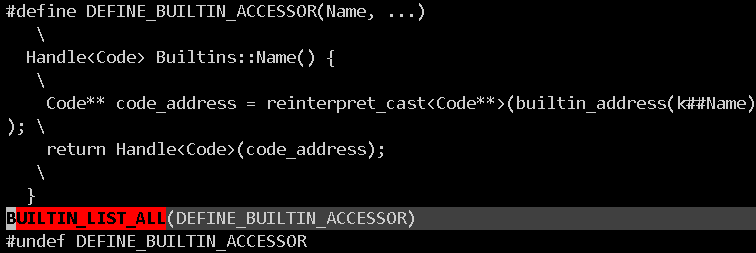
最终上文的宏被交接给BUILTIN\_LIST宏进行处理, 该宏将宏参数当作新的宏函数, 并将此宏内定义的一系列字符传入被作为函数的宏参数.



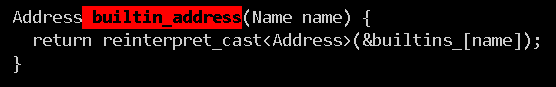
而字符”InterpreterEntryTrampoline”被传入此宏的ASM参数宏函数. 而此时ASM参数宏函数即为前文定义的DECLARE\_BUILTIN\_ACCESSOR宏, 结合具体代码不难得知该宏在builtins.h源文件内申明了一个名为InterpreterEntryTrampoline的Code对象访问句柄.

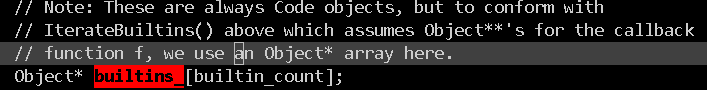


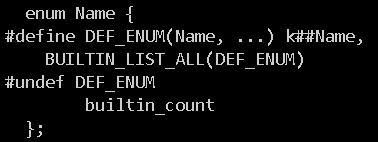
在V8代码内搜索”->InterpreterEntryTrampoline”可以发现一系列此访问句柄的引用, 显然这些代码就是通过builtins.h内宏生成的访问句柄来对解释器入口汇编函数进行访问的. 同样以此函数为例, 下文开始分析该句柄的具体实现.



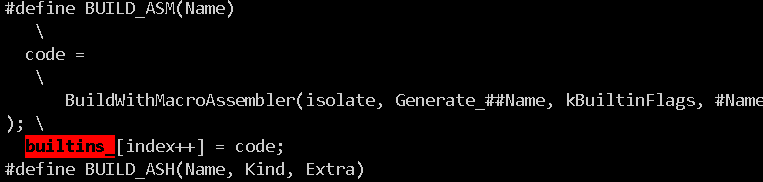
在builtins.cc源文件内, 宏DEFINE\_BUILTIN\_ACCESSOR具体定义了访问句柄的实现, 可见宏向builtin\_address函数内传入k##Name以获取返回数据, 随后通过reinterpret\_cast关键字将数据类型转换为Code\*\*, 最终通过Handle<Code>返回最终数据, 可以猜测Handle或许对数据进行了两次解引用. 而该宏再次被传入BUILTIN\_LIST\_ALL宏内, 结合上文的分析可知, 最终传入该宏的Name参数是一系列由BUILTIN\_LIST宏定义的字符, 再次印证此处确为访问句柄提供了具体实现.





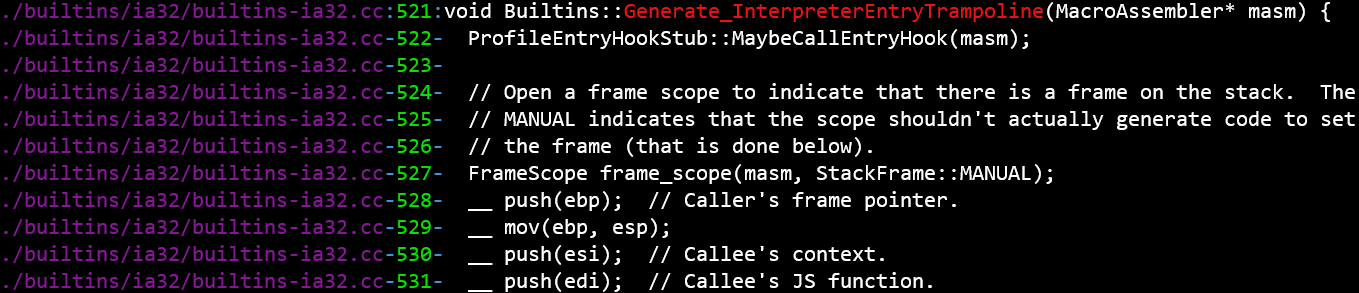


builtins.h源文件定义了函数builtin\_address的实现, 此函数通过传入的Builtins::Name类型的name参数以从名为builtins\_的数组内取出数据, 随后以Handle<Address>形式处理其引用并返回. 跟踪builtins\_成员可知其事实上为一个存储Object\*类型数据的数组, 根据开发者注释可知此类型解引用后就是Code对象的引用. 而在Builtins类声明内, 可以找到名为Name的枚举类型, 此枚举内的数据同样由BUILTIN\_LIST\_ALL宏生成. 结合DEFINE\_BUILTIN\_ACCESSOR宏内访问句柄的实现代码可以猜测, 此枚举内生成的每一个int数据, 都与builtins\_成员内存储的Code对象有着映射关系. 而随后在builtins.cc内的一系列宏内, 我找到了支撑此结论的代码.

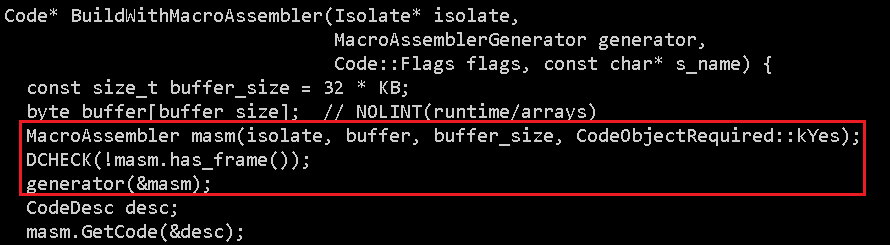




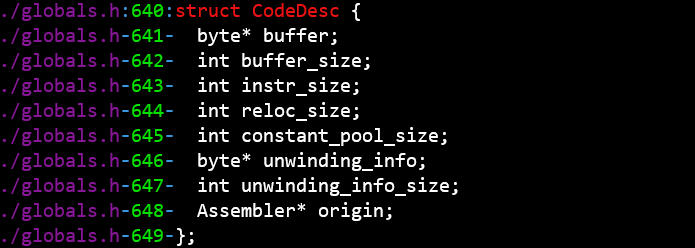
BUILD\_ASM宏内定义了这样的代码, 宏通过BuildWithMacroAssembler函数将获取到的数据转储至code成员, 随后写入builtins\_数组. 根据代码上下文可知该宏由函数Builtins::SetUp定义, code成员即为Code对象指针, 且此宏被作为参数传入BUILTIN\_LIST宏内, 结合分析可知BUILTIN\_LIST宏将会通过定义的一系列字符以及参数宏函数对SetUp函数内定义的宏按序进行不断调用, index在每次调用后均会自增, 显然写入的数据与Builtins类内的枚举数据存在映射关系.

