## Bytecode调试器开发思路(待定)

**Author: MG193.7**

需求档用于记录什么:

* 在开发时需要解决的需求, 以及在开发时需要记录的小规模问题
  + 对无需新档进行分析的小规模问题, 以及解决方案进行记录
  + 当小规模问题以及需求不断累积至无法立即解决时, 认定为大规模问题时, 将问题/需求转移至开发档进行详细的解决

**TODO与需求列表**

* 调试启用时的数据收集
  + 需要明确当前执行的字节码集所在函数
    - ~~FLAG\_ignition\_filter实现分析~~ **弃用, 在编译早期实现 20240229**
    - 干预InterpreterEntryTrampoline封装过程, 将edi寄存器传递至外部可控处 **√ 20240311**
      * edi寄存器内JSFunction获取函数的方案需要修改 20240315
  + 需要在适当时机插入int3(0xcc)
    - Bytecode句柄的编译时期过早, 如何控制int3插入 **√ 20240227**
    - ~~Bytecode句柄的编译时期早于FLAG\_ignition\_filter实现~~ **被证伪! 20240301**
    - ~~在Builtin中的DispatchTable封装处call到外部可控处, 间接控制Bytecode执行(可能可行)~~ **弃用, 关键数据传递不可行 20240308**
    - InterpreterAssembler::TraceBytecode分析, 利用CallRuntime向外部传递关键数据, 间接控制Bytecode执行~~(可能可行)~~ **√ 20240308**
  + 挂起时机控制 20240305
    - Bytecode句柄编译时可能可以控制, 利用TraceBytecode接口 **√ 20240308**
    - 需要新档来分析TraceBytecode **√ 20240308**
* 调试时允许单步任意字节码
  + 传递挂起时机/因素 **√ 20240313**
    - 依赖FLAG宏来传递全局常量, 逗号分割
      * 需要挂起的字节码 --dbg\_bytecode √
      * 需要挂起的函数 --dbg\_function √
      * 需要挂起的地址 --dbg\_address
  + 在适当时机进入调试状态机 **√ 20240313**
    - 由于目前的方案是通过干预指令集的封装来间接控制字节码执行, 所以进入状态机的时机要在CallRuntime之后, 在指令集内实现状态机几乎不可能. 20240310
  + Return,ReThrow导致字节码流变动分析 **√ 20240320**
  + 监听CTRL\_C信号实现随时挂起 **√ 20240320**
  + 字节码执行时的显示格式 **√ 20240326**
    - ~~字节码本身~~
    - ~~具体字节~~
    - ~~操作数/寄存器/累加器~~
  + ~~需要多进程, IPC~~ **弃用, 目前没有必要 20240310**
* 调试时允许随时改变调试信息
  + 进入状态机时允许添加/删除断点
    - 状态机命令 breakrm/br,breakadd/ba **√ 20240314**
* ~~调试时允许跨越任意数量字节码~~
* 调试时允许查看寄存器
  + 随时输出寄存器状态
  + 寄存器Watcher
  + 需要新档来分析Register实现 **√ 20240410**
  + 需要新档来分析V8帧空间与结构 **√ 20240415**
  + ~~在新窗口实时显示寄存器状态~~
* 调试时允许查看累加器
* 调试时允许查看常量池 **√ 20240327**

// 延期note 20240321开始编写Blackhat slide和其它事项, 暂时不会进行开发

## 目前Hook/修改过的源文件

* interpreter/interpreter\_assembler.cc
  + Hook InterpreterAssembler::InterpreterAssmebler
* builtins/ia32/builtins-ia32.cc
  + Hook InterpreterEntryTrampoline
* runtime/runtime.h
  + 注册V8\_BYTECODE\_DEBUGGER宏
* flag-definitions.h
  + 注册dbg\_系列外部常量

## 调试器Bug记录

**20240315**

传递至V8BytecodeDebugger::BytecodeInterrupt函数内的参数bytecode\_offset有时会导致越界读.

