调试器

关键函数和结构体

- `DEBUG_EVENT`调试事件

```
typedef struct _DEBUG_EVENT {
  DWORD dwDebugEventCode;   // 调试信息编码，用来表示发生了什么事
  DWORD dwProcessId;        // 来源自哪个进程id
  DWORD dwThreadId;         // 来源自哪个线程id
  union {                   // 相当于变体，dwDebugEventCode成员对应下列的结构
    EXCEPTION_DEBUG_INFO      Exception;          // 异常调试信息
    CREATE_THREAD_DEBUG_INFO  CreateThread;       // 创建线程调试信息
    CREATE_PROCESS_DEBUG_INFO CreateProcessInfo;  // 创建进程调试信息
    EXIT_THREAD_DEBUG_INFO    ExitThread;         // 退出线程调试信息
    EXIT_PROCESS_DEBUG_INFO   ExitProcess;        // 退出进程调试信息
    LOAD_DLL_DEBUG_INFO       LoadDll;            // 加载DLL调试信息
    UNLOAD_DLL_DEBUG_INFO     UnloadDll;          // 卸载DLL调试信息
    OUTPUT_DEBUG_STRING_INFO  DebugString;        // OutputString
    RIP_INFO                  RipInfo;            // RIP
  } u;
} DEBUG_EVENT, *LPDEBUG_EVENT;
```

详细参考https://docs.microsoft.com/en-us/windows/win32/api/minwinbase/ns-minwinbase-debug_event

- `WaitForDebugEvent` 等待调试事件

```
BOOL WaitForDebugEvent(
  LPDEBUG_EVENT lpDebugEvent,   // 调试事件
  DWORD         dwMilliseconds  // 超时时间
);
```

- `ContinueDebugEvent`交还控制权

```
BOOL ContinueDebugEvent(
  DWORD dwProcessId,
  DWORD dwThreadId,
  DWORD dwContinueStatus     // 以何种状态继续
                         // DBG_CONTINUE代表我处理
                         // DBG_EXCEPTION_NOT_HANDLED代表我不处理
                         // DBG_REPLY_LATER
);
```

- `DebugActiveProcess`附加调试

```
BOOL DebugActiveProcess(
  DWORD dwProcessId
);
```

- `CreateProcess`
  - 标志`DEBUG_ONLY_THIS_PROCESS`，仅仅当前进程调试
  - 标志`DEBUG_PROCESS`，后续的子进程也会进入调试状态
- `VirtualQueryEx`查看进程的内存状态

```
SIZE_T VirtualQueryEx(
  HANDLE                    hProcess,
  LPCVOID                   lpAddress,
  PMEMORY_BASIC_INFORMATION lpBuffer,
  SIZE_T                    dwLength
);
```

- `SymXXX`系列函数自动解析符号文件

- `GetThreadContext`获取指定线程的上下文信息

```
BOOL GetThreadContext(
  HANDLE    hThread,
  LPCONTEXT lpContext
);
```

异常事件

```
EXCEPTION_DEBUG_EVENT    // 被调试的调试程序的时候来,会在调试的程序中下一个int3断点.如果被调试的时候,则回来,属于系统断点

CREATE_THREAD_DEBUG_EVENT        // 被调试的程序创建线程的时候会来

CREATE_PROCESS_DEBUG_EVENT       // 被调试的程序创建进程的时候会来

EXIT_THREAD_DEBUG_EVENT          // 被调试的程序退出线程的时候会来

EXIT_PROCESS_DEBUG_EVENT         // 被调试的程序退出进程的时候会来

LOAD_DLL_DEBUG_EVENT             // 被调试的程序加载DLL的时候会来

UNLOAD_DLL_DEBUG_EVENT           // 被调试的程序卸载DLL会来

OUTPUT_DEBUG_STRING_EVENT        // 被调试的程序调试输出的时候会来

RIP_EVENT                        // 64位系统的事件,如果编写32位调试器,这个则不重要.
```