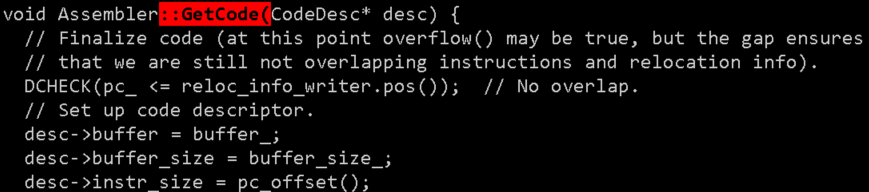


而BuildWithMacroAssembler函数返回的Code对象与宏参数Name有关, 仍然以解释器入口汇编函数为例, 需要注意到宏内传入此函数的Generate\_##Name, 根据该格式, 最终可以在builtins-ia32.cc源文件内找到如下函数Generate\_InterpreterEntryTrampoline, 而此函数就进行了大量架构下的指令操作, 显然这与Code对象内的指令集成员有关, 那么BuildWithMacroAssembler函数自然是分析的重点.

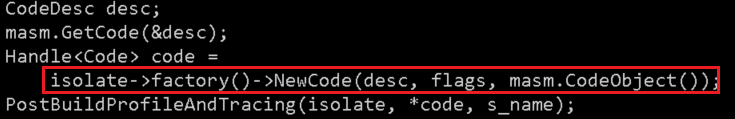


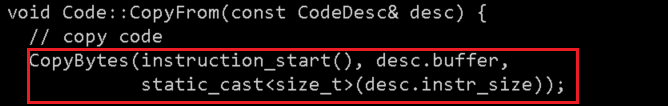
如此函数所示, 程序首先初始化了一个默认32KB的buffer成员, 并将其传入MacroAssembler类的构造函数内以初始化masm成员, 随后, generator参数被作为函数进行调用, 同时传入masm的引用, 若结合上文以及前一份V8分析文档可知, 此处的generator事实上是一个专门通过MacroAssembler操作以及封装架构下指令集的函数, 那么显然Code对象0x3f偏移下的具体内容与generator有关, 但在具体分析generator前, 先继续分析后文操作.



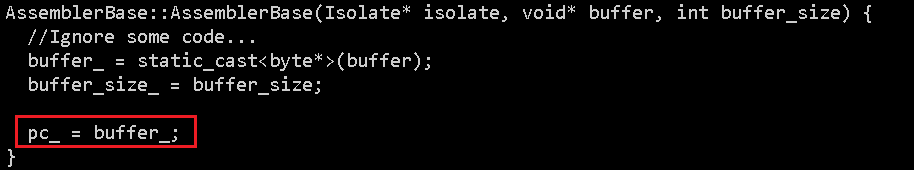


在generator函数执行完毕后的下文中, 代码调用了masm内的GetCode函数, 且传入结构CodeDesc的引用, 结合CodeDesc的具体结构以及GetCode函数的实现不难得知, 函数意在转储masm域内各类成员的信息至CodeDesc结构内, 而其中buffer\_以及instr\_size成员需要特别注意, instr\_size为generator执行完毕后封装的总指令长度.

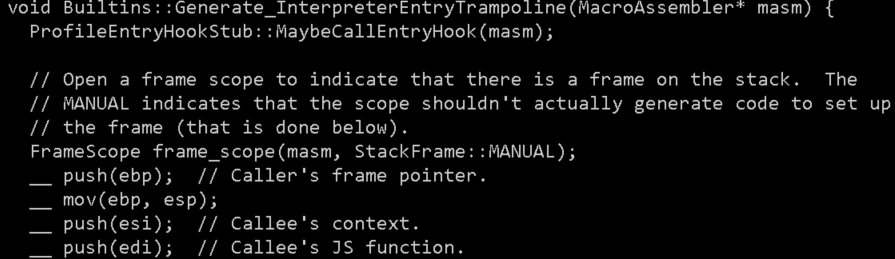


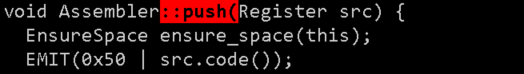


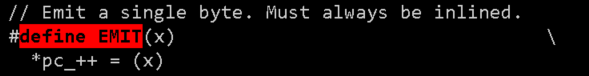
在CodeDesc转储函数执行后, 代码通过Factory::NewCode函数返回了类型为Handle<Code>的Code对象, 前文转储了masm域内数据的CodeDesc被传入该函数, 忽略无关代码, NewCode会将CodeDesc作为名为desc的参数传入Code::CopyFrom函数. 而在此函数内, desc的buffer成员被将按照总的指令长度(instr\_size)拷贝至instruction\_start()函数指定的地址下, 结合前一份V8分析文档可知此地址即为Code基址偏移0x3f处, 换言之, masm域内的buffer\_成员存储着封装完毕的架构对应指令集. 此时, 本档将回到代码上文以分析generator的具体执行.



MacroAssembler继承AssemblerBase, generator被执行前, 空间为32KB的buffer成员被传入MacroAssembler的构造函数内, 那么忽略无关代码, buffer的指针最终将被传入AssemblerBase的构造函数, 可见buffer被转换为byte\*指针类型后转储至masm域内的buffer\_成员, 随后域内pc\_成员再次被赋予buffer指针的值, 这意味着buffer\_不仅用以存储架构对应指令集, 对于pc\_的操作也将影响/改变buffer\_内容.







衔接前文, 以封装函数Generate\_InterpreterEntryTrampoline为例, 此函数内, 代码调用了大量的指令操作函数以操作寄存器或控制流, 通过分析这些指令函数可知, 指令操作的本质实际上就是对pc\_成员的操作, 通过在特定的指令函数内指定或封装架构对应的机器码并填入pc\_成员所指内存中, 封装函数得以向buffer\_成员写入对应指令集, 并操作实际的控制流. 随后, 在封装函数执行完毕后, buffer\_成员的内容将被封装至Code对象偏移0x3f处. 至此, 关于Builtin如何对指令集进行封装的问题已经分析完毕.

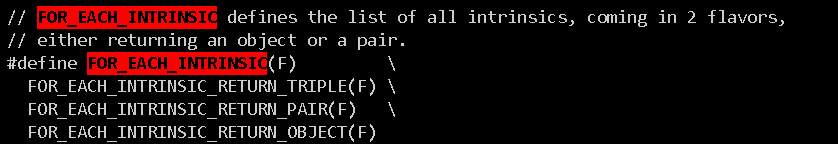
//本章写毕 20240227

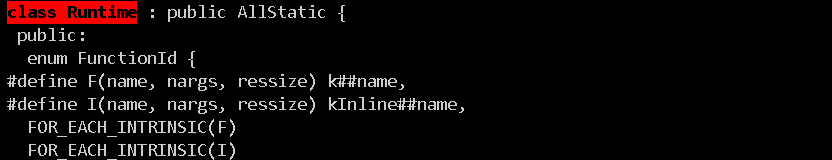
~~//TODO BuildWithMacroAssembler目前暂不分析, 由于分析字节码生成过程的最终目的是为了方便操作dispatch\_table中的字节码句柄指令集以便于调试, 而最简单的调试方案即在字节码内部插入int3(0xcc)指令以通知调试器, 又由于v8自带插入int3的函数, 故此处暂无分析必要, 若有更精细的调试需求时再进行分析. 而关于如何插入int3, 本档下一章皆有释义. 20240223~~

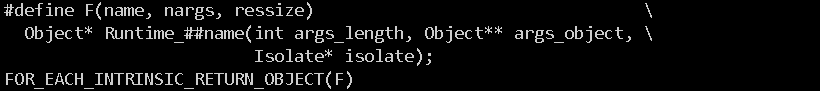
//TODO 目前的需求是如何获取当前指令集在哪个JS函数下运行, 以及当前指令集代表什么字节码, 需要以上信息以进行更细粒的调试.

