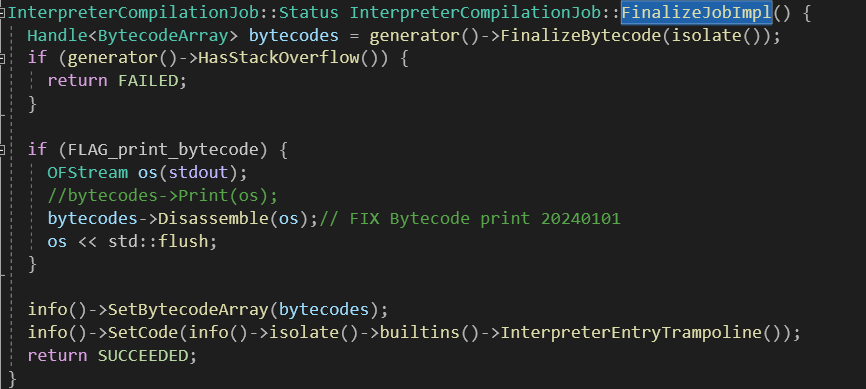


宏函数生成的set\_function\_data 用以设置bytecodearray

~~暂时不关心bytecode是如何装载的, 目前只分析bytecode是怎么被v8执行的~~

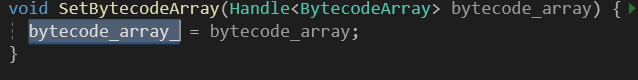
//20240207 **Author: MG193.7**

## 最开始是如何跟入到set\_function\_data处的?



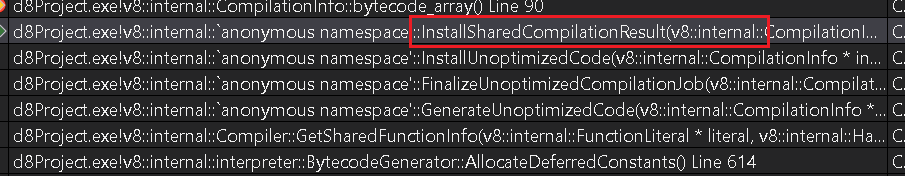
Bytecode解释模块interpreter.cc# FinalizeJobImpl函数内, bytecodes->Disassemble(os)将在标准输出流内打印bytecode的字节码解释信息, 而模块在解释完bytecode后必然要重新取出bytecode进行执行.

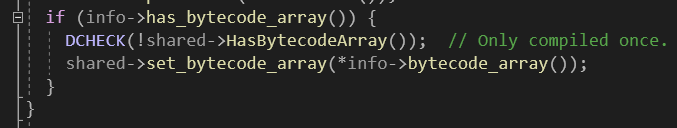
在取出bytecode前, info()->SetBytecodeArray函数对bytecode进行了存储.





对此函数进行跟踪, 可以发现最终bytecode\_array()函数负责取出该值, 对此函数下断点.

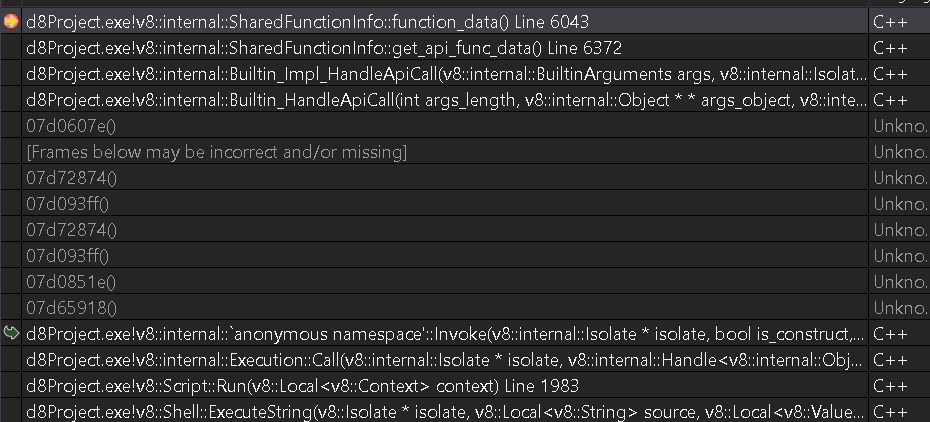


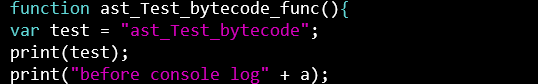


在触发断点后会发现interpreter.cc在解释完毕字节码后调用到InstallSharedCompilationResult函数, 该函数调用SharedFunctionInfo::set\_bytecode\_array将取出的字节码转储至SharedFunctionInfo对象内, 而需要注意的是set\_bytecode\_array最终会进入到名为set\_function\_data的宏函数, 由ACCESSORS宏进行定义.

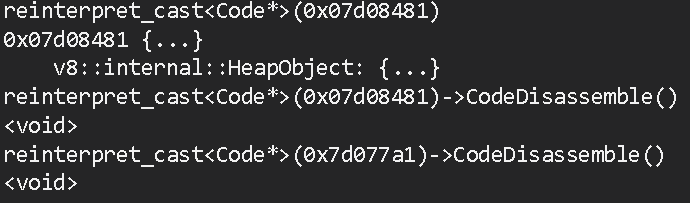
此宏函数会根据偏移存储输入值到目标堆的相应偏移内.