* 原因是因为函数内获取的bytecode\_array其字节码总大小小于偏移大小.
* BytecodeInterrupt与Runtime\_V8BytecodeDebugger函数内获取的BytecodeArray有时会不一致, 原因不明
* 调用栈:  
  Runtime\_V8BytecodeDebugger

3e68607e() Unknown

3e6e9f60() Unknow, maybe dispatch\_table

3e689408() InterpreterEntryTrampoline

3e686cd6() InterruptCheck

3c461aa1() Unknow, maybe dispatch\_table, can reinterpret Code

3e68851e() Unknown

3e6e5918() JSEntryStub::Generate

* 每次Hook都更新字节码流可以解决该问题, 但仍不知道字节码流变动的原因
* **在执行Return字节码时更新流,不再出现该问题. 20240316**越界读问题可能出现在导致字节码流变更的字节码被执行时, 目前可能导致字节码流变更的字节码:
  + Return, ReThrow
  + Return字节码会导致执行返回到字节码跳板(InterpreterEntryTrampoline)

## Windows.h问题

直接使用#include<windows.h>会出现各种无法排查原因的问题, 比如函数重载, 某个对象无法通过编译, 很有可能是因为windows.h底下各类预定义的宏导致的问题, 而v8为了解决此类问题, 已经提供了一个完整的windows系列头文件的封装源文件”src/base/win32-headers.h”, 引入此文件即可解决大部分问题. 20240314

## Runtime宏注册/函数实现

为避免影响到FOR\_EACH\_INTRINSIC系列宏的内置常量出现异常, 新注册的宏常量由定义新宏来实现. 需要注意只有FOR\_EACH\_INTRINSIC\_RETURN\_OBJECT宏会注册Runtime\_前缀句柄

// 注册调试宏

// Define Debugger Macro For V8 Bytecode

#define V8\_BYTECODE\_DEBUGGER(F) \

F(V8BytecodeDebugger, 0, 1)

此处注册的测试句柄其宏常量为”V8BytecodeDebugger”, 利用RUNTIME\_FUNCTION宏实现Runtime\_V8BytecodeDebugger

需要有返回值, return isolate->heap()->undefined\_value()

利用CONVERT\_ARG\_HANDLE\_CHECKED宏或手动提取args参数内容.

宏内实现: Handle<Type> name = args.at<Type>(index);

## Bytecode调试信息提取

调试器进行时不可避免地需要使用到字节码流相关的数据, 关于处理字节码流, 参考原Runtime\_InterpreterTraceBytecodeEntry函数代码:

int offset = bytecode\_offset - BytecodeArray::kHeaderSize + kHeapObjectTag;

interpreter::BytecodeArrayIterator bytecode\_iterator(bytecode\_array);

AdvanceToOffsetForTracing(bytecode\_iterator, offset);

~~利用AdvanceToOffsetForTracing函数, 当前偏移以及字节码迭代器来获取运行时的每个字节码.~~

~~这里可能仍然需要分析迭代器的具体实现. 20240308~~

利用LoadRegister接口将字节码流所在寄存器传递至Runtime\_句柄内, 配合Handle组件即可操作BytecodeArray指针, 后续的字节码偏移提取与其它信息提取流程类似, 无需再利用其它接口实现. 20240309

实现示例:

Handle<BytecodeArray> bytecode\_array = args.at<BytecodeArray>(0);

int bytecode\_offset = args.smi\_at(1);

20240326 由于调试器的大部分框架都编写完毕, 所以目前的需求是需要更详细的字节码调试信息, 并随时控制运行时调试信息的输出

* 字节码的前缀抽取: Wide系列前缀, 详看Ignition设计文档内Wide operands章节
  + Wide, ExtraWide, DebugBreakExtraWide, DebugBreakWide offset + 1
* 字节码抽取
* 操作数抽取
  + 操作数数量
  + 操作数规模选择
    - switch (bytecode) {

case Bytecode::kExtraWide:

case Bytecode::kDebugBreakExtraWide:

return OperandScale::kQuadruple;

case Bytecode::kWide:

case Bytecode::kDebugBreakWide:

return OperandScale::kDouble;

default:

UNREACHABLE();