调试器框架

调试器一般拥有两个线程：

1. UI控制
2. 调试控制
    - `CreateProcess` 标志 `DEBUG_ONLY_THIS_PROCESS` 创建进程
    - `DebugActiveProcess` 附加进程
    - 在调试循环中等待调试事件并处理

调试循环

```
#include <windows.h>

DWORD OnCreateThreadDebugEvent(const LPDEBUG_EVENT);
DWORD OnCreateProcessDebugEvent(const LPDEBUG_EVENT);
DWORD OnExitThreadDebugEvent(const LPDEBUG_EVENT);
DWORD OnExitProcessDebugEvent(const LPDEBUG_EVENT);
DWORD OnLoadDllDebugEvent(const LPDEBUG_EVENT);
DWORD OnUnloadDllDebugEvent(const LPDEBUG_EVENT);
DWORD OnOutputDebugStringEvent(const LPDEBUG_EVENT);
DWORD OnRipEvent(const LPDEBUG_EVENT);

// 调试循环
void EnterDebugLoop(const LPDEBUG_EVENT DebugEv)
{
    // 状态
    DWORD dwContinueStatus = DBG_CONTINUE;

    while(1)
    {
        // 等待调试事件
        WaitForDebugEvent(DebugEv, INFINITE);

        switch (DebugEv->dwDebugEventCode) // 异常编码
        {
            case EXCEPTION_DEBUG_EVENT: // 异常调试
                switch(DebugEv->u.Exception.ExceptionRecord.ExceptionCode)
                {
                    case EXCEPTION_ACCESS_VIOLATION: // 异常访问
                        break;

                    case EXCEPTION_BREAKPOINT: // 断点
                        break;

                    case EXCEPTION_DATATYPE_MISALIGNMENT:
                        break;

                    case EXCEPTION_SINGLE_STEP: // 单步
                        break;

                    case DBG_CONTROL_C:
                        break;
```

```
                    default:    // 其他
                        break;
                }

                break;

            case CREATE_THREAD_DEBUG_EVENT: // 创建线程
                dwContinueStatus = OnCreateThreadDebugEvent(DebugEv);
                break;

            case CREATE_PROCESS_DEBUG_EVENT: // 创建进程
                dwContinueStatus = OnCreateProcessDebugEvent(DebugEv);
                break;

            case EXIT_THREAD_DEBUG_EVENT: // 退出线程
                dwContinueStatus = OnExitThreadDebugEvent(DebugEv);
                break;

            case EXIT_PROCESS_DEBUG_EVENT: // 退出进程
                dwContinueStatus = OnExitProcessDebugEvent(DebugEv);
                break;

            case LOAD_DLL_DEBUG_EVENT: // 加载DLL
                dwContinueStatus = OnLoadDllDebugEvent(DebugEv);
                break;

            case UNLOAD_DLL_DEBUG_EVENT: // 卸载DLL
                dwContinueStatus = OnUnloadDllDebugEvent(DebugEv);
                break;

            case OUTPUT_DEBUG_STRING_EVENT: // OutputString
                dwContinueStatus = OnOutputDebugStringEvent(DebugEv);
                break;

            case RIP_EVENT: // RIP
                dwContinueStatus = OnRipEvent(DebugEv);
                break;
        }
        // 恢复执行报告调试事件的线程
        ContinueDebugEvent(DebugEv->dwProcessId, DebugEv->dwThreadId,
dwContinueStatus);
    }
}
```

详细参考：https://docs.microsoft.com/en-us/windows/win32/debug/writing-the-debugger-s-main-loop

反汇编引擎

随意选择了一个反汇编引擎，其中提供了两个导出函数

- `Decode2Asm`无机器码解析
- `Decode2AsmOpcode`带机器码解析

```
// 无机器码解析
```

```
extern "C"
void
__stdcall
Decode2Asm(IN PBYTE pCodeEntry,   // 需要解析指令地址
           OUT char* strAsmCode,  // 得到反汇编指令信息
           OUT UINT* pnCodeSize,  // 解析指令长度
           UINT nAddress);

// 带机器码解析
extern "C"
void
__stdcall
Decode2AsmOpcode(IN PBYTE pCodeEntry,   // 需要解析指令地址
           OUT char* strAsmCode,  // 得到反汇编指令信息
           OUT char* strOpcode,   // 解析机器码信息
           OUT UINT* pnCodeSize,  // 解析指令长度
           UINT nAddress);
```