## 一些调试技巧





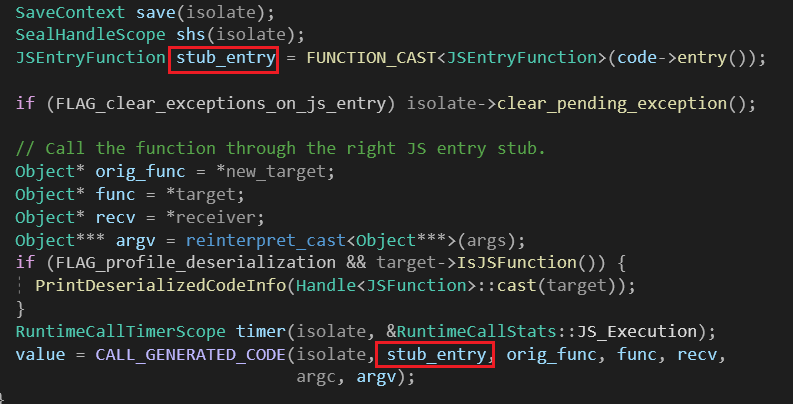
在function\_data()宏生成函数处断点, 随后执行JS文件, 以上为调用栈

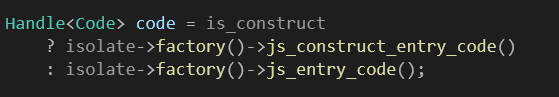


如调用栈中不带符号的帧, 即为builtin初始化时的内置汇编函数. 另外是一个调试技巧, 在immediate window内可以将汇编函数开头的地址减去0x3f后转换为Code指针以便进行其它操作. 至于为什么减去0x3f, 是因为此处内置汇编函数本质上为Code对象内的function\_data, 而根据builtin中对内置汇编逻辑以及ACCESSOR宏函数的分析可知, 若要访问到function\_data则需要将Code对象所在地址偏移kHeaderSize – TAGSize=0x3f位(Code对象本质上继承堆对象, builtin通过偏移来存储内容), 所以function\_data反向偏移0x3f即为Code对象的开始地址, 所以将此地址通过**reinterpret\_cast<Code\*>(address)**解释指针为对象即可拿到Code.

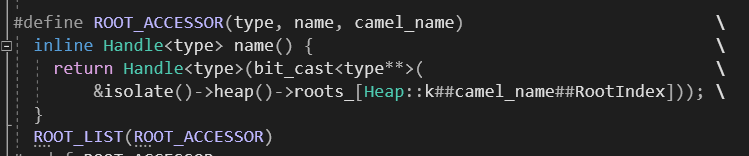
## execution.cc#Invoke函数分析

execution.cc#Invoke几乎是JS层触发函数调用的直接入口,故对此函数的分析是有必要的,这将有利于后续字节码层面函数间调用以及整体执行流程的分析.

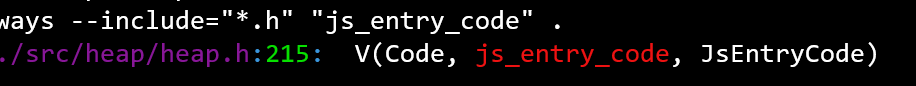




Invoke函数会将局部变量stub\_entry传入宏函数CALL\_GENERATED\_CODE内, 此宏函数会将第二个参数(即stub\_entry)当作函数进行调用, 显然stub\_entry为cpu对应架构的指令集入口, 如果要分析此指令集内部实现, 则需要分析stub\_entry的构造过程.

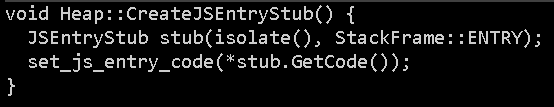


显然stub\_entry的构造由变量code(类型为Code对象)决定, 向上寻找code引用, 此处以js\_entry\_code函数为例, 显然该函数将返回Code对象. 此函数为factory.h#ROOT\_ACCESSOR宏函数在编译期展开而来.

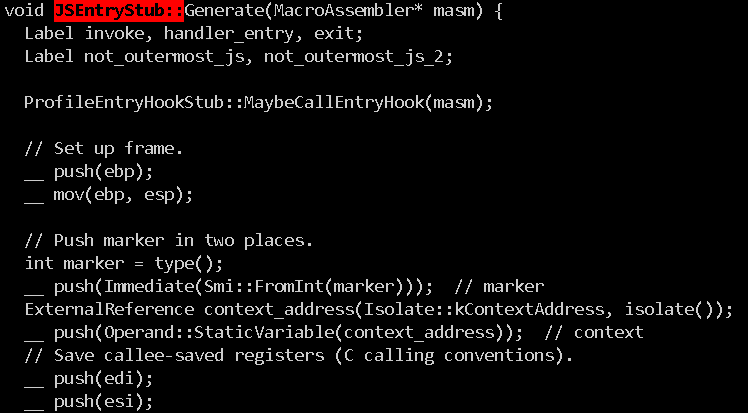


经过搜索可知heap.h#STRONG\_ROOT\_LIST宏将被ROOT\_LIST调用, 并且将ROOT\_ACCESSOR作为宏函数内的参数V, 换言之此处即为js\_entry\_code函数的初始化操作.

