Capstone反汇编引擎数据类型及API分析与示例(一)

kabeor / 2019-07-26 09:44:00 / 浏览数 3962 安全技术 二进制安全 顶(0) 踩(0)

最近准备用开源的反汇编引擎做个项目，研究了OllyDebug的ODDisasm，disasm与assembl部分代码的思想都很值得学习，但毕竟是2000年的产物，指令集只支持x86，

Capstone反汇编引擎可以说是如今世界上最优秀的反汇编引擎，IDA，Radare2，Qemu等著名项目都使用了Capstone Engine，所以选择它来开发是一个不错的选择。
但在开发时发现官方并未给出详细API文档，网上也没有类似的分析，因此想到自己阅读源码和试验，由此写出了一个简单的非官方版本的API手册，希望能与大家分享。

个人博客： kabeor.cn

0x0 开发准备

Capstone官网： http://www.capstone-engine.org

自行编译lib和dll方法

源码： https://github.com/aquynh/capstone/archive/4.0.1.zip

下载后解压
文件结构如下：

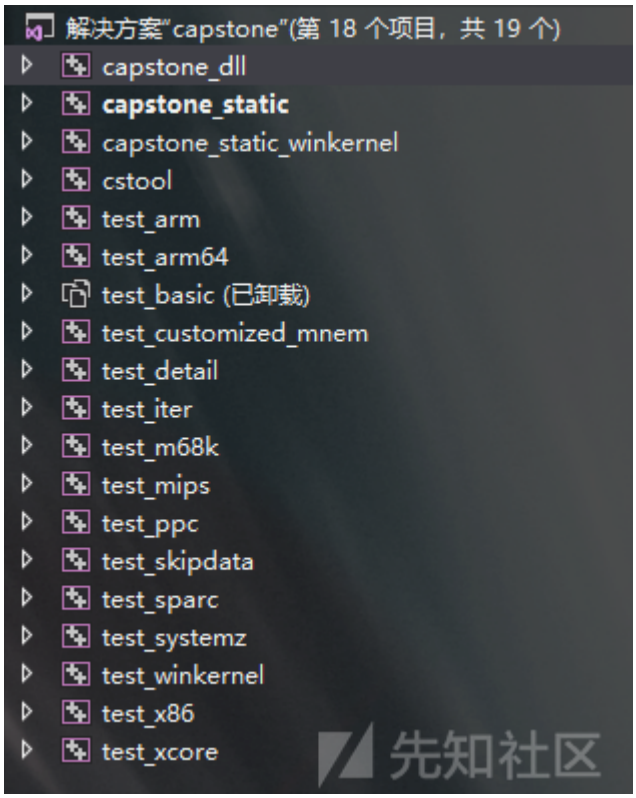
```
. <- 主要引擎core engine + README + 编译文档COMPILE.TXT 等
├── arch <- 各语言反编译支持的代码实现
│   ├── AArch64 <- ARM64 (aka ARMv8) 引擎
│   ├── ARM <- ARM 引擎
│   ├── EVM <- Ethereum 引擎
│   ├── M680X <- M680X 引擎
│   ├── M68K <- M68K 引擎
│   ├── Mips <- Mips 引擎
│   ├── PowerPC <- PowerPC 引擎
│   ├── Sparc <- Sparc 引擎
│   ├── SystemZ <- SystemZ 引擎
│   ├── TMS320C64x <- TMS320C64x 引擎
│   ├── X86 <- X86 引擎
│   └── XCore <- XCore 引擎
├── bindings <- 中间件
│   ├── java <- Java 中间件 + 测试代码
│   ├── ocaml <- Ocaml 中间件 + 测试代码
│   └── python <- Python 中间件 + 测试代码
├── contrib <- 社区代码
├── cstool <- Cstool 检测工具源码
├── docs <- 文档，主要是capstone的实现思路
├── include <- C头文件
├── msvc <- Microsoft Visual Studio 支持（Windows）
├── packages <- Linux/OSX/BSD包
├── windows <- Windows 支持(Windows内核驱动编译)
├── suite <- Capstone开发测试工具
├── tests <- C语言测试用例
└── xcode <- Xcode 支持 (MacOSX 编译)
```