使用如下：

```
// 指令
unsigned char szCode[] = {0x89,0x75,0xFC,0xEB,0x0E,0x33,0xC0,0x40,0xC3};
char szAsmCode[260];    // 存放反汇编后的字符串
char szOpCode[260];     // 存放机器码的字符串
UINT nCodeSize;         // 当前指令的长度
unsigned char *pCode = szCode;  // 需要解析的指令的地址
int nCodeBegin = 0x00401000;    // 设置当前指令的地址
for (int i = 0; i < 8; i++)
{
    Decode2AsmOpcode(pCode, szAsmCode, szOpCode, &nCodeSize, nCodeBegin);
    strlwr(szAsmCode);
    pCode += nCodeSize;     // 下一条指令
    printf("%-10p%-20s%-20s\n", nCodeBegin, szOpCode, szAsmCode);
    nCodeBegin += nCodeSize;    // 下一条指令的地址
}
```

在汇编中的使用

在汇编中调用C/C++模块，需要先生成C/C++的`obj`文件，然后在汇编中声明接口，编译同C/C++的模块一起链接进可执行文件

因为汇编中默认使用`__stdcall`调用约定，为了方便使用，在C/C++中对于接口也应该使用`__stdcall`

注：为了防止C++的名称粉碎，都用上`extern "C"`强制使用C的名称粉碎

比如：

```
; 读取指令
invoke ReadProcessMemory,g_hProcess,
                         [ebx].u.Exception.pExceptionRecord.ExceptionAddress,
                         @OpCode,sizeof @OpCode, NULL
invoke ShowAsm,addr @OpCode,[ebx].u.Exception.pExceptionRecord.ExceptionAddress


; 封装 Decode2AsmOpcode
ShowAsm proc pCode :ptr byte, pAddr:ptr Byte
    LOCAL  @szAsmCode[256]:BYTE
```

```
    LOCAL  @szOpCode[256]:BYTE
    LOCAL  @CodeSize1:DWORD

    invoke Decode2AsmOpcode, pCode, addr @szAsmCode, addr @szOpCode,
                             addr @CodeSize1, pAddr
    invoke crt_printf, offset g_szDcode, pAddr, addr @szOpCode, addr @szAsmCode

    ret
ShowAsm endp
```

断点

调试器的核心便是断点，根据断点的类型不同，可以分为：软断点，硬断点，内存断点。

软断点

软断点实际上是一个单字节的指令，改指令可以引发软中断，将当前进程的控制权交给OS。在X86的架构上，是`int3`即`0xCC`

简单地说就是将你要断下的指令地址处的第一个字节设置为`0xCC`，软件执行到`0xCC`（对应汇编指令`int3`）时，会触发异常代码为`EXCEPTION_BREAKPOINT`的异常

`int3`通知调试器让程序挂起，没有调试器的时候触发异常，`int3`没有跑完，有调试器的时候`int3`是执行了的，断下来后在调试器中获取的`eip`是`int3`后面指令的地址

- 断点的设置：

  1.首先,系统断点第一次来,然后在创建进程的时候会有一个地址,我们使用`Read...`读取地址内容,然后反汇编出来显示

  2.读取出来之前,使用`VirtualProtectEx`将保护属性去除,(注意保存旧的)

  3.使用`WriteProcessMemory`往地址写入CC(注意保存以前的值)

  4.重新修改保护属性,改回去(使用旧的)

- 单步

  1.判断是否使我们设置的断点

  2.修改内存保护属性(注意保存旧的)

  3.写入CC,(int 3断点)