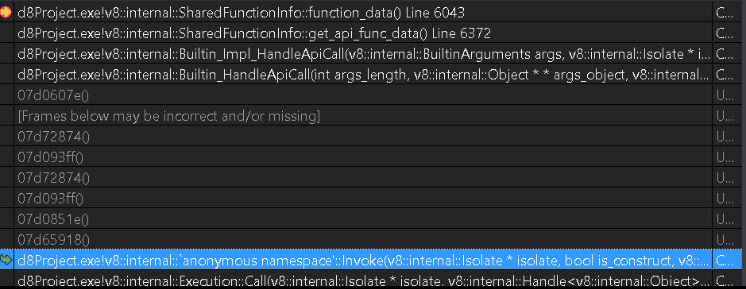


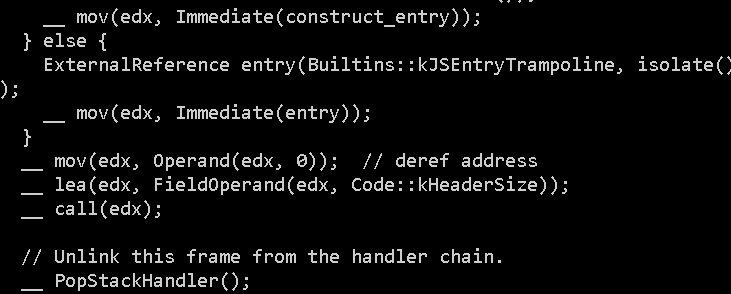


同样经过搜索, js\_entry\_code不仅有访问宏函数, 还有set宏函数. 显然js\_entry\_code的内容由JSEntryStub对象内的GetCode决定.

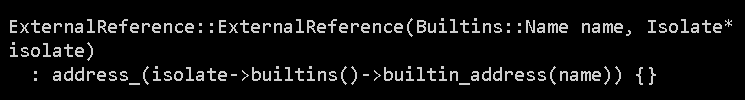


而通过分析JSEntryStub对象可知, GetCode最终返回的内容由code-stub-ia32.cc内的JSEntryStub::Generate函数决定, 该函数会向Code对象内存入预先定义的汇编指令, 随后返回.



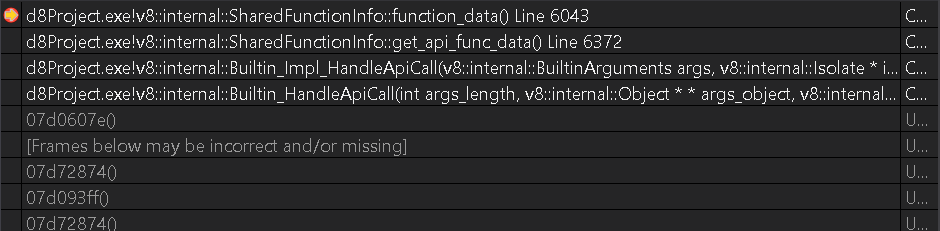


如图, 在Generate函数内, JSEntryStub内置的汇编指令会实例化ExternalReference对象, 并将该对象写入edx寄存器, 随后将edx解引用取出其address\_成员所指地址后偏移kHeaderSize位, 并重新写入edx, 随后调用call指令执行edx地址下的指令集, 显然edx最初是一个Code对象, Code对象偏移kHeaderSize位后就是function\_data所在位置.

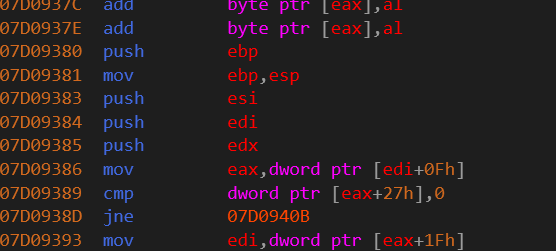


ExternalReference对象构造时会根据传入的Builtins::Name对象决定解引用后的Code地址.

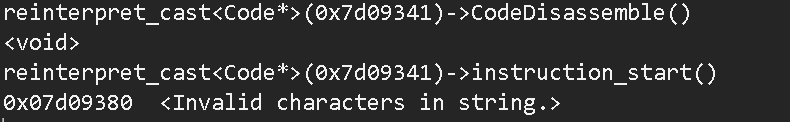
## Bytecode-Level函数间调用执行分析

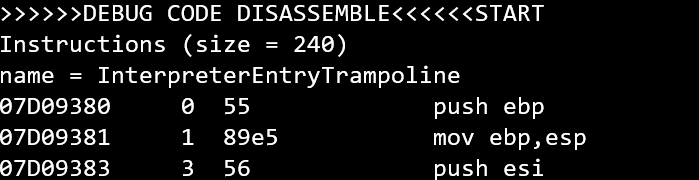


如图, 函数在取出Code内的function\_data后bytecode就已经在执行中了. 以此处CallStack为例, 无符号帧为Builtins初始化时填充至对应Code对象内的IA32架构指令集. 在栈顶function\_data()宏函数处不断步出(step out), 会发现当步出至0x07d093ff地址处时, 再次执行步出操作将立即触发function\_data()宏函数断点, 这意味着程序在0x07d093ff处尝试执行至ret时触发了取function\_data操作, 这表示很有可能下一个JS字节码被执行了, 所以才需要通过取function\_data来执行下一个字节码句柄.