## TraceBytecode实现分析与调试器开发

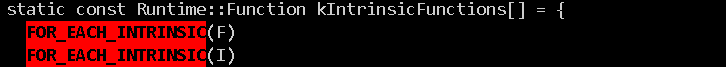
若要利用TraceBytecode进行调试器开发, 则需要对该方法的实现细节进行粗略分析(至少在当前需求下没有必要进行过于详细的分析), 以便仿照V8开发人员的”规范”, 以相对小的代价对V8进行改动和拓展.

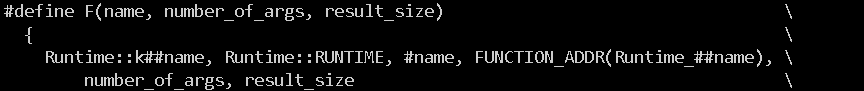






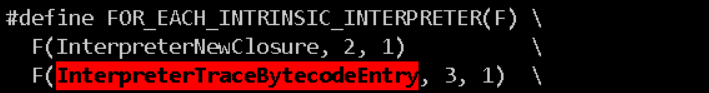
runtime.h源文件内, FOR\_EACH\_INTRINSIC宏内包含了3个”子宏”, Runtime对象利用主宏初始化了大量的FunctionId枚举数据, 同时,代码定义的F宏也利用其子宏对Runtime\_前缀句柄进行批量申明. 显然, 主宏与BYTECODE\_LIST宏的职能类似(预先定义宏常量数据, 以便其它模块利用宏进行批量操作). 关于BYTECODE\_LIST宏的更多细节, 详看上一档对于字节码句柄封装流程的分析.



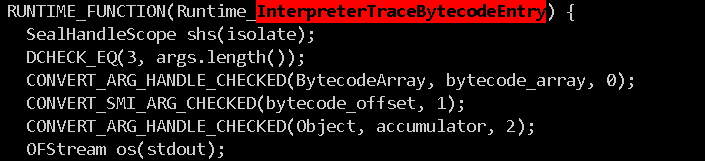


同样在runtime.cc内, 代码利用FOR\_EACH\_INTRINSIC宏批量构造Function结构并存储至kIntrinsicFunctions数组内. 传递至宏内的F宏定义了详细的构造实现, F宏存在3个输入参数, 顾名思义, 这些参数分别对应函数名称, 函数的参数数量以及返回大小. 仔细观察, runtime.h代码内定义的F宏也存在含义类似的输入参数. 而关于kIntrinsicFunctions成员, 其可以间接体现枚举数据FunctionId, Function结构以及Runtime\_句柄间的对应关系: **由于宏函数对于数据的处理顺序固定, 故结构内的FunctionId与Runtime\_句柄相互关联.**

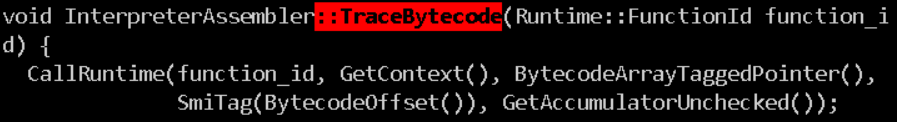


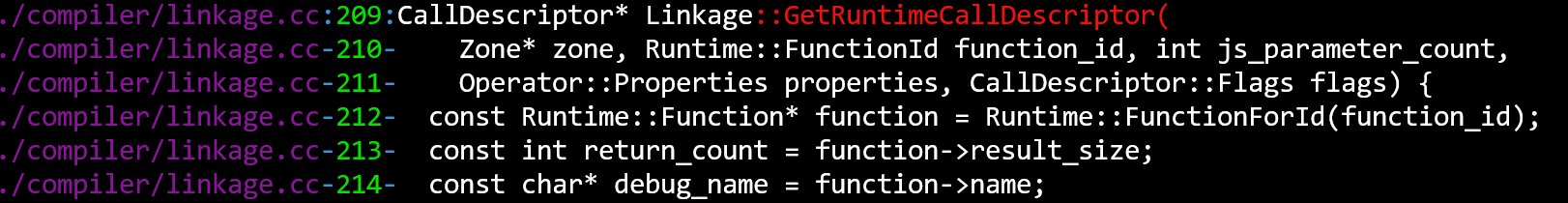


在interpreter-assembler.cc源代码内, 本章欲分析的TraceBytecode方法在对象InterpreterAssembler构造时被调用, 其中, 传递至此方法的FunctionId枚举数据为”kInterpreterTraceBytecodeEntry”, 显然, 这是由宏批量生成的值, 在runtime.h源文件内, 子宏预定义的常量”InterpreterTraceBytecodeEntry”被传递至F宏内, 结合上文分析不难推断出, 该常量拥有对应的FunctionId, 函数句柄, 并且宏为之构造了参数量为3的Function结构. 显然, TraceBytecode的后续关键操作需要使用到该宏常量生成的相关代码, 那么在利用TraceBytecode以进行调试器开发时, 大概率也是需要手动注册宏常量的, 故此, 记录FOR\_EACH\_INTRINSIC宏的相关处理细节与规范, 以便进行后续拓展.



runtime-interperter.cc通过RUNTIME\_FUNCTION宏实现了函数Runtime\_InterpreterTraceBytecodeEntry的具体代码, 显然该函数是runtime.h内利用宏常量申明的Runtime\_前缀句柄.

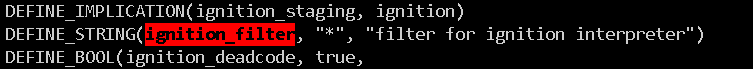




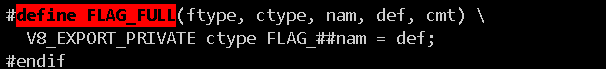
上文, 本档分析了宏函数是如何申明InterpreterTraceBytecodeEntry的函数句柄, 并生成对应的Runtime::Function结构体的. 那么此处开始分析本章的核心函数TraceBytecode的具体实现, 可见代码在InterpreterAssembler域内调用了CallRuntime接口以干预AST树解释为架构下指令集的过程, 当前函数参数所指定的FunctionId被传入此接口, 同时字节码流Tag指针, 字节码偏移以及累加器等关键数据均被传入. 在CallRuntime的后续实现中, 代码在调用函数CallRuntimeCallDescriptor时通过输入的FunctionId在Function结构体内取出了对应的Runtime\_前缀句柄, 显然CallRuntime函数在AST树内增加的调用节点最终指向FunctionId对应的句柄. 那么TraceBytecode的核心实现就是通过干预AST树的节点以直接干预AST树解释为指令集的过程, 最终, 该InterpreterAssembler对象正在解释的字节码在翻译为指令集后, 会通过push指令压入所有关键数据, 随后利用call指令进入Runtime\_函数句柄. 当然, 调试器开发可以利用到该技巧, 以Hook任意字节码.