  4.读取内存数据

  5.显示反汇编

  - 单步步入

    1.打开线程获得线程句柄

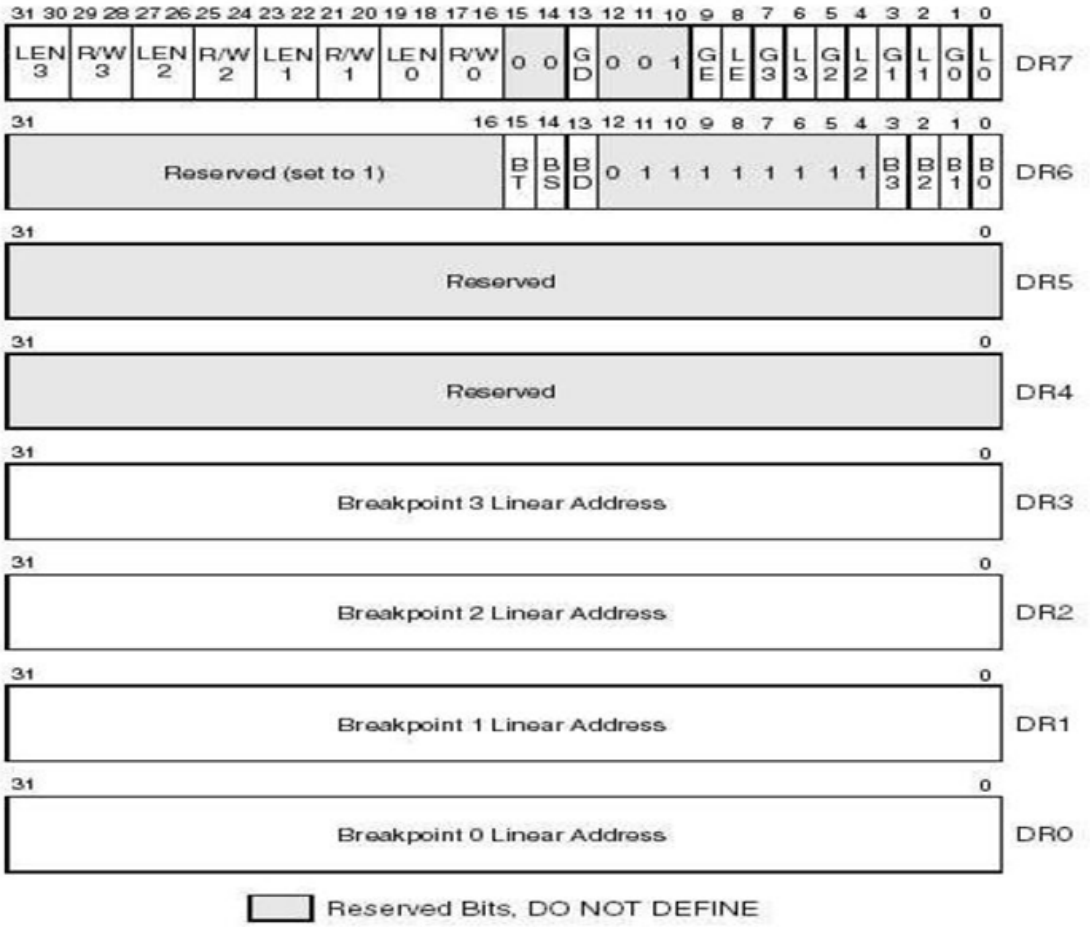
    2.使用`GetThreadContext`获取寄存器的值

    3.设置单步标志,单步表示是要我们设置的,他是第9个标志

    `or [esi].regflag,0100h`这样设置即可.设置第九位为1

    4.设置寄存器环境`SetThreadContext`

硬件断点



硬件断点 收到单步异常
dr0-dr7 每个四字节
dr0-3存放地址
dr4-5保留
dr6状态寄存器组
  0~3位，表示dr0-dr3哪一个生效
  BS = 1 硬件断点
  BT 任务切换
  BD 保护位，不允许修改
dr7控制寄存器组
  0~7位 L进程内断点，G全局断点，L0-L3表示dr0-dr3哪一个生效
  GL、GE精确断点
  四组LEN（R/W）描述断点类型

在dr7设置好硬件断点的信息

在调试循环中单步命中，检查dr6的BS位

内存断点

修改内存属性，触发异常

还原属性、单步