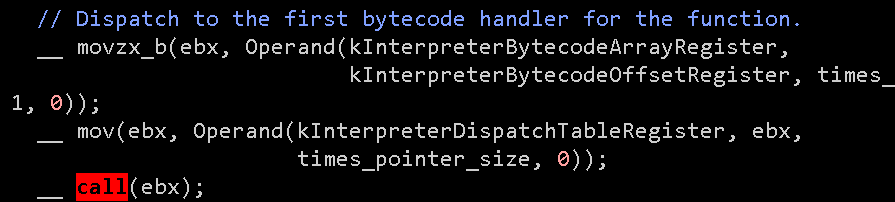


跟入0x07d093ff处指令集, 并不断向上寻找push指令, 最终在0x07d09380处发现push ebp指令, 随后程序立即执行mov ebp,esp以初始化栈指针, 显然这是指令集函数的开头, 记录该指令集并反向偏移0x3f位以定位至Code对象开头.





通过修改d8的部分代码, 可以在程序运行时打印Code对象0x3f偏移处(function\_data)的汇编形式, 根据指令集名称可知其为Buitlins初始化的InterpreterEntryTrampoline函数, 跟入builtins-ia32.cc即可定位至对应的指令集初始化方法.

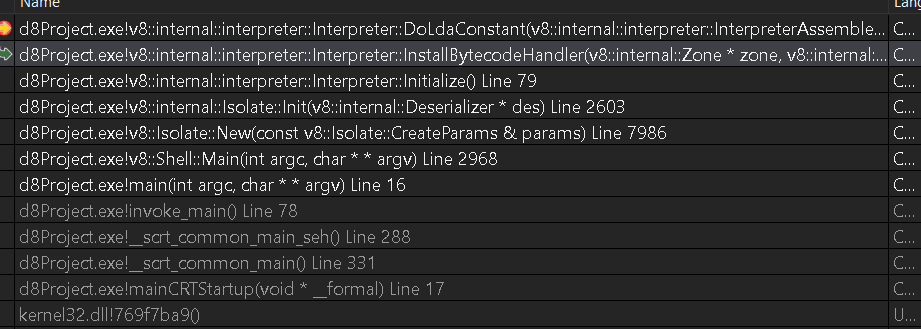


在Generate\_InterpreterEntryTrampoline初始化函数内, call(ebx)操作即为前文0x07d093ff处的上一个指令(即返回地址). 简单分析后可知BytecodeArray寄存器存储当前执行的JS的字节码集, DispatchTable寄存器按序存储JS字节码对应的处理句柄(关于DispatchTable, 请查看Ignition设计文档), 而BytecodeOffset寄存器**此时**则存储首个字节码的偏移量, 在call指令的上文中, 程序首先将(BytecodeArray + BytecodeOffset \* 1) + 0后的地址写入ebx寄存器, 此时ebx寄存器存储的是即将执行的JS函数的第一个字节码地址, 随后程序将(DispatchTable + ebx \* pointer\_size) + 0后的地址重新写入ebx, 这意味着ebx此时存储着首个JS字节码对应的处理句柄, 接着call指令被触发. 显然其句柄将被执行, 这也是当前指令集在执行至ret前就触发function\_data()宏函数断点的原因: **此封装函数将会触发并执行另一个JS函数**.

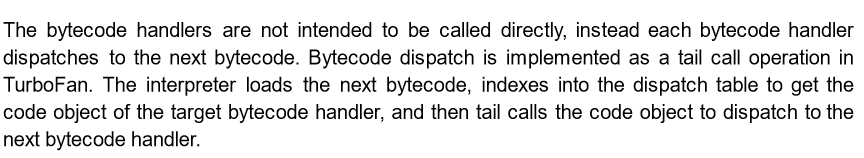
**~~//TODO~~** ~~此处分析可能有误, InterpreterEntryTrampoline封装函数内DispatchTable相关的逻辑可能与单条Bytecode执行无关.~~