下面演示Windows10使用Visual Studio2019编译

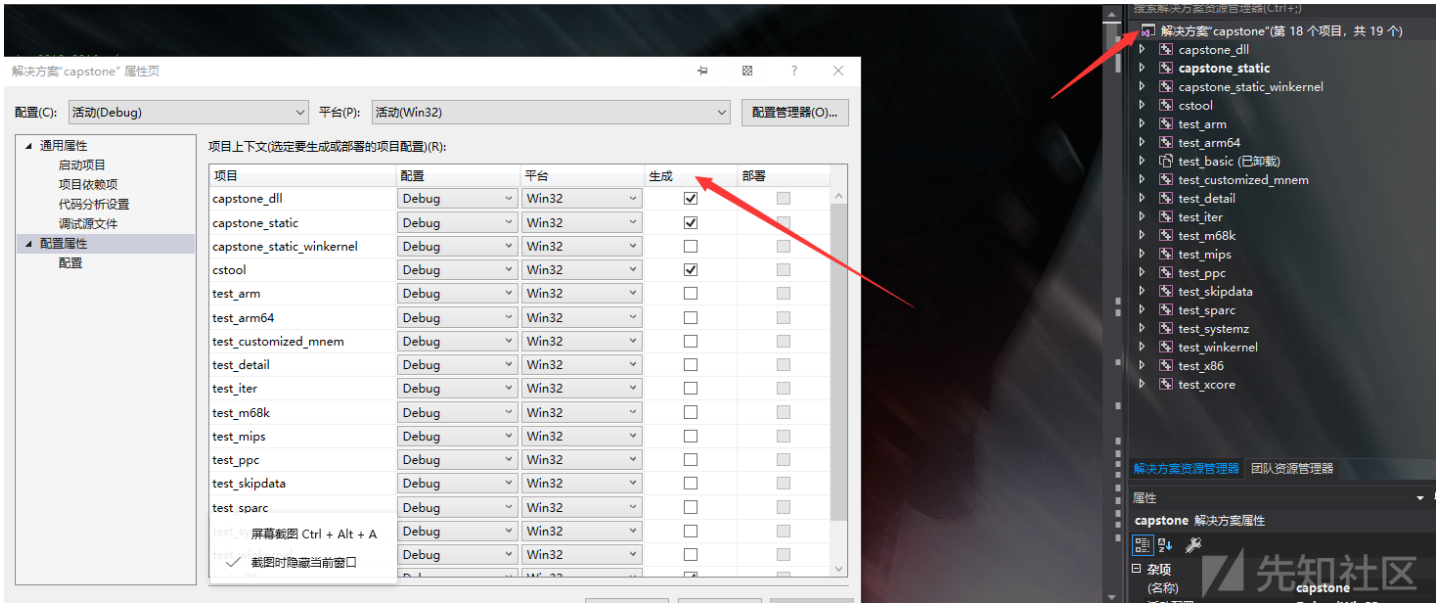
复制msvc文件夹到一个比较清爽的位置（强迫症专用），内部结构如下：

capstone_dll	2019/7/19 13:06	文件夹	
capstone_static	2019/7/19 13:12	文件夹	
capstone_static_winkernel	2019/1/10 21:45	文件夹	
cstool	2019/7/19 13:06	文件夹	
Debug	2019/7/19 13:12	文件夹	
test_arm	2019/7/19 13:06	文件夹	
test_arm64	2019/7/19 13:06	文件夹	
test_customized_mnem	2019/7/19 13:06	文件夹	
test_detail	2019/7/19 13:06	文件夹	
test_iter	2019/7/19 13:06	文件夹	
test_m68k	2019/7/19 13:06	文件夹	
test_mips	2019/7/19 13:06	文件夹	
test_ppc	2019/7/19 13:06	文件夹	
test_skipdata	2019/7/19 13:06	文件夹	
test_sparc	2019/7/19 13:06	文件夹	
test_systemz	2019/7/19 13:06	文件夹	
test_winkernel	2019/1/10 21:45	文件夹	
test_x86	2019/7/19 13:06	文件夹	
test_xcore	2019/7/19 13:06	文件夹	
capstone.sln	2019/7/19 13:12	Visual Studio Sol...	16 KB
README	2019/1/10 21:45	文件	2 KB