//Draft! ~~20240307~~ 20240404

## FLAG宏与全局宏常量/调试器开发

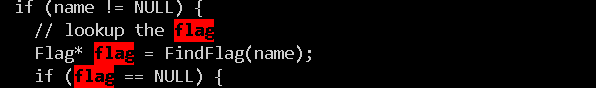




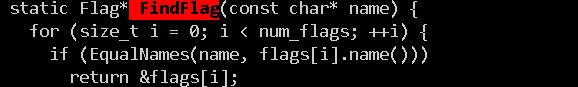


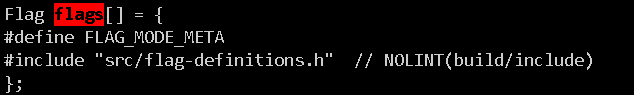
以V8帮助文档内定义的ignition\_filter参数为例, 该参数的传参利用flag-definitions.h源文件内的DEFINE\_STRING宏来实现. 该宏的实现又由FLAG宏的代码所定义. 最终, FLAG宏调用到FLAG\_FULL宏, 此宏内的参数被用以生成一个面向全局且被extern关键词修饰的导出常量, 常量的名称, 类型以及缺省值分别由宏参数nam, ctyle与def定义.

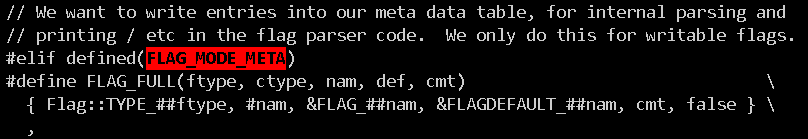




在V8代码内, 程序可以通过直接调用宏生成的全局常量来获取外部传递的参数. 当在命令行内启动D8并携带参数时, argv将被传递至flag.cc源文件内的FlagList::SetFlagsFromCommandLine接口进行处理, 顾名思义, 此函数会将外部参数解析为全局常量.







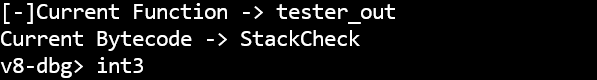
首先, 代码将从argv中提取出参数名, 随后利用FindFlag接口在flags数组内寻找与之相匹配的Flag结构, 一旦匹配则返回该结构的指针. 接着分析Flag结构数组的实现, 可见代码在数组内定义了FLAG\_MODE\_META宏后立即使用include关键字在此处引入flag-definitions.h源文件, 而该头文件的代码在判定FLAG\_MODE\_META宏存在时会重写FLAG\_FULL宏的实现代码, 重写后的代码将根据宏参数构造Flag结构, 故此, flags数组即存放由DEFINE\_系列宏通过预定义常量批量构造的Flag结构. 那么在调试器开发时, 也可利用该系列宏自行定义全局宏常量.

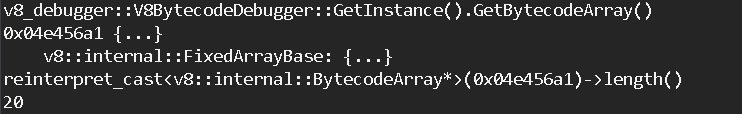
//Draft! 20240312

## 字节码流执行时变动分析

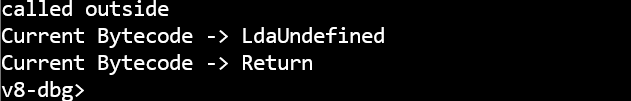
截至目前(20240319), 调试器的大致框架已经完成, 包括任意时机的挂起, 单步, 多步执行, 实时调试真实页面的需求已基本实现(详看调试器开发档), 但当前仍遇到了问题. 在V8调试器对真实页面进行调试时, V8BytecodeDebugger对象作用域下的中断判断函数BytecodeInterrupt出现了运行时异常:

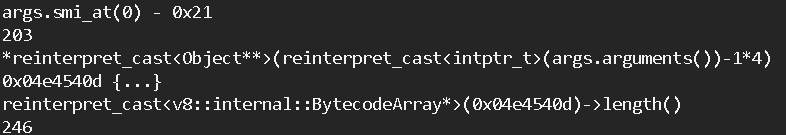
V8调试器尝试利用域内存储的BytecodeArray对象以读取即将执行的字节码时(调用GetFirstBytecodeAddress接口并偏移bytecode\_offset位), 进程被挂起, 并抛出内存越界访问异常. 初步调试可以诊断是由于输入的bytecode\_offset参数远大于当前BytecodeArray对象内的字节码流总长度造成的. 例如, 在调试www.baidu.com时, 随机点击页面将会触发该问题: 某函数的字节码流长度为200左右, 但输入的字节码偏移为7000左右, 从而导致越界访问.



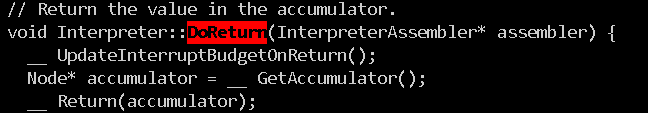


再次调试以尝试复现问题. 在运行时导致越界访问的JS函数内设置中断, 并在V8调试器执行函数的首个字节码StackCheck时利用int3指令触发进程调试器挂起. 此刻尝试访问V8BytecodeDebugger类作用域下存储的BytecodeArray成员指针, 可见其存储的地址为0x04e456a1, 且该字节码流的总长度为20.

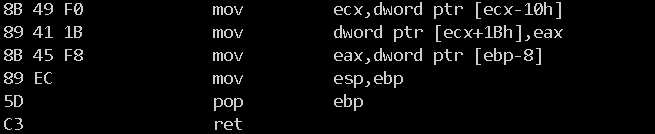




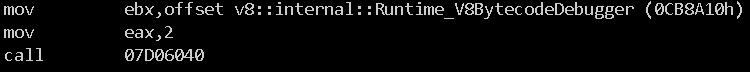
通过continue指令使调试器继续执行字节码流, 随后, 越界访问异常被触发. 由于异常是在中断判断函数内被抛出的, 根据V8调试器开发档与项目代码可知中断判断由宏函数Runtime\_V8BytecodeDebugger发起, 这意味着在触发越界访问前此宏函数一定会被执行, 那么就可以利用进程调试器挂起该宏函数以辅助分析. V8调试器的核心逻辑是Hook解释器的InterpreterAssembler对象, 使得每个字节码句柄在生成时插入外部调用, 并传递关键数据, 以此干预所有字节码句柄指令集的执行流程, 并调节字节码的挂起时机. V8调试器利用该方案向宏函数内传递了即将执行的字节码偏移以及当前字节码流的BytecodeArray指针. 先通过args参数访问字节码偏移, 并减去0x21 (kHeaderSize – 指针Tag长度 = BytecodeArray到首个字节码的偏移)可得出其偏移量为203, 而通过上文可知存储在域内的字节码流长度仅20, 继续访问args内的第二个参数, 可以获取输入的BytecodeArray对象指针, 其地址为0x04e4540d, 显然该地址与域内存储的BytecodeArray指针并不一致, 且输入的字节码流总长度为246. 故此可以进一步明确越界访问的根本原因: 正在执行的字节码流与存储的字节码流存在偏差, 所以bytecode\_offset最终存储的偏移量不可作用于在域内存储的字节码流.