**// 分析已修正 20240229**

## Bytecode句柄封装分析

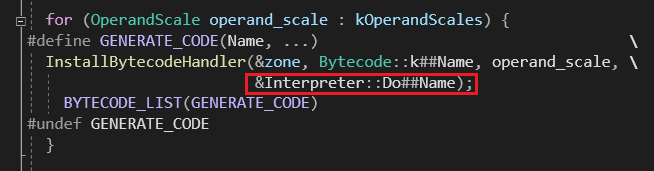


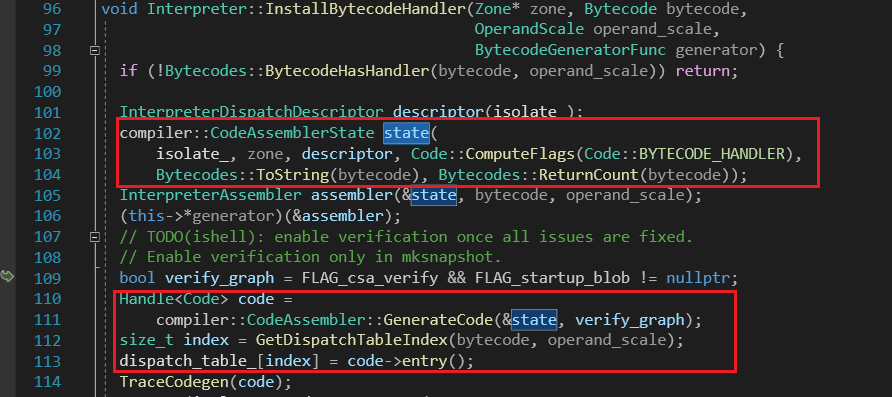
在DoLdaConstant函数处下断点, 接着运行V8以执行JS文件, 进程随后被调试器挂起. 而DoLdaConstant函数并非在字节码被执行时才被触发, 而是在较早的时期被调用, 这是因为该函数实际上为字节码句柄的封装函数, 早在Ignition解释字节码之前的编译期间, 该封装函数就会被调用, 且生成相应的Code对象以存储在DispatchTable内. 关于此部分, 可以查阅V8设计文档的[Generation of Bytecode Handlers](https://docs.google.com/document/d/11T2CRex9hXxoJwbYqVQ32yIPMh0uouUZLdyrtmMoL44/edit#heading=h.ualrlzgsdtv7)章节



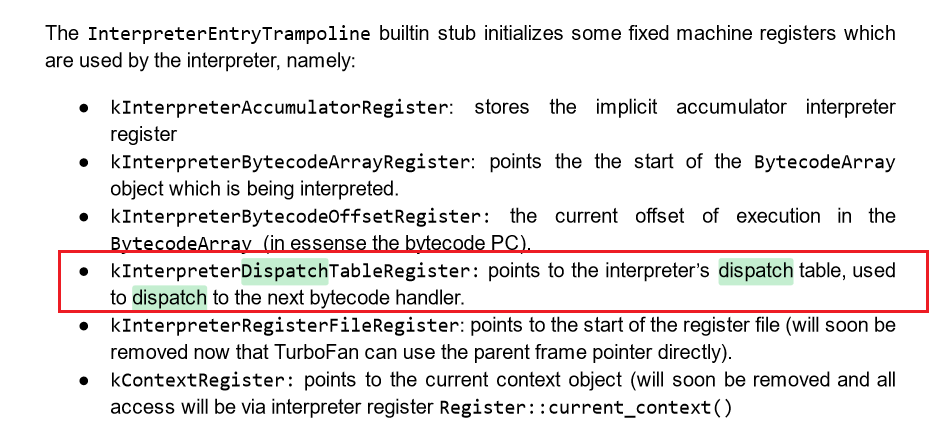
如文档所述的相关细节, 字节码句柄并非被直接执行, 而是被封装为指令集(详见下图中提到的BYTECODE\_LIST宏函数)后依次存入DispatchTable中, 字节码执行时, 编译前期Builtin内封装函数生成的指令集会映射当前字节码至DispatchTable中存储的对应字节码句柄. 而关于如何映射和执行, 该部分将在下一章节进行分析.

// **本章档案细节补充 20240229**





这便是为什么LdaConstant以及其它的字节码处理句柄仅执行一次后便不再执行的原因, 如上图interpreter.cc所示代码片段, Interpreter::Initialize函数内定义的 GENERATE\_CODE宏利用BYTECODE\_LIST宏函数将源代码内定义的每个字节码依次传入InstallBytecodeHandler函数进行处理, 该函数则利用字节码对应的句柄封装函数封装架构指令集, 并将指令集的入口地址按序存入dispatch\_table\_

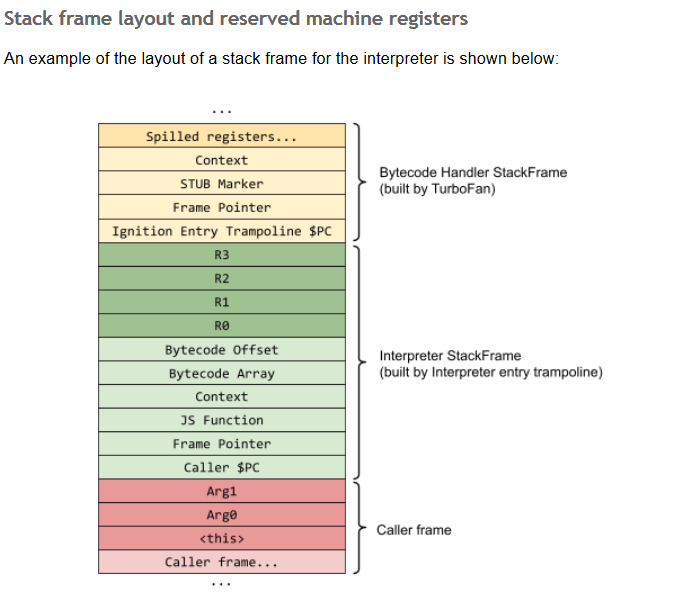


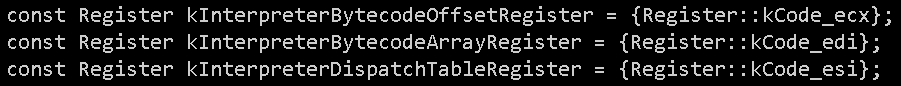
~~dispatch\_table\_用以按序存储字节码处理句柄, 相关信息在V8 Ignition官方设计文档内的~~[~~Interpreter Code Execution~~](https://docs.google.com/document/d/11T2CRex9hXxoJwbYqVQ32yIPMh0uouUZLdyrtmMoL44/edit#heading=h.7b9wptqehp6i)~~章节均有说明. 而dispatch\_table在Builtin的InterpreterEntryTrampoline初始化函数内被预先处理, 相关结构可查询设计文档.~~