VS打开capstone.sln项目文件，解决方案自动载入这些



可以看到支持的所有语言都在这里了，如果都需要的话，直接编译就好了，只需要其中几种，则右键解决方案->属性->配置属性 如下



生成选项中勾选你需要的支持项即可
编译后会在当前文件夹Debug目录下生成capstone.lib静态编译库和capstone.dll动态库这样就可以开始使用Capstone进行开发了

如果不想自己编译，官方也提供了官方编译版本
Win32： https://github.com/aquynh/capstone/releases/download/4.0.1/capstone-4.0.1-win32.zip
Win64： https://github.com/aquynh/capstone/releases/download/4.0.1/capstone-4.0.1-win64.zip

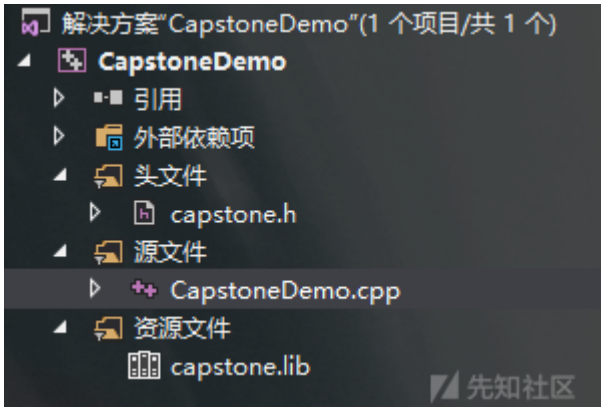
选x32或x64将影响后面开发的位数

引擎调用测试

新建一个VS项目，将..\capstone-4.0.1\include\capstone中的头文件以及编译好的lib和dll文件全部拷贝到新建项目的主目录下

.vs	2019/7/19 13:17	文件夹	
Debug	2019/7/19 14:49	文件夹	
x64	2019/7/19 14:51	文件夹	
arm.h	2019/1/10 21:45	C/C++ Header	19 KB
arm64.h	2019/1/10 21:45	C/C++ Header	28 KB
capstone.dll	2019/1/10 21:54	应用程序扩展	3,758 KB
capstone.h	2019/7/19 14:34	C/C++ Header	29 KB
capstone.lib	2019/1/10 21:54	Object File Library	6 KB
CapstoneDemo.cpp	2019/7/19 14:51	c_file	1 KB
CapstoneDemo.sln	2019/7/19 13:17	Visual Studio Sol...	2 KB
CapstoneDemo.vcxproj	2019/7/19 14:33	VC++ Project	8 KB
CapstoneDemo.vcxproj.filters	2019/7/19 14:33	VC++ Project Fil...	2 KB
CapstoneDemo.vcxproj.user	2019/7/19 13:17	Per-User Project...	1 KB
evm.h	2019/1/10 21:45	C/C++ Header	5 KB
m68k.h	2019/1/10 21:45	C/C++ Header	14 KB
m680x.h	2019/1/10 21:45	C/C++ Header	13 KB
mips.h	2019/1/10 21:45	C/C++ Header	17 KB
platform.h	2019/1/10 21:45	C/C++ Header	4 KB
ppc.h	2019/1/10 21:45	C/C++ Header	26 KB
sparc.h	2019/1/10 21:45	C/C++ Header	12 KB
systemz.h	2019/1/10 21:45	C/C++ Header	14 KB
tms320c64x.h	2019/1/10 21:45	C/C++ Header	9 KB
x86.h	2019/1/10 21:45	C/C++ Header	42 KB
xcore.h	2019/1/10 21:45	C/C++ Header	5 KB

