用 C/C++编写 IDA 插件

[1.0 版]

steve@binarypool.com

内容目录

第一章 入门

1.1 为什么会有这本手册?

花了大量时间在IDA SDK中，来阅读那些头文件，以及学习别人的插件源代码后，我觉得应该有一个更简单的方法来开始IDA插件编写。尽管这些头文件中的注释十分翔实，但我发现这样浏览和搜索这些注释有点困难，因为我需要它们时，并不想通过大劳动量的搜索。我想我该写这样一本手册，来帮助那些希望开始学习插件开发的朋友。因此，我决定用一个篇章来介绍如何配置开发环境，让您能更快速地入门。

1.2 涵盖的内容

这本手册将引导您开始编写IDA插件，首先将介绍SDK，然后是介绍在多个平台下，配置插件开发环境。您将得到如何使用各种类和结构的经验，接下来是介绍一些作用广泛的SDK导出函数。最后，我将介绍如何使用IDA API来完成基本的任务，例如，用循环来分析函数，钩挂调试器和操作IDA数据库文件（IDB文件）。当您读完后，您应该能运用自己的知识来编写您自己的插件，希望您能通过社区把您的插件公布出来。

1.3 不包括的内容

尽管IDA的标准版和高级版支持许多其他的平台，但我主要关注于x86平台，因为我在这平台上面有最多的经验。因此，如果您需要全面掌握所有的IDA函数，我建议您去看看其他的那些头文件。

这本手册主要介绍的是“只读(read only)”函数，而不大介绍其他的函数，如添加注释，错误校验，定义结构等等函数。SDK资料中的种类很庞大，不介绍这些函数，是想让手册体积适中。

我开始想介绍netnodes的概念，可是因为IDA SDK的结构和类的成员很复杂，而且还有很多特殊原因，您知道的，手册不会包含一切。如果您确实需要这些知识，请您写信告诉我，可能会在下一个手册版本中来介绍这些，如果没别的特殊原因的话。

1.4 前置知识

首先最重要的是，您应该掌握如何使用IDA，这样您就能够舒服地浏览反汇编代码，以及配置调试器。还有，您应该准备C/C++语言的知识，最好还有x86汇编语言。在这里，C++是非常重要的，因为SDK有相当多的C++代码。如果您对C++不熟悉，但很精通C，您应该至少理解OOP的概念，如类，对象，方法以及继承。

1.5 居家旅行必备良药

编写、运行IDA插件，您需要IDA pro反汇编器4.8版或4.9版，还有IDA SDK（您可以从 http://www.datarescue.com处获得，但需要IDA授权许可），以及一个C/C++编译器，象Visual Studio，GCC平台，Borland系列，或其他的。

请注意，4.9版中所做的一些改变，已经被写进手册了。而且，对于4.9版，SDK是稳定的，4.9版的一些函数将不会再改变，也就是说，给4.9版写的插件（通常是二进制形式）也可以在以后的版本中正常工作。

1.6 C/C++之外的选择

如果您对C也不感冒，那么可以看看IDAPython，它是一个函数集，用高级语言Python封装了所有C++ API。要获取更多详细资料，请去 http://d-home.net/idapython 。还有一份使用IDApython的手册在 http://dkbza.org/idapython_intro.html，里面有很多详尽的介绍，作者是Ero Carrera。

还有一份介绍使用VB6和C#编写IDA插件的文章，请登陆：http://www.openrce.org/articles/full_view/13。

1.7 关于这本手册

如果您有任何问题、建议或您发现一些错误，请您告诉我，Steve Micallef，邮箱是 steve@binarypool.com。如果您真的从手册中读到对您有帮助的内容，我仍然会写信感谢您，这么做是非常值得的。

因为SDK会不断“长胖”，所以，这本手册也会适时的升级。您将从这里 http://www.binarypool.com/idapluginwriting/ 处获得最新版本的手册拷贝。

1.8 致谢

我必须感谢下面列出的朋友，他们对本手册提供了，审校、鼓励以及反馈，下列牛人排名不分顺序：

Iifak Guilfanov，Pierre Vandevenne，Eric Landuyt，Vitaly Osipov，Sccott Madison，Andrew Griffiths，Thorsten Schneider和Pedram Amini。