**// 沉杂分析, 可以无视. 20240229**

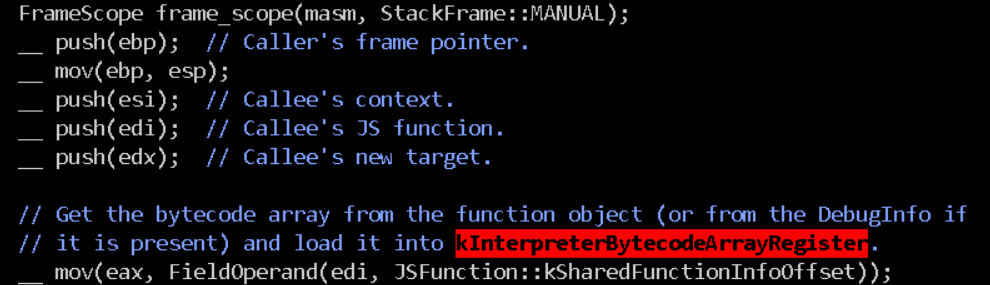
## ­­Bytecode执行流程分析

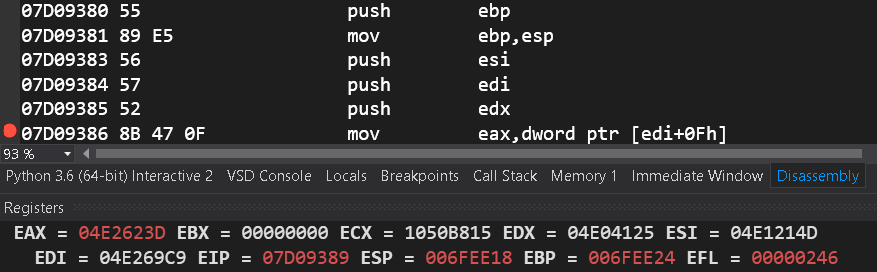
前文, V8设计文档内的Generation of Bytecode Handlers章节简要描述了字节码执行的关键细节. 字节码的句柄并非被设计为直接调用的形式, 而是在早期就被编译为架构下的指令集并存储至diapatch\_table中(详见上文分析). 在编译前期, 解释器会在字节码句柄生成指令集时向末尾添加尾部调用操作(Tail Calls), 该操作生成的指令集将允许字节码在dispatch\_table中定位至下一个字节码的指令集入口地址, 并以此形式执行字节码流中的所有指令. 同时在上文, 本档分析过Builtin封装函数InterpreterEntryTrampoline的部分运行流程: 当封装函数执行call(ebx)指令时, JS函数的首个字节码句柄指令集将被执行. 换言之, 如果要分析完整的字节码执行流程, 就必须完整分析该封装函数与尾部调用的指令定位操作.





如V8设计文档所示, InterpreterEntryTrampoline将初始化特殊的栈帧, 该栈包含一系列用以指令定位, 偏移定位或存储关键信息的固定机器寄存器. 其中, 以IA32架构CPU为例, 当前字节码偏移寄存器(BytecodeOffset)被定义为ecx, 字节码流寄存器(BytecodeArray)被定义为edi, 字节码封装表寄存器则被定义为esi寄存器.