在VS解决方案中，头文件添加现有项capstone.h，资源文件中添加capstone.lib，重新生成解决方案



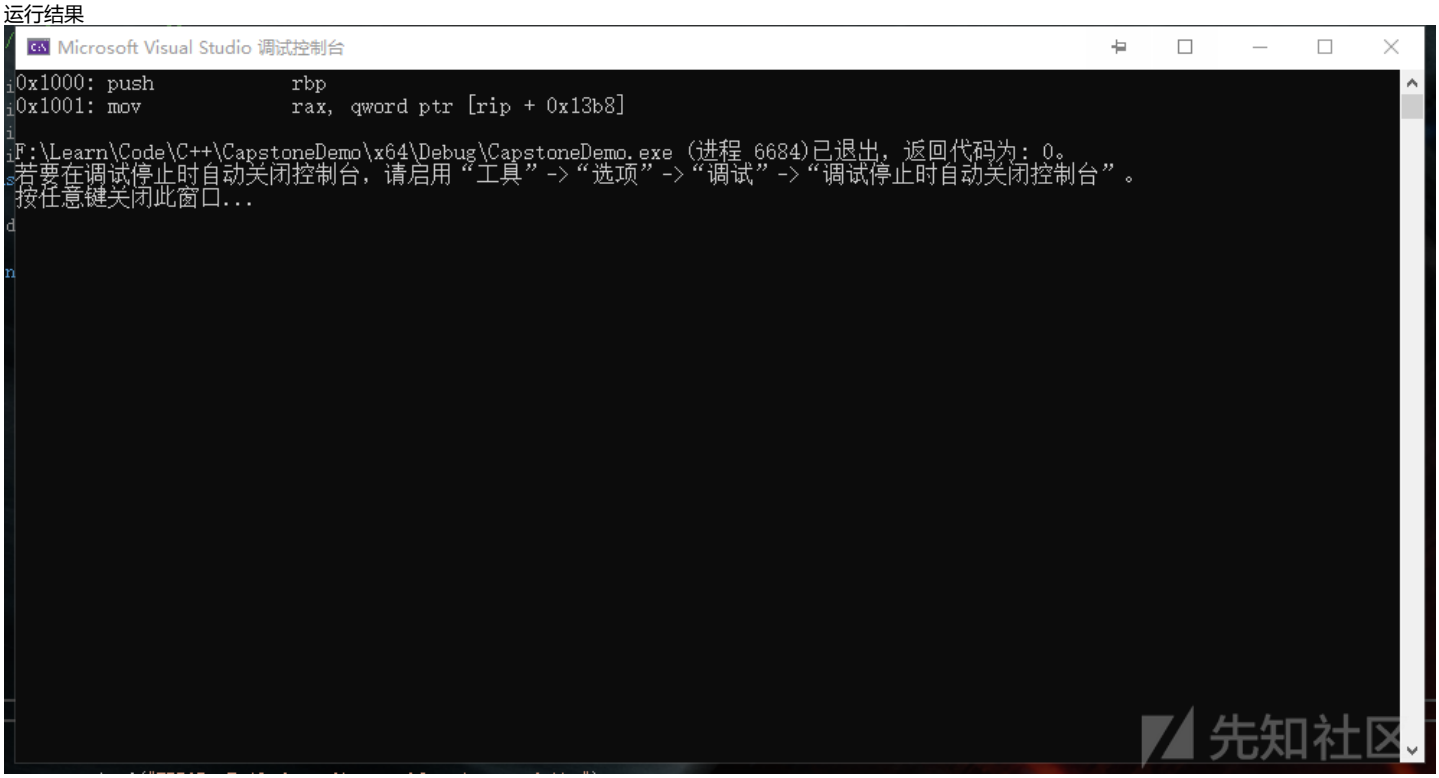
那么现在来测试一下我们自己的capstone引擎吧

主文件写入如下代码

```
#include <iostream>
#include <stdio.h>
#include <cinttypes>
#include "capstone.h"
using namespace std;

#define CODE "\x55\x48\x8b\x05\xb8\x13\x00\x00"

int main(void)
{
   csh handle;
   cs_insn* insn;
   size_t count;

   if (cs_open(CS_ARCH_X86, CS_MODE_64, &handle)) {
       printf("ERROR: Failed to initialize engine!\n");
       return -1;
   }

   count = cs_disasm(handle, (unsigned char*)CODE, sizeof(CODE) - 1, 0x1000, 0, &insn);
   if (count) {
       size_t j;

       for (j = 0; j < count; j++) {
           printf("0x%""Ix"":\t%s\t\t%s\n", insn[j].address, insn[j].mnemonic, insn[j].op_str);
       }

       cs_free(insn, count);
   }
   else
       printf("ERROR: Failed to disassemble given code!\n");

   cs_close(&handle);

   return 0;
}
```