1.9 其他资料

在手册的编写过程中，参考了一份关于IDA插件的文档，该文档介绍了如何使用4.9版的通用脱壳插件，其中包括如何编写这种插件，以及如何运行插件。它可以在：http://www.datarescue.com/idabase/unpack_pe/unpacking.pdf 处被找到。如果您对编写插件很积极，您可以去Datarescue的论坛去寻求帮助(http://www.datarescue.com/cig-local/ultimatebb.cgi)，当没有官方支持的时候，您可以向Datarescue的人（或者IDA老手）求助，他们会乐意帮助您。

另一个非常棒的地方是 http://www.openrce.org， 在那儿，您将不仅仅找到很多逆向工程方面的好文章，还有工具，插件以及文档。那儿还有牛人在论坛里，他们将尽可能帮您解决IDA或者一般的逆向工程问题。

第二章 IDA SDK全局组织

IDA是一款非常好的反汇编器，而且最近还发布了一个调试器。IDA单独实现了很多了不起的功能，可是有时候，您可能需要实现一些IDA并没提供的功能，比如自动化或者做一些特殊的任务。另人欣慰的是，发布IDA SDK的Dataresuce公司的那些哥们儿，提供了一些接口，供您自己扩展IDA的功能。

下面是您能够用IDA SDK编写的四种类型模块，插件模块将是本手册的主题：

模块类型	作用
处理器	增加对不同处理器架构的支持，也被称做 IDP（IDA Processor）模块。
插件	扩展 IDA 功能。
文件加载器	增加对不同的可执行文件格式的支持。
调试器	在不同平台下，增强与其他调试器（或远程调试）交互的调试能力。

上面的“插件(plug-in)”术语将代替“插件模块(plug-in module)”，除非有特别说明。

IDA SDK 包含了您需要编写 IDA 插件的所有头文件和库文件。还支持 Linux 和 Win 平台下很多的编译器，而且还给出了一个插件例子的源代码，用来示范一些简单的功能。

不管您是一位逆向工程师、漏洞研究员、病毒分析员，或是身兼以上数职，SDK 都提供了一个格外强大灵活的平台。您差不多都能用它编写您自己的调试器/反汇编器，这当然是另人满意的。下面保守地列出了一些您能够用 SDK 做的事情：

- 自动分析和脱壳。
- 自动搜索被使用的函数（比如， LoadLibrary(),strcpy()，和您想到的其他函数）。
- 分析程序数据流，寻找您感兴趣的东西。
- 二进制文件比对。
- 编写一个反编译器。
- 还有其他的功能……

查看别的朋友用IDA SDK编写的插件代码，请登陆IDA Palace网站，地址是 http://home.arcor.de/idapalace/。

2.1 安装方法

很简单的。当您得到了 SDK(一般都应该是.zip 格式的文件)，解压缩它到您选择的目录。我通常是在 IDA 的安装目录下，建立一个 sdk 目录，把所有的玩意儿都放那儿，但这不是很要紧的，您也可以按您的想法办。

2.2 目录结构

我将探讨一些关于插件编写的内容，以及讨论它们的实质，但不涉及所有 SDK 的内容。

目录	内容
/	此根目录下有一些不同平台下的 makefile 编译配置文件，以及您应该首先阅读的 readme.txt，特别是当版本有变化的时候。
include/	此目录下包括：以功能分类的头文件。我建议仔细查看这些头文件，并且纪录一下这些函数，当您看完本手册后，便可以应用这些函数。
libbor.wXX/	用于 Borland C 编译时，所使用的 IDA 库文件。
libgccXX.lnx/	Linux 下的 GCC 编译时，要用到的 IDA 库文件。
libgcc.wXX/	Windows 下，GCC 编译时，所使用的 IDA 库文件。
libvc.wXX/	Windows 下，Visual C++编译时，要用到的 IDA 库文件。
plugins/	插件例子代码。

XX 表示 32 位或者 64 位，这要看您所运行的平台架构。

2.3 头文件介绍

在 include 目录中的五十多个头文件中，我发现很多常用于编写插件的头文件。如果您需要所有头文件中的说明信息，可以看看 SDK 根目录下的 readme.txt 说明文档，或者是头文件中自带的说明。下面这个列表只是简单描述了它们的大致功能，更详细的介绍请参阅以后的章节。