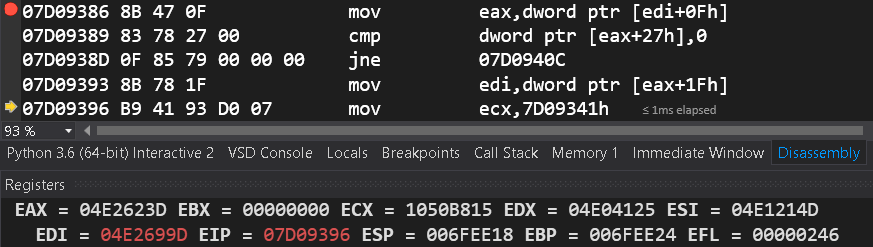




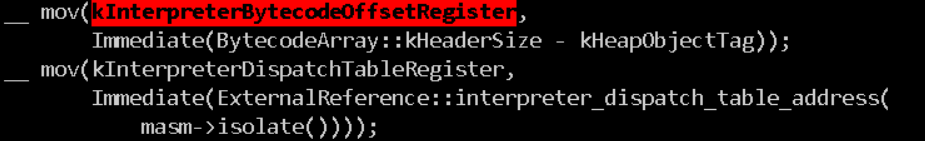


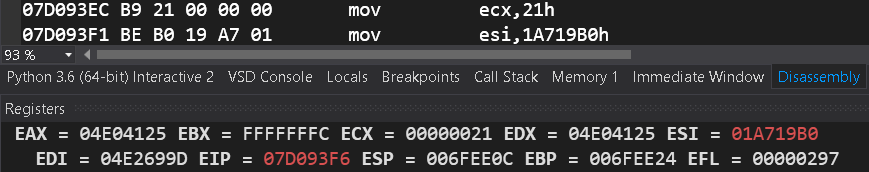
封装函数开头, 程序在栈初始化完毕后立即将edi寄存器偏移0xF位后的内容转储至eax寄存器. 此处, 源码定义edi寄存器指向JSFunction对象, 而JSFunction::kSharedFunctionInfoOffset其偏移值为0x10, 由于V8使用了标记指针技术, 那么0xF显然就是0x10 - 0x1运算后的值, 故此, 最终转储至eax寄存器的内容(04E2623D)即指向SharedFunctionInfo对象的指针.



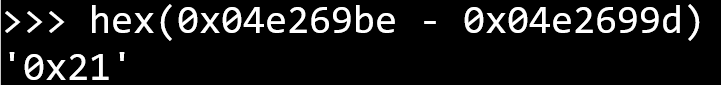


随后, 指令立即将eax寄存器偏移0x1F位(kFunctionDataOffset – 0x1)以获取BytecodeArray对象的指针(04E2699D), 并转储至edi寄存器内. 注意, edi寄存器此时也承担上文所提的字节码流寄存器的职能. 接着, edi在指令下文被推入栈中, 该部分不再赘述.





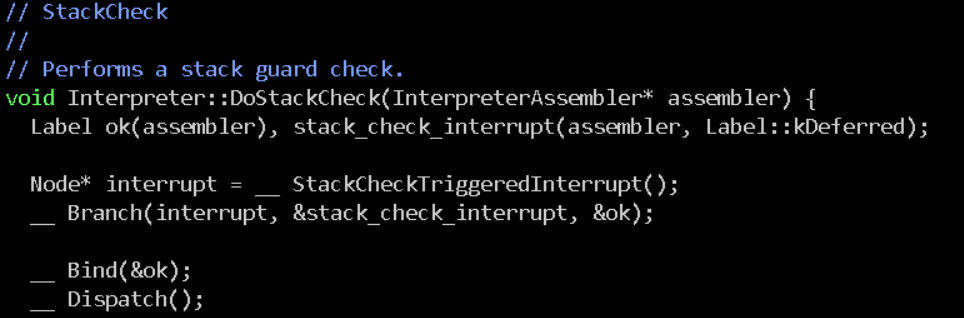


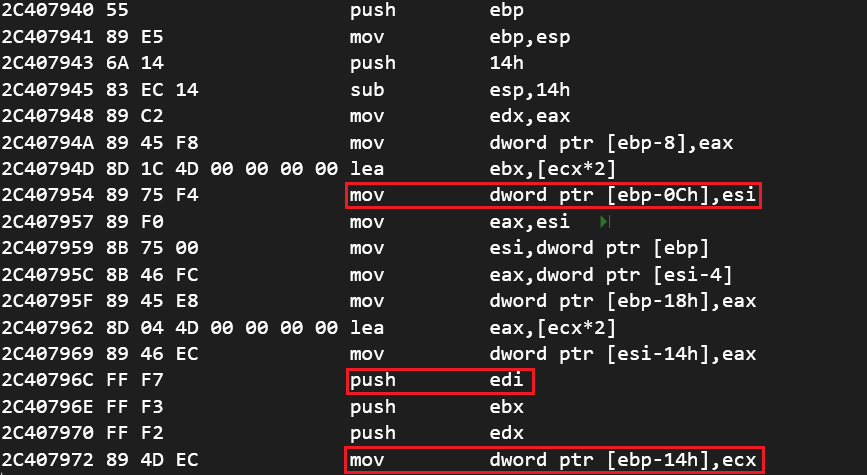


忽略无关指令继续向下执行, 指令将0x21与0x01A719B0分别转储至ecx与esi寄存器, 根据封装函数源码可知此时ecx承担当前字节码偏移寄存器的职能, 而esi则承担字节码封装表寄存器的职能. 换言之, 0x21即为此刻的字节码偏移. 利用上文获取的edi寄存器内的对象指针, 并将其转换为BytecodeArray对象指针以获取当前字节码流内的首个字节码地址, 显然该地址与对象指针地址的偏移量为0x21, 这也呼应了本档之前章节中对该封装函数的分析: 封装函数将执行首个字节码的句柄指令集.