事实上这是官方给出的C语言开发唯一几个例子之一，但注意到代码cs_open(CS_ARCH_X86, CS_MODE_64,
&handle)，测试的是archx64的反编译，因此编译选项也需要设置为x64，除此以外，如果你的项目像我一样是c++开发，那么printf("0x%""Ix"":\t%s\t\t%s\n",
insn[j].address, insn[j].mnemonic,
insn[j].op_str);处官方给出的"0x%"PRIx64":\t%s\t\t%s\n"应修改为我这里的"0x%""Ix"":\t%s\t\t%s\n"，这是inttypes支持问题。

运行结果



0x1 数据类型及API分析

数据类型

csh

用于生成调用capstone API的句柄

```
size_t csh
```

用法： `csh handle;`

cs_arch

架构选择

```
enum cs_arch {
   CS_ARCH_ARM = 0,    ///< ARM ■■ (■■ Thumb, Thumb-2)
   CS_ARCH_ARM64,      ///< ARM-64, ■■ AArch64
   CS_ARCH_MIPS,       ///< Mips ■■
  CS_ARCH_X86,     ///< X86 ■■ (■■ x86 & x86-64)
   CS_ARCH_PPC,        ///< PowerPC ■■
   CS_ARCH_SPARC,      ///< Sparc ■■
   CS_ARCH_SYSZ,       ///< SystemZ ■■
   CS_ARCH_XCORE,      ///< XCore ■■
   CS_ARCH_M68K,       ///< 68K ■■
   CS_ARCH_TMS320C64X, ///< TMS320C64x ■■
   CS_ARCH_M680X,      ///< 680X ■■
   CS_ARCH_EVM,        ///< Ethereum ■■
   CS_ARCH_MAX,
   CS_ARCH_ALL = 0xFFFF, // All ■■ - for cs_support()
} cs_arch;
```

用法：API中cs_arch参数填入枚举内容，如API中cs_open(cs_arch arch, cs_mode mode, csh *handle);第一个参数填CS_ARCH_X86则支持X86 架构

cs_mode

模式选择

```
enum cs_mode {
   CS_MODE_LITTLE_ENDIAN = 0,  ///< little-endian ■■ (default ■■)
   CS_MODE_ARM = 0,    ///< 32-bit ARM
   CS_MODE_16 = 1 << 1,    ///< 16-bit ■■ (X86)
   CS_MODE_32 = 1 << 2,    ///< 32-bit ■■ (X86)
```

```
   CS_MODE_64 = 1 << 3,    ///< 64-bit ■■ (X86, PPC)
   CS_MODE_THUMB = 1 << 4, ///< ARM's Thumb ■■, including Thumb-2
   CS_MODE_MCLASS = 1 << 5,    ///< ARM's Cortex-M series
   CS_MODE_V8 = 1 << 6,    ///< ARMv8 A32 encodings for ARM
   CS_MODE_MICRO = 1 << 4, ///< MicroMips ■■ (MIPS)
   CS_MODE_MIPS3 = 1 << 5, ///< Mips III ISA
   CS_MODE_MIPS32R6 = 1 << 6, ///< Mips32r6 ISA
   CS_MODE_MIPS2 = 1 << 7, ///< Mips II ISA
   CS_MODE_V9 = 1 << 4, ///< SparcV9 ■■ (Sparc)
   CS_MODE_QPX = 1 << 4, ///< Quad Processing eXtensions ■■ (PPC)
   CS_MODE_M68K_000 = 1 << 1, ///< M68K 68000 ■■
   CS_MODE_M68K_010 = 1 << 2, ///< M68K 68010 ■■
   CS_MODE_M68K_020 = 1 << 3, ///< M68K 68020 ■■
   CS_MODE_M68K_030 = 1 << 4, ///< M68K 68030 ■■
   CS_MODE_M68K_040 = 1 << 5, ///< M68K 68040 ■■
   CS_MODE_M68K_060 = 1 << 6, ///< M68K 68060 ■■
   CS_MODE_BIG_ENDIAN = 1 << 31,   ///< big-endian ■■
   CS_MODE_MIPS32 = CS_MODE_32,    ///< Mips32 ISA (Mips)
   CS_MODE_MIPS64 = CS_MODE_64,    ///< Mips64 ISA (Mips)
   CS_MODE_M680X_6301 = 1 << 1, ///< M680X Hitachi 6301,6303 ■■
   CS_MODE_M680X_6309 = 1 << 2, ///< M680X Hitachi 6309 ■■
   CS_MODE_M680X_6800 = 1 << 3, ///< M680X Motorola 6800,6802 ■■
   CS_MODE_M680X_6801 = 1 << 4, ///< M680X Motorola 6801,6803 ■■
   CS_MODE_M680X_6805 = 1 << 5, ///< M680X Motorola/Freescale 6805 ■■
   CS_MODE_M680X_6808 = 1 << 6, ///< M680X Motorola/Freescale/NXP 68HC08 ■■
   CS_MODE_M680X_6809 = 1 << 7, ///< M680X Motorola 6809 ■■
   CS_MODE_M680X_6811 = 1 << 8, ///< M680X Motorola/Freescale/NXP 68HC11 ■■
   CS_MODE_M680X_CPU12 = 1 << 9, ///< M680X Motorola/Freescale/NXP CPU12
                  ///< ■■ M68HC12/HCS12
   CS_MODE_M680X_HCS08 = 1 << 10, ///< M680X Freescale/NXP HCS08 ■■
} cs_mode;
```