文件	内容
area.hpp	文件包括：area_t 和 areacb_t 类，他们表示代码中的“域（areas)”，稍后将详细介绍“域”的概念。
bytes.hpp	反汇编文件中，处理字节的函数和一些定义。
dbg.hpp & idd.hpp	这些文件中包括：调试器类和函数。
diskio.hpp & fpro.h	这些文件包括 IDA 自己的 fopen()，open()，等等。以及一些其他函数。还有文件操作函数（获取硬盘剩余空间，当前工作目录，等等)。
entry.hpp	获取和操作执行文件的入口点(entry point)信息的相关函数。
frame.hpp	处理堆栈、函数帧、局部变量以及地址标志的函数。
funcs.hpp	funcs_t 类和许多与函数相关的东西。
ida.hpp	idainfo 结构，它包含被反汇编的文件的许多重要信息。

文件	内容
kernwin.hpp	用于跟 IDA 界面进行交互的函数和类。
lines.hpp	相关的函数和定义，用来处理反汇编文本、代码着色，等等。
loader.hpp	加载或操作 IDB 文件的一些函数。
name.hpp	获取或者设置名称的函数和定义（例如局部变量，函数名，等等）。
pro.hpp	所有其他的函数的定义。
search.hpp	各种函数和定义，用来搜索反汇编文件中的文本，数据，代码等等。
segment.hpp	segment_t 类和所有处理二进制文件中的段(区块)的函数。
strlist.hpp	string_info_t 结构和用来获取 IDA 的字符串列表的一些函数。
ua.hpp	insn_t，op_t 和 optype_t 类分别表示指令、操作数与操作数类型，以及与 IDA 分析器一同运作的函数。
xref.hpp	处理交叉参考引用代码，和数据参考引用的函数。

2.4 使用 SDK

大致来说，头文件中的任何您能使用的函数都有一个前缀：ida_export，而全局变量则冠以 ida_export_data 前缀。这点规矩是为了与底层函数(在头文件中也有定义)保持安全距离，并且提供了一些封装好的高层函数。您还可以使用任何定义好的类、结构和枚举。

第三章 配置一个编译环境

Borland 用户注意事项：IDA SDK 所支持的编译器中，本节将不讨论 Borland 公司的编译器。您应该阅读 SDK 根目录下 install_cb.txt 与 makeenv_br.mak 以决定需要用的编译器和连接标志。

开始编程之路前，最好是有一个合适的环境，以使得开发过程简单容易。一般流行的环境都会被介绍，如果您不是一般流行环境，比如您是 ENIAC 的忠实用户，那么我只能说抱歉了。如果您已经配置好了开发环境，请从容地跳到下一章。

3.1 Windows 下使用 Visual Studio

这里将用 Visual Studio .NET 2003 做示范，一般也适用于这以后的版本，可能也适应以前的版本。

一旦您的 Visual Studio 开始运行，请您关闭您可能打开的任何一个工程或方案，我们需要一个完全干净的状态。

1	到菜单,文件->新建->项目...(Ctrl-Shift-N)
2	展开“Visual C++项目”，在“Win32”子目录下，选择“Win32 项目”图标，给项目取一个您喜欢的名称，点击确定。
3	“Win32 应用程序向导”应该会出现，单击“应用程序设置”标签，而且确保“Windows 应用程序”被选定，再单击“空项目”复选框，点击“完成”。
4	在右侧的“解决方案浏览器”中，右键单击“源文件”目录，再依次 添加->添加新项...
5	选择 C++文件(.cpp)图标，给文件一个恰当的名称。单击打开。您要添加其他文件到此项目的话，就重复这个步骤。
6	到菜单“项目”-> “项目名”（刚才您输入的名称） 属性...
7	改变如下设置（一些是减小插件尺寸，因为 VS 的输出文件有些大）： 配置属性->常规：更改“配置类型”为动态链接库(.dll) C/C++->常规：设置“检测 64 位可移植性问题”为否 C/C++->常规：设置“调试信息格式”为禁用 C/C++->常规：添加 SDK 的 include 路径到“附加包含目录”一栏。比如 C:\IDA\SDK\Include C/C++->预处理器：添加__NT__；__IDP__到“预处理器定义” C/C++->代码生成：关掉“缓冲区安全检查”和“基本运行时检查”，设置“运行时库”为单线程 C/C++->高级：“调用约定”为__stdcall 连接器->常规：把“输出文件”的.exe 后缀改为.plw，并让它生成到 IDA 的插件目录中。 连接器->常规：添加您的 libvc.w32 路径到“附加库目录”。比如 C:\IDA\SDK\libvc.w32 连接器->输入：添加 ida.lib 到“附加依赖项” 连接器->调试：禁用“生成调试信息”