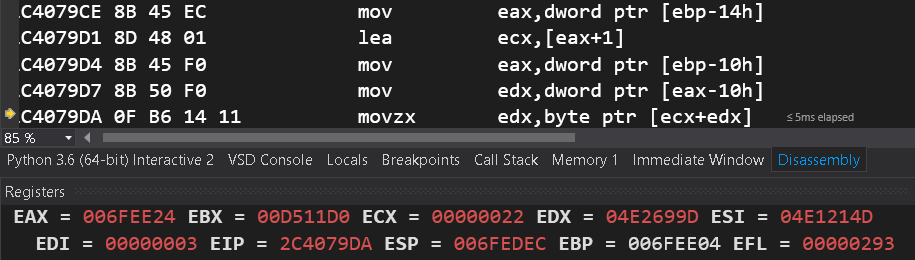


与前面章节的分析一致, 指令通过ecx寄存器获取字节码流中的字节码, 随后利用esi寄存器引索至封装表中的指定句柄, 随后立即调用call函数进入句柄指令集, 不再赘述.­ 由于0x04e269be处存储的第首个字节码0x78为StackCheck指令, 故下文将分析Interpreter::DoStackCheck封装函数内的句柄指令集.

0x78字节码以及其它字节码句柄的封装函数不同于Builtin初始化时使用到的部分封装函数, 字节码句柄在封装为指令集时并未使用mov, push等通过EMIT宏函数直接操作pc\_与buffer\_成员的指令函数(关于更详细的封装细节, 请查看下一档关于Bytecode封装为Code的分析), 而是直接使用了可被最终转换为指令集的功能封装函数, 故直接分析句柄封装函数并不直观. 由于本档章节仅关注字节码指令如何引索至下一个字节码句柄在封装表中的偏移, 而Dispatch函数就实现了这一套指令集(换言之, Dispatch函数即上文所述的尾部调用操作), 故此, 下文将只分析0x78字节码的句柄指令集.



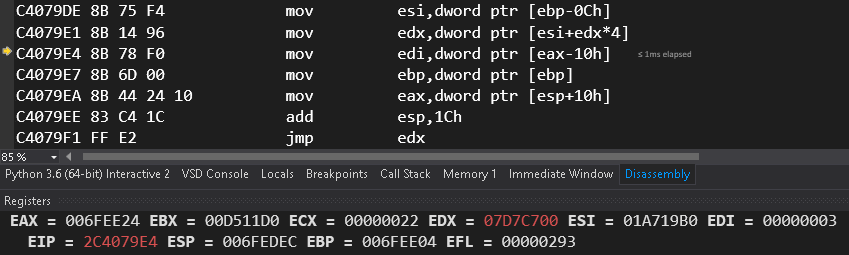
在0x78字节码指令集的头部, 指令将字节码封装表寄存器(即esi寄存器)转储至栈内0xC偏移处, 字节码流寄存器(edi寄存器)被推入栈内, 当前字节码偏移寄存器(ecx寄存器)被转储至栈内0x14偏移处. 此时, 三个关键寄存器均转储完毕.

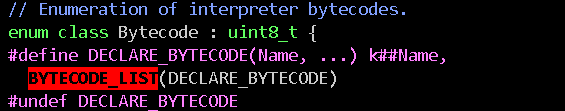




忽略功能性指令, 直接单步至尾部调用的相关指令集(Dispatch函数封装的指令). 可见尾部调用操作首先从栈内0x14处取出当前字节码偏移, 并转储至eax寄存器, 随后, lea指令将eax + 0x1后的值转储至ecx内, 从寄存器列表中可以监视到ecx已经从0x21增长为0x22, 此操作实际上与Dispatch封装函数利用Advance函数计算下一个字节码偏移有关, 在源代码中, Advance函数取出当前字节码成员bytecode\_并计算其总长度, 随后利用该值构造Node对象并返回. 由此就可以看出lea指令操作的0x1实际上就是0x78字节码的总长度, 同样也不难得出这样的结论: V8通过将当前字节码偏移量继续偏移当前字节码的总长度以计算出下一个字节码的偏移量, 并以此定位到字节码流的其它字节码.

接着, 指令通过一些偏移操作取出栈内的某个值并转储至edx寄存器, 观察监视器可知该值实际上就是BytecodeArray对象的指针(04E2699D). 紧接着, 指令集通过movzx将[ecx + edx]处的值(字节码流指针偏移头部长度与0x2个字节码长度)转储至edx, 显然edx将会是下一个字节码的具体值.





上文, 在0x78字节码指令集起始部分中被转储至栈内0xC处的封装表指针于此刻被取出, 并转储至esi寄存器, 接着[esi+edx\*4]地址处的内容被转储至edx, 结合前文对该地址结构进行分析, 操作数内的edx表示的是即将执行的下一个字节码, 而所有字节码均是由BYTECODE\_LIST宏生成的枚举数据, 同样, esi寄存器指向的封装表也由此宏按序对每个字节码指令集的入口地址进行封装, 这意味着字节码本身的值就是其句柄指令集地址在封装表内的下标(即0x78字节码的句柄指令集入口位于dispatch\_table内偏移0x78个指针大小处), 那么操作数内的0x4显然就是指针大小, 如此一来操作数[esi+edx\*4]的意义就很明确了. 那么最终转储至edx内的值0x7D7C700显然就是Code对象偏移0x3F位处的指令集入口, 最终, 指令集执行jmp指令跳转至0x7D7C700处以执行下一个字节码的句柄指令集. 至此, Bytecode的完整执行流程已经分析完毕, 后续的所有字节码也均遵循此尾部调用操作用以定位和执行其它字节码句柄.

// **写毕于 20240305**

**杂项笔记**

heap.h#Heap对象内的成员均由STRONG\_ROOT\_LIST宏初始化