用法：API中cs_mode参数填入枚举内容，如API中cs_open(cs_arch arch, cs_mode mode, csh *handle);第二个参数填CS_MODE_64则支持X64模式

cs_opt_mem

内存操作

```
struct cs_opt_mem {
   cs_malloc_t malloc;
   cs_calloc_t calloc;
   cs_realloc_t realloc;
   cs_free_t free;
   cs_vsnprintf_t vsnprintf;
} cs_opt_mem;
```

用法：可使用用户自定义的malloc/calloc/realloc/free/vsnprintf()函数，默认使用系统自带malloc(), calloc(), realloc(), free() & vsnprintf()

cs_opt_mnem

自定义助记符

```
struct cs_opt_mnem {
   /// ■■■■■■■■ID
   unsigned int id;
   /// ■■■■■■■■
   const char *mnemonic;
} cs_opt_mnem;
```

cs_opt_type

反编译的运行时选项

```
enum cs_opt_type {
   CS_OPT_INVALID = 0, ///< ■■■■■
   CS_OPT_SYNTAX,  ///< ■■■■■■
   CS_OPT_DETAIL,  ///< ■■■■■■■■■■■■
   CS_OPT_MODE,    ///< ■■■■■■■■■■
   CS_OPT_MEM, ///< ■■■■■■■■■■■■■■
   CS_OPT_SKIPDATA, ///< ■■■■■■■■■■■■■■■■■■SKIPDATA■■
   CS_OPT_SKIPDATA_SETUP, ///< ■SKIPDATA■■■■■■■■■■■
```

```
   CS_OPT_MNEMONIC, ///<■■■■■■■■
   CS_OPT_UNSIGNED, ///< ■■■■■■■■■■■■■■
} cs_opt_type;
```

用法：API cs_option(csh handle, cs_opt_type type, size_t value);中第二个参数

cs_opt_value

运行时选项值(与cs_opt_type关联)

```
enum cs_opt_value {
   CS_OPT_OFF = 0,  ///< ■■■■■■■ - ■■■CS_OPT_DETAIL, CS_OPT_SKIPDATA, CS_OPT_UNSIGNED.
   CS_OPT_ON = 3, ///< ■■■■■■■ (CS_OPT_DETAIL, CS_OPT_SKIPDATA).
   CS_OPT_SYNTAX_DEFAULT = 0, ///< ■■asm■■ (CS_OPT_SYNTAX).
   CS_OPT_SYNTAX_INTEL, ///< X86 Intel asm■■ - ■■■■ X86 (CS_OPT_SYNTAX).
   CS_OPT_SYNTAX_ATT,   ///< X86 ATT ■■■■ (CS_OPT_SYNTAX).
   CS_OPT_SYNTAX_NOREGNAME, ///< ■■■■■■■■■■ (CS_OPT_SYNTAX)
   CS_OPT_SYNTAX_MASM, ///< X86 Intel Masm ■■ (CS_OPT_SYNTAX).
} cs_opt_value;
```

用法：API cs_option(csh handle, cs_opt_type type, size_t value);中第三个参数

cs_op_type

通用指令操作数类型，在所有架构中保持一致

```
enum cs_op_type {
   CS_OP_INVALID = 0,  ///< ■■■■/■■■■■■
   CS_OP_REG,          ///< ■■■■■■■
   CS_OP_IMM,          ///< ■■■■■■
   CS_OP_MEM,          ///< ■■■■■■
   CS_OP_FP,           ///< ■■■
} cs_op_type;
```