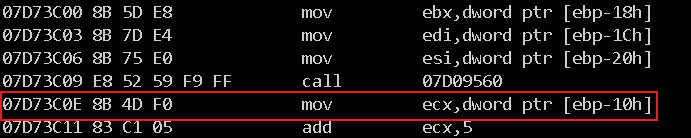


经过多轮的异常复现, 可以发现异常仅出现在Return字节码被执行完毕之时. 通过解释器内的Return句柄可以发现该字节码区别于大多数字节码的执行流程, 根据V8分析档的”Bytecode执行流程分析”章节可知, 字节码句柄会在执行末尾插入尾部调用操作(Tail Call), 并以此定位到下一个字节码句柄入口在封装表内的位置, 而Return字节码句柄并不存在尾部调用操作.



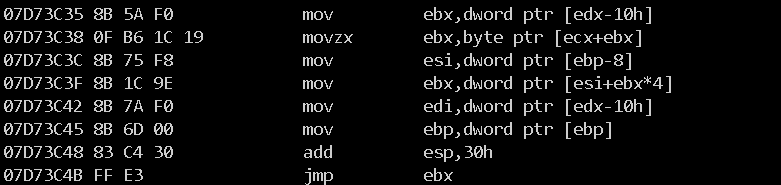
如上为Return字节码句柄指令集末尾的部分指令, 句柄在末尾仅简单保存部分寄存器后就立即恢复栈指针, 随后执行ret(0xC3)指令以回到前一次call指令保存的返回地址. 同样, 参考V8分析档可知大部分句柄指令集的尾部指令一旦在封装表内定位到入口后, 会直接调用jmp指令跳转到目标句柄, 而非使用call指令进入句柄, 那么这就使得当前的栈帧不会发生变化, 换句话来说, 前一次call指令只能由字节码跳板函数InterpreterEntryTrampoline调用.







单步执行Return句柄内的指令至ret以返回至跳板指令集, 接着, 继续在指令集内单步执行至ret以回到跳板指令集的上一个栈帧, 通过观察返回地址处的上下文可以发现, 该指令集首部的部分指令将两个带有关键数据的寄存器推入栈内, 随后利用call指令进入宏函数Runtime\_V8BytecodeDebugger内, 而通过V8调试器代码以及前文分析可知只有被Hook的字节码句柄指令首部才会尝试进入到该宏函数并传递执行相关的参数, 换言之, 跳板指令集的调用者仍是字节码. 继续向相对于返回地址处的低位地址进行审计, 可以发现在0x7D73B40处存在用以初始化栈帧的push与mov指令, 那么该地址显然是此字节码句柄的入口. 在封装表内定位该地址的偏移, 可以推导该指令集即为Call字节码句柄.



单步执行至此字节码句柄指令集的末尾, 可见尾部的指令显然是典型的尾部调用操作: 句柄尝试获取下一个字节码的偏移, 接着在封装表内获取对应的句柄入口并转储至ebx寄存器, 随后利用jmp进入该句柄. 而唯一需要注意的是, 此句柄的执行数据以及隶属的字节码流不同于前文所提的Return字节码句柄, 因为二者处于完全不同的栈帧. 分析至此, 内存越界访问的原因已经很清晰了, Return字节码执行完毕时会销毁当前栈帧并回到上一个栈帧内的字节码句柄, 这将导致当前正在执行的字节码流发生变动, 由于V8调试器考虑到执行效率问题, 不会频繁更新字节码流, 所以实际开发时选用了这样一种方案, Hook跳板指令集的首部指令, 使得字节码流的更新操作跟随JS函数的变动以执行, 这就导致Return字节码执行完毕时, 调试器仍然使用Return字节码所述栈帧的字节码流以读取上一个栈帧内即将执行的字节码偏移, 从而导致越界异常发生.

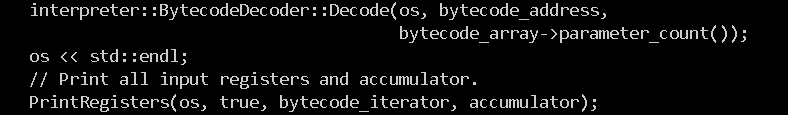
对所有的这些分析进行总结, 无外乎一点: Return字节码会导致正在执行的字节码流出现变动(ReThrow字节码同理). 所以只需要在Return字节码执行之后再次更新传入的字节码流即可.

## 寄存器状态存储与调试

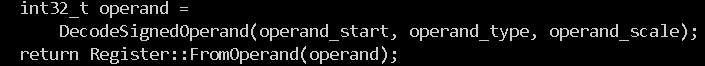
目前的需求是随时/实时获取任意寄存器的内容, 那么就需要知道字节码寄存器的结构, 以及如何对其内容进行提取. 实时显示寄存器的需求可能没有必要, 但实时监控寄存器是Watcher的硬性需求. 20240404

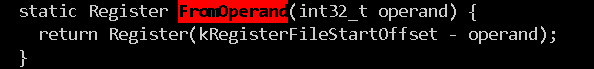
通过分析和分解需求, 在正式进行开发前仍有疑问等待解决:

* 寄存器的数量是恒定的吗? **全局不恒定 20240410**
* 每个栈帧都有自己的寄存器吗? **是 20240410**
* Return/Rethrow导致的栈帧变动是否会导致寄存器也发生变动? **是 20240410**
* 什么时候记录寄存器能使开销尽可能小?
* 需要对每次指令执行时详细的寄存器操作进行记录吗?

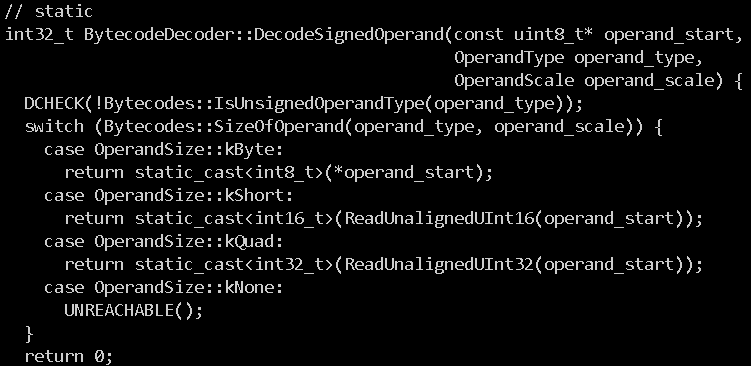


Runtime\_InterpreterTraceBytecodeEntry句柄调用了PrintRegister接口以输出Register的详细信息, 所以关于寄存器调试信息的提取, 可以参考该接口. 而句柄内调用到的BytecodeDecoder::Decode接口是打印字节码以及操作数的具体实现. 在该接口内, 代码通过调用DecodeRegisterOperand以获取当前字节码操作的具体寄存器.