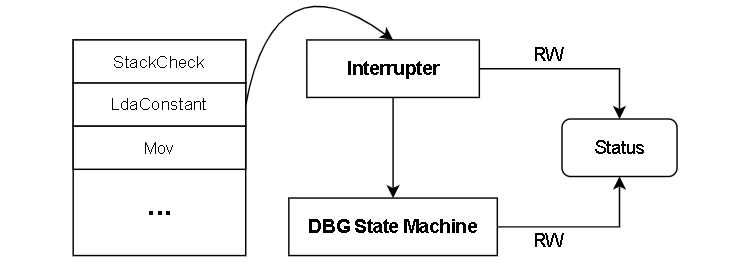
return OperandScale::kSingle;

}

## 调试器状态机架构

调试器需要一个全局状态来协调整个状态机的执行, 该状态由V8调试类中的Status枚举类型实现, 该枚举类型又由宏批量实现了数个状态, 截至目前, 状态机内拥有的状态有: 20240313

* SUSPEND
  + 挂起V8字节码的执行
* INT3
  + 挂起V8字节码的执行, 并且通知调试器!
* SKIP
  + 跳过挂起, 继续执行



如图为字节码执行至调试状态机的大概架构, 每次字节码执行将经过中断器判定是否进行挂起. 原则上, 中断器与状态机均可读写当前状态, 中断器对状态的写操作基于中断需求, 状态机对状态的写操作基于用户需求. 状态永远不会被重置, 且在下次经过中断器时充当中断因素

状态机命令:

* c/continue
  + 继续执行
* s/step/回车
  + 单步执行
* ba/breakadd/br/breakrm
  + 增加/删除断点
* bbl/bfl
  + 列出所有字节码/函数断点
* sb/showbytecode/sf/showfunction
  + 显示/不显示执行时字节码/函数

## Register调试信息

该需求记录于20240404, 由于问题规模扩大, 转移至开发档 20240409

## ~~多线程调试/随时挂起~~ 方案弃用! 20240310

考虑到不同版本,以及不同平台的可移植性, 初步方案是利用C++11的标准thread类库实现多线程方案, 而非使用MFC的系统类库.

挂起操作可利用输入的InterpreterAssembler指针向pc\_/buffer\_成员插入0xcc指令. 20240309

挂起操作利用标准mutex类库以及condition\_variable实现中断时机控制, 需要实现在执行任意字节码时都可以随时挂起. 20240309

利用此二者类库模拟字节码执行以及调试器挂起的测试代码, 参考如下:

using namespace std;

mutex mtx\_lock;

condition\_variable cv;

atomic<bool> int3(true);

atomic<int> c\_bytecode(0); // for record current bytecode

vector<int> bytecode = {0x78, 0x55, 0xff, 0xa1, 0xa2, 0x3f44, 0xf677, 0x97ff, 0xff0f, 0x1337};

// Simulate V8 bytecode execution

void process\_v8() {

for (int c : bytecode) {

{

unique\_lock<mutex> lck(mtx\_lock);

cout << "[-] Exec bytecode -> " << hex << c << "\n";

c\_bytecode = c;

int3 = true;

}

cv.notify\_all();

{

unique\_lock<mutex> lck(mtx\_lock);

cv.wait(lck, [] { return !int3; });

}

}

}

// Simulate V8 debugger

void debugger() {

while (true) {

{

unique\_lock<mutex> lck(mtx\_lock);

cv.wait(lck, [] { return c\_bytecode != 0; });

int bytecode\_to\_hook = c\_bytecode.load();

if (bytecode\_to\_hook == 0x1337) {

int3 = false;

cv.notify\_all();

break; // End of bytecode

}

if (bytecode\_to\_hook == 0xf677 || bytecode\_to\_hook == 0xa1) {

cout << ">> " << hex << bytecode\_to\_hook << " bytecode Hooked!\n";

}

c\_bytecode = 0;

int3 = false;

}

cv.notify\_all();

}

}

int main() {

thread ps\_v8(process\_v8);

thread dbg(debugger);

ps\_v8.join();

dbg.join();

return 0;

}