目前开放的API中未调用

cs_ac_type

通用指令操作数访问类型，在所有架构中保持一致
可以组合访问类型，例如:CS_AC_READ | CS_AC_WRITE

```
enum cs_ac_type {
   CS_AC_INVALID = 0,        ///< ■■■■/■■■■■■■
   CS_AC_READ    = 1 << 0,   ///< ■■■■■■■■■■■■■■
   CS_AC_WRITE   = 1 << 1,   ///< ■■■■■■■■■■■■■■
} cs_ac_type;
```

目前开放的API中未调用

cs_group_type

公共指令组，在所有架构中保持一致

```
cs_group_type {
   CS_GRP_INVALID = 0,  ///< ■■■■/■■■■■
   CS_GRP_JUMP,    ///< ■■■■■■■(■■■■+■■■■+■■■■)
   CS_GRP_CALL,    ///< ■■■■■■■
   CS_GRP_RET,     ///< ■■■■■■■
   CS_GRP_INT,     ///< ■■■■■■■(int+syscall)
   CS_GRP_IRET,    ///< ■■■■■■■■■
   CS_GRP_PRIVILEGE,    ///< ■■■■■■■
   CS_GRP_BRANCH_RELATIVE, ///< ■■■■■■■■■
} cs_group_type;
```

目前开放的API中未调用

cs_opt_skipdata

用户自定义设置SKIPDATA选项

```
struct cs_opt_skipdata {
   /// Capstone■■■■■■■■■■■■“■■”
```

```
   /// ■■■■■■■■■■■■■■“■■■”■■■
   /// ■■■■■■(@mnemonic■NULL)■ Capstone■■“.byte”
   const char *mnemonic;

   /// ■■■■■■■■■■■■Capstone■■■■■■■■
   /// ■■■■■■■■■■■■■■(>0)■Capstone■■■■■■■■■■■■■■■■■■■■0,Capstone■■■■■■■■■■cs_disasm()■■
   /// ■■:■■■■■■■■■■■■Capstone■■■■■■■■■■■■■■■■■:
   /// Arm:     2 bytes (Thumb mode) or 4 bytes.
   /// Arm64:   4 bytes.
   /// Mips:    4 bytes.
   /// M680x:   1 byte.
   /// PowerPC: 4 bytes.
   /// Sparc:   4 bytes.
   /// SystemZ: 2 bytes.
   /// X86:     1 bytes.
   /// XCore:   2 bytes.
   /// EVM:     1 bytes.
   cs_skipdata_cb_t callback;  // ■■■■ NULL

   /// ■■■■■■■■■■■■■@callback■■■■
   void *user_data;
} cs_opt_skipdata;
```

目前开放的API中未调用

cs_detail

注意:只有当CS_OPT_DETAIL = CS_OPT_ON时，cs_detail中的所有信息才可用

在arch/ARCH/ARCHDisassembler.c的ARCH_getInstruction中初始化为memset(., 0, offsetof(cs_detail, ARCH)+sizeof(cs_ARCH))

如果cs_detail发生了变化，特别是在union之后添加了字段，那么相应地更新arch/ arch/ archdisassembly .c

```
struct cs_detail {
   uint16_t regs_read[12]; ///< ■■■■■■■■■■■■■■
   uint8_t regs_read_count; ///< ■■■■■■■■■■■■■■

   uint16_t regs_write[20]; ///< ■■■■■■■■■■■■■■
   uint8_t regs_write_count; ///< ■■■■■■■■■■■■■■

   uint8_t groups[8]; ///< ■■■■■■■■■■■■■
   uint8_t groups_count; ///< ■■■■■■■■■■

   /// ■■■■■■■■■■■
   union {
       cs_x86 x86;     ///< X86 ■■, ■■ 16-bit, 32-bit & 64-bit ■■
       cs_arm64 arm64; ///< ARM64 ■■ (aka AArch64)
       cs_arm arm;     ///< ARM ■■ (■■ Thumb/Thumb2)
       cs_m68k m68k;   ///< M68K ■■
       cs_mips mips;   ///< MIPS ■■
       cs_ppc ppc;     ///< PowerPC ■■
       cs_sparc sparc; ///< Sparc ■■
       cs_sysz sysz;   ///< SystemZ ■■
       cs_xcore xcore; ///< XCore ■■
       cs_tms320c64x tms320c64x;  ///< TMS320C64x ■■
       cs_m680x m680x; ///< M680X ■■
       cs_evm evm;     ///< Ethereum ■■
   };
} cs_detail;
```