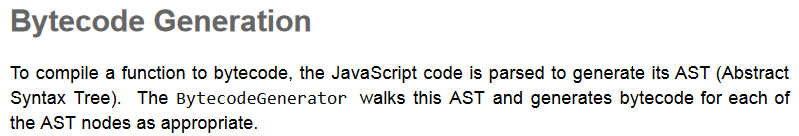




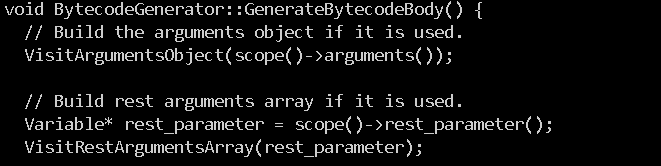
如上为DecodeRegisterOperand接口的具体实现, 其通过另一个接口DecodeSignedOperand获取到int32\_t类型的值operand, 随后利用operand获取特定的寄存器. 由于Register构造函数的携带参数可以指定当前寄存器的索引, 故此FromOperand接口返回的寄存器其索引由(kRegisterFileStartOffset – operand)表达式决定. 而经过调试可知operand一般是负数.

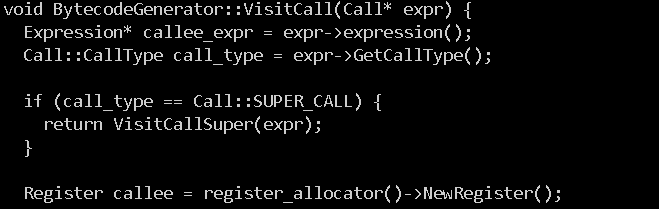


DecodeSignedOperand的实现如上图代码片段所示, 显然, 当前字节码正在操作的寄存器取决于: 操作数地址, 操作数类型以及操作数规模, 通过这些信息可以分析得知, 任何字节码都可以自由指定带有特定索引的寄存器. 但通过switch句柄及其表达式函数SizeOfOperand的输入的参数可知, 操作数类型以及操作数规模仅被解释器用于决定以何种处理方案返回结果, 最终返回的operand仍然由操作数地址指向的具体值决定, 结合上文对于DecodeRegisterOperand函数的分析不难推断出这样的结论: 寄存器的索引由操作数本身直接决定, 而并不由解释器进行分配. 换言之, 仅分析解释器仍然无法对寄存器的规模, 总数进行判断.

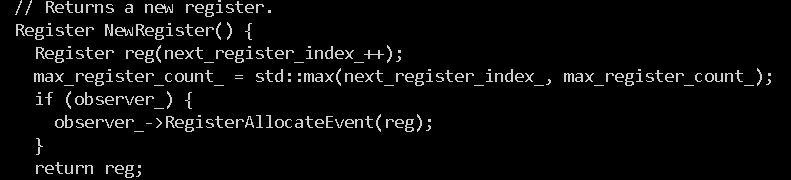


在解释器对字节码流以及其中字节码操作的寄存器进行解释时, 寄存器的索引就已经被决定了, 那么本档就需要对解释器起作用之前更早的时期进行分析. 根据V8 Ignition设计文档所述, V8首先会将函数内的JS代码抽象为AST, 随后在BytecodeGenerator内对AST的所有节点进行分析, 并输出对应的字节码, 直到最后, 这些字节码才会经过解释器进行执行. 而bytecode-generator.cc源文件内的GenerateBytecodeBody函数就负责AST节点分析与字节码生成.





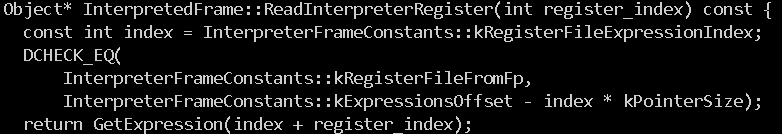
如代码片段所示, GenerateBytecodeBody函数将利用Visit系列句柄访问AST内的每个节点, 并在Visit句柄内根据节点信息利用BytecodeArrayBuilder组件为节点生成对应的字节码. 那么以VisitCall句柄为例, 该句柄在访问节点时会向Builder申请分配临时寄存器以存储被调用者数据. 如代码所示, 句柄通过register\_allocator接口获取寄存器分配器对象指针, 随后利用分配器对象下的NewRegister接口分配新寄存器. 那么接下来的目标就很明确了, 只要能够理清分配寄存器时发生了什么, 就能够知道最多可以分配多少个寄存器.

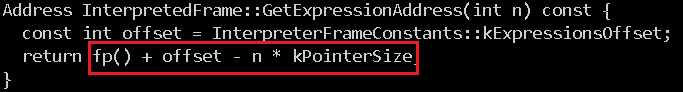


跟踪分配器对象BytecodeRegisterAllocator组件的NewRegister接口, 可见接口代码会在每次执行时将对象内的int类型成员next\_register\_index\_其值增加1, 随后利用该值与另一个int成员max\_register\_count\_进行对比, 取出二者中的最大值, 覆写max\_register\_count\_值, 最后, 利用next\_register\_index\_指定寄存器对象的索引并返回. 通过简单分析不难得知, 分配器每分配一次寄存器时, 分配器内的寄存器索引对象均会增加1, 并且, 随着索引对象值的增加, 对象 max\_register\_count\_也会随之增加, 而跟踪此对象可以得知其记录的正是V8在解释当前JS函数AST时需要使用到的最大寄存器数, 换言之, 每个JS函数所对应字节码流的寄存器总量在解释时是固定的, 编译前是不定的, 但对于全局的调试器开发来说, 字节码调试器不太可能干预编译期并为所有JS栈帧记录最大寄存器数, 这样做的代价与侵入性过大, 所以在全局上来看, 寄存器的数量应被视为不定的.

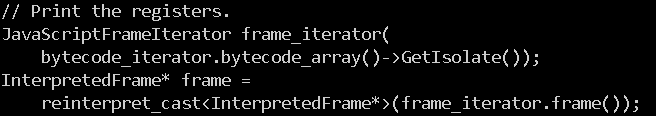
## V8 Ignition帧空间与结构

为什么要研究Ignition的帧结构? 由于解释器为每次字节码流执行而分配的栈帧包含了许多关键信息, 且解释器内大部分与寄存器相关的读写操作均与帧操作相关, 为了支撑寄存器Watcher相关的字节码调试器开发工作, 对Ignition帧结构的分析是有必要的.

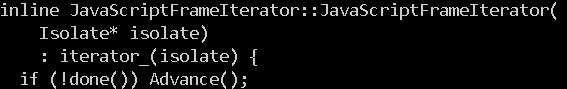


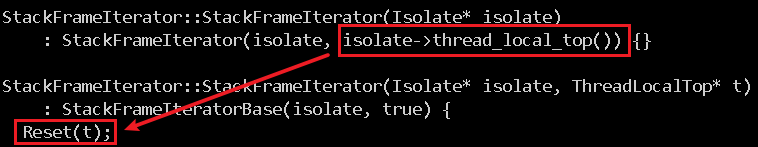


以runtime-interpreter.cc源代码内的PrintRegisters函数为例, 代码利用组件InterpretedFrame内的接口ReadInterpreterRegister对当前函数栈帧内的寄存器进行读取. 跟踪接口的详细实现, 代码将需要读取的寄存器索引与寄存器文件偏移量” kRegisterFileExpressionIndex”(关于栈帧空间内的寄存器文件, 详看Ignition设计文档内的解释器执行章节)相加后传入GetExpression. 最终, 该值被传递至GetExpressionAddress接口, 该接口通过fp()获取了帧指针地址, 接着利用传入值对帧指针进行偏移后返回相应地址. 这意味着函数的帧空间内存储了各个寄存器的详细内容. 换言之, 如果要访问任意函数栈帧内的寄存器, 就必须定位到其帧空间.