cs_insn

指令的详细信息

```
struct cs_insn {
   /// ■■ID(■■■■■■■■■■■■■■■■■■ID)
   /// ■■■■■■■■■■■■■■■'[ARCH]_insn' enum■■■■id■■ARM.h■■'arm_insn'■■ARM, X86.h■■'x86_insn'■■X86■…
   /// ■■■CS_OPT_DETAIL = CS_OPT_OFF■■■■■■■■■■
   /// ■■:■Skipdata■■■■■■id■■■“data”■■■0
   unsigned int id;
```

```
   /// ■■■■ (EIP)
   /// ■■■CS_OPT_DETAIL = CS_OPT_OFF■■■■■■■■■■
   uint64_t address;

   /// ■■■■
   /// ■■■CS_OPT_DETAIL = CS_OPT_OFF■■■■■■■■■■
   uint16_t size;

   /// ■■■■■■■■■■■■■■■■■■@size■■
   /// ■■■CS_OPT_DETAIL = CS_OPT_OFF■■■■■■■■■■
   uint8_t bytes[16];

   /// ■■■Ascii■■■■■
   /// ■■■CS_OPT_DETAIL = CS_OPT_OFF■■■■■■■■■■
   char mnemonic[CS_MNEMONIC_SIZE];

   /// ■■■■■■■Ascii■■
   /// ■■■CS_OPT_DETAIL = CS_OPT_OFF■■■■■■■■■■
   char op_str[160];

   /// cs_detail■■
   /// ■■:■■■■■■■■■■■■■■detail■■■■■:
   /// (1) CS_OP_DETAIL = CS_OPT_ON
   /// (2) ■■■■■■Skipdata■■(CS_OP_SKIPDATA■■■■■■CS_OPT_ON)
   ///
   /// ■■2:■■■Skipdata■■■detail■■■■■■■■■■■■■■■NULL■■■■■■■■■■■■■■■
   cs_detail *detail;
} cs_insn;
```

cs_err

Capstone API遇到的各类型的错误时cs_errno()的返回值

```
typedef enum cs_err {
   CS_ERR_OK = 0,   ///< ■■■
   CS_ERR_MEM,      ///< ■■■■: cs_open(), cs_disasm(), cs_disasm_iter()
   CS_ERR_ARCH,     ///< ■■■■■■■: cs_open()
   CS_ERR_HANDLE,   ///<■■■■■■: cs_op_count(), cs_op_index()
   CS_ERR_CSH,      ///< csh■■■■■■: cs_close(), cs_errno(), cs_option()
   CS_ERR_MODE,     ///< ■■■■■■■■■■■: cs_open()
   CS_ERR_OPTION,   ///< ■■■■■■■■■■■: cs_option()
   CS_ERR_DETAIL,   ///< ■■■■■■■■■detail■■■■■■■
   CS_ERR_MEMSETUP, ///< ■■■■■■■■■■■(■ CS_OPT_MEM)
   CS_ERR_VERSION,  ///< ■■■■■■ (bindings)
   CS_ERR_DIET,     ///< ■“diet”■■■■■■■■■■■■
   CS_ERR_SKIPDATA, ///< ■SKIPDATA■■■■■■■“■■”■■■■■■■■
   CS_ERR_X86_ATT,  ///< X86 AT&T ■■■■■■(■■■■■■■)
   CS_ERR_X86_INTEL, ///< X86 Intel ■■■■■■(■■■■■■■)
   CS_ERR_X86_MASM, ///< X86 Intel ■■■■■■(■■■■■■■)
} cs_err;
```

本文下一部分将分析Capstone API，敬请期待

上一篇：Linux kernel Expl... 下一篇：关于对antSword(蚁剑)进行...

1. 0 条回复
   - 动动手指，沙发就是你的了！

目录