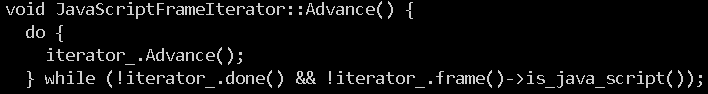


仍以PrintRegisters函数为例, 在代码操作帧空间前, 函数利用JS帧迭代器组件JavaScriptFrameIterator来定位到当前字节码流的对应帧空间. 由代码实现可知定位帧空间需要利用到字节码流对应的Isolate(关于Isolate, 详看[Blink工作原理文档](https://docs.google.com/document/d/1aitSOucL0VHZa9Z2vbRJSyAIsAz24kX8LFByQ5xQnUg)), 随后, 代码利用迭代器的frame函数获取了帧对象指针, 利用该对象即可对帧空间进行访问和操作.

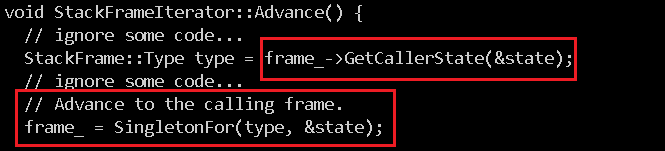




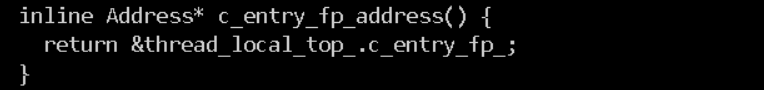
在JS帧迭代器的构造函数内, 代码利用输入的Isolate初始化了域内的另一个迭代器对象StackFrameIterator(本章下文称之为SFI), SFI的构造函数同样使用了Isolate, 并通过其thread\_local\_top接口获取ThreadLocalTop对象(本章下文称之为TLT). 尽管原生V8引擎的开发团队并未对此组件的功能进行详细说明, 但关联代码上下文可以推断, 此组件是一个用以存放当前线程内最顶部栈帧信息的线程局部存储对象. 随后, 局部存储对象被输入到SFI域内的Reset接口. 在该接口中, 代码将通过输入的TLT取出线程最顶部的帧指针, 并使用该帧指针初始化域内帧对象frame\_成员, 其类型为StackFrame. 该部分代码不再赘述.



在此时, SFI的构造函数已经完成执行, 执行权限回到JS帧迭代器域内, 根据前文给出的代码片段, 接下来迭代器将执行域内的Advance函数, 随后, 代码重新回到了SFI域内, 并执行该域的Advance函数. 当执行完毕时, JS帧迭代器将利用SFI#done接口检查SFI是否已经迭代完毕, 若仍然可以继续迭代, 则继续检查SFI域内持有的帧指针是否属于当前正在执行的JS函数, 当结果为True时停止迭代, 此时JS帧迭代器内的帧指针将指向JS函数所属的帧空间.

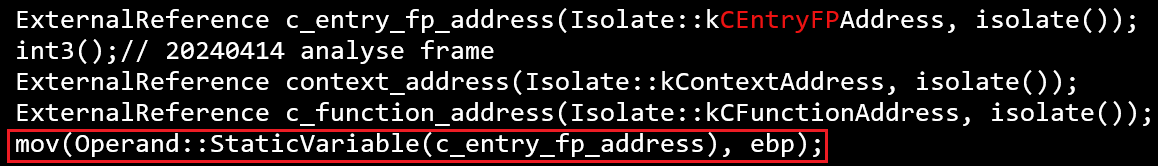


在SFI#Advance函数内, 代码通过当前帧对象的GetCallerState接口以获取当前帧指针所处函数的调用者帧指针地址(即帧指针所指帧空间的上一层帧), 并将调用者帧的相关状态保存至局部变量state中, 随后更新当前帧对象为调用者帧. 接下来, 本档的需求是对帧进行具体的调试, 结合上文对JS帧迭代器以及相关组件的分析可以推断, 如果要逐步跟踪到JS函数的帧空间, 就必须从所有帧的入口, 即TLT组件内记录的当前线程最顶部栈帧数据内的帧空间进行入手.

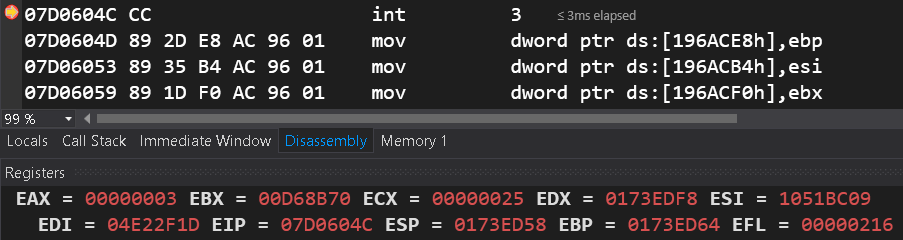




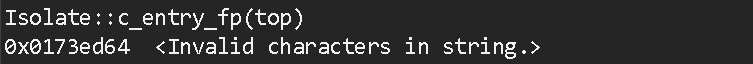
分析TLT组件的结构可知, TLT域内的c\_entry\_fp\_成员即为顶部帧指针的地址, 本档称之为”入口帧”, 而Isolate则提供了c\_entry\_fp\_address接口以访问域内存储的TLT对象内的入口帧地址, 除了该接口, isolate.h源代码文件内的宏函数FOR\_EACH\_ISOLATE\_ADDRESS\_NAME也为该入口帧定义了访问别名. 那么, 通过此接口以及别名, 就可以很容易跟踪到入口帧被初始化的代码.



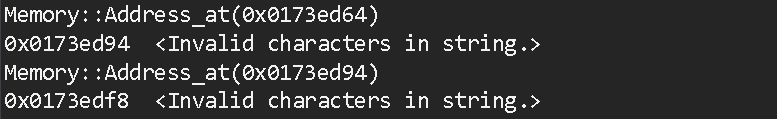
在macro-assembler-ia32.cc源代码文件内, EnterExitFramePrologue函数通过CEntryFP别名对入口帧进行了访问. 该函数的职能与Builtins模块内的部分指令生成句柄类似, 其通过封装有EMIT宏的指令接口操作pc\_指针以直接控制最终buffer\_成员内封装的指令集(此部分详见本档关于Builtins封装Code对象的分析). 在获取入口帧的地址后, 代码立即调用mov指令将ebp寄存器的具体值转储至入口帧地址内, 显然, 根据IA32文档对通用寄存器的描述可知ebp即存储当前指令集的帧指针. 由于V8引擎并不会直接调用该函数, 而是调用此函数输出并存储至内存中的指令集, 故此, 在函数的指令生成过程中插入int3中断指令, 使指令集被执行时将进程挂起

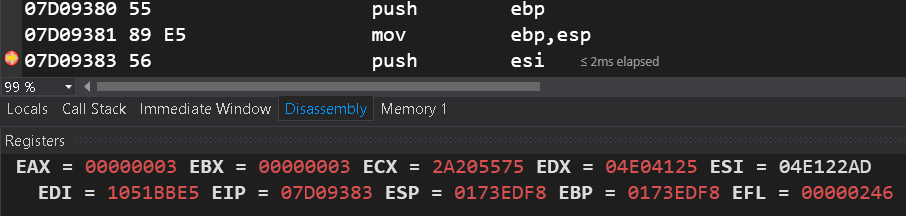


EnterExitFramePrologue

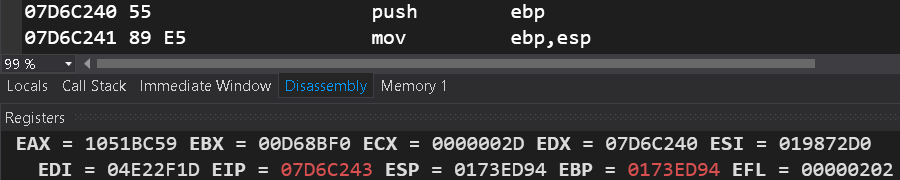


当V8在0xcc处挂起时, 记录此刻ebp寄存器的值. 随后在SFI域内的Reset接口中设置断点, 当代码获取传入的TLT时挂起进程, 可以观察到TLT内储存的入口帧地址与0xcc被触发时记录的ebp寄存器值一致, 这证明了指令集句柄函数EnterExitFramePrologue是入口帧的初始化处, 同时也是当前JS执行线程的最顶部函数. 在上文中, 用以获取当前帧对象上一层帧空间地址的函数接口GetCallerState其实现细节是对当前帧指针进行解引用. 故此, 本档将模拟该过程, 不断对上一层帧空间进行定位.



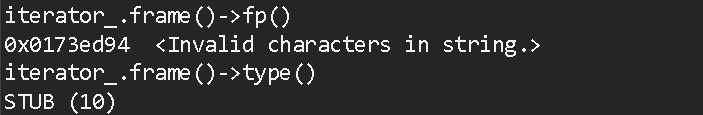


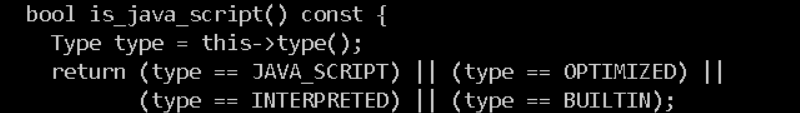
InterpreterEntryTrampoline



Random Bytecode Handler

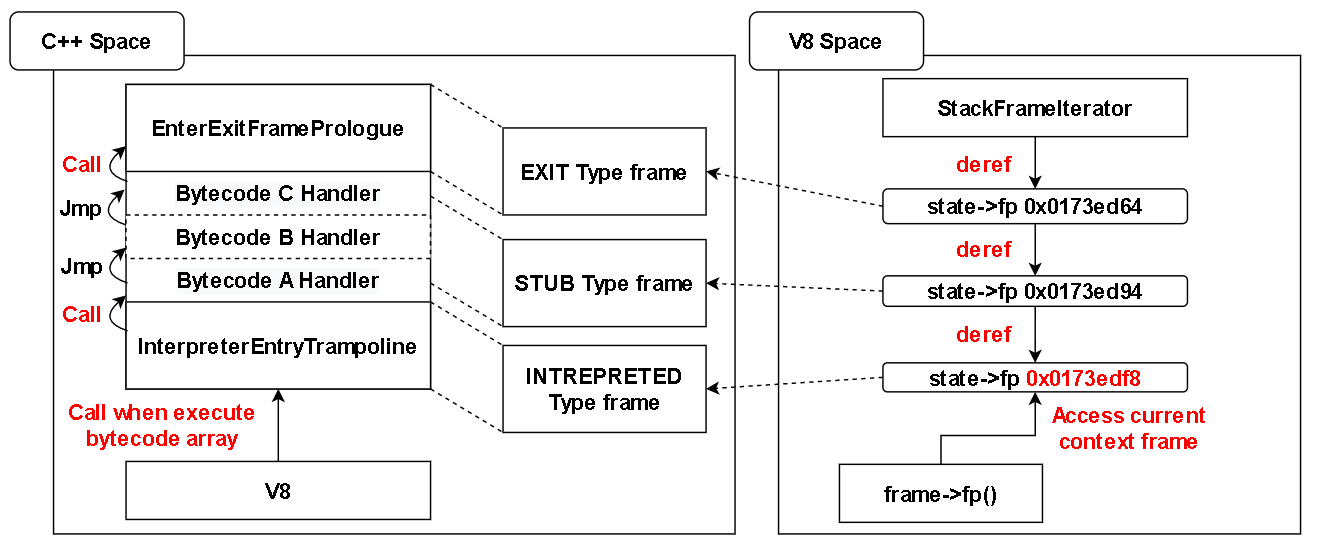
可以观察到, 在对入口帧指针进行两次解引用后, 将得到地址0x0173edf8, 而此值正是字节码跳板指令集InterpreterEntryTrampoline内的帧指针地址(ebp寄存器). 根据前一档针对V8字节码流间执行的分析, 跳板指令集在接近末尾的数条指令处, 会利用call指令进入字节码流内的首个字节码句柄指令集内, 该操作会开辟新的栈帧空间, 那么0x0173edf8帧的下一层帧空间即为针对入口帧倒数第一次解引用时的地址0x0173ed94, 在任意字节码句柄指令集首部开辟栈帧后挂起, 可以发现此刻ebp寄存器其值正是0x0173ed94, 换言之, 该值也是字节码流执行空间内对应的帧指针地址.







此时回到上文针对JS帧迭代器构造函数的分析, 代码将在构造函数内不断迭代SFI所指向的帧空间, 只有当帧指针属于正在执行的JS函数时, 迭代才会停止, 那么在JS帧迭代器内进行调试, 当SFI内的帧对象迭代至指向字节码流执行空间时挂起, 却发现该帧空间却为STUB类型, 迭代操作仍将继续, 这证明字节码句柄的帧空间内不会包含寄存器信息. 而观察负责判断帧空间类型的实现代码is\_java\_script接口可以得知, 解释器所属的帧空间, 即INTERPRETED类型也被认为是属于JS函数的帧空间类型, 那么解释器空间内的帧指针0x0173edf8是否就属于SFI最终所指向的帧空间? 答案是肯定的, 在函数PrintRegisters其代码获取JS帧迭代器的最终帧对象时挂起进程, 并在此刻访问帧对象内存储的帧指针, 可见其指针地址正是0x0173edf8, 利用该指针, V8可以对帧空间内存储的所有寄存器进行操作.



上图为JS帧迭代器访问当前上下文帧空间指针的大致流程, 在每次执行的字节码操作寄存器时, JS帧迭代器将从入口帧开始解引用并以此访问所有帧空间来判断帧所属类型, 显然, 使用迭代器来监听所有寄存器的效率相对较低. 故此, 在进行寄存器相关的调试器开发工作时, 可以通过Hook字节码跳板指令集以获取解释器空间内的帧指针, 并以此直接在帧空间内操作寄存器, 避免在后续的Tracer或Watcher开发工作中因为频繁使用JS帧迭代器访问所有帧空间而导致的低效操作.