	连机器->命令行：添加/EXPORT:PLUGIN 生成事件->生成后事件：设置“命令行”为您的 idag.exe，这样每次编译成功都会自动启动 IDA（可选） 单击 确定
8	回到第 6 步，但在第 7 步前，把“配置”从“活动(Debug)”改为 Release，然后再重复更改第 7 步的设置。单击 确定
9	跳到 3.4 章节

3.2 Windows 下使用 GCC 的 Dev-C++和 MinGW

您可以从http://www.bloodshed.net/dev/devcpp.html获取Dev-C++，GCC和MinGW的一个集合软件包。先安装并设置好，现在假设一切都能正常工作。

同上，启动Dev-C++，并确保没有其他的工程文件被打开，我们需要一个纯净的状态。

1	到菜单的 文件->新建->工程，选择Empty Project，确保 C++工程被选定，然后指定一个合适的名称，单击确定。
2	选择一个目录保存工程文件，任何地方都可以。
3	到菜单的 工程->新建单元，这样就会保存源代码到您的插件。如果您需要添加其他的文件到工程，就重复这个步骤。
4	到菜单的 工程->工程属性，单击“参数”标签
5	在C++编译器下，添加： -DWIN32 -D__NT__ -D__IDP__ -v -mrtd
6	在连接器下，添加： ../path/to/your/sdk/libgcc.w32/ida.a -Wl,--dll -shared 注意，在路径的开始，最好添加../，因为msys好像是拒绝单独的 / 斜杠，再设法从msys目录的根路径引用它。
7	单击“文件/目录”tab，以及sub-tab中的“包含文件目录”，添加您的IDA SDK include目录的路径到该列表中。
8	单击“Build选项”tab，设置“可执行文件输出目录”为您的IDA plugins目录，然后“覆盖同名的文件”一栏中改为.plw文件。单击确定。
9	跳到3.4章节

3.3 Linux下使用GCC

Windows下的插件是.plw扩展名，Linux插件有所有不同，是以.plx为扩展名。因此，在示例中，没有GUI IDE，所以，不再是一步步的配置过程，我将只用Makefile来做示范，下面的例子中，可能并不是一个最纯净的Makefile，但它应该能工作。

在这个示例中，IDA安装于/usr/local/idaadv，SDK位于sdk子目录。把下面的Makefile放到您插件的源代码目录。

设置SRC为您的插件包含的源代码，以及OBJS为将要被编译的目标文件（相同文件名，仅仅是换了一个扩展名为.o）

```
SRC=file1.cpp file2.cpp
OBJS=file1.o file2.o
CC=g++
LD=g++
CFLAGS=-D__IDP__ -D__PLUGIN__ -c -D__LINUX__ \
-I/usr/local/idaadv/sdk/include $(SRC)
LDFLAGS=--shared $(OBJS) -L/usr/local/idaadv -lida \
--no-undefined -Wl,--version-script=./plugin.script
all:
$(CC) $(CFLAGS)
$(LD) $(LDFLAGS) -o myplugin.plx
cp myplugin.plx /usr/local/idaadv/plugins
```

要编译您的插件，make程序将完成这项任务，再为您复制插件到IDA plugins目录。

3.4 一份插件模板

IDA“钩挂(hooks in)”到您的插件是通过PLUGIN类来实现的，而且一般来说，是唯一需要被导出的一个对象（因而能被IDA使用）。还有，您应该通过#include预编译指令，来包含一些基本的头文件，例如，ida.hpp,idp.hpp和loader.hpp。

下面的模板，可以作为您开始学习插件的一个开始。如果您将它粘贴到您的开发环境的一个文件里，它应该能编译，而且当在IDA里运行它的时候（Edit->Plugins->pluginname,或者直接按下定义的热键），它将在IDA的日志窗口中插入“Hello World”文本。

```
#include <ida.hpp>
#include <idp.hpp>
#include <loader.hpp>
int IDAP_init(void)
{
    //在这里做一些校验，以确保您的插件是被用在合适的环境里。
}
void IDAP_term(void)
{
    //当结束插件时，一般您可以在此添加一点任务清理的代码。
    return;
}
// 插件可以从plugins.cfg文件中，被传进一个整型参数。
// 当按下不同的热键或者菜单时，您需要一个插件做不同
// 的事情时，这非常有用。
void IDAP_run(int arg)
{
  // 插件的实体
```

```
  msg("Hello world!");
  return;
}
// 这些不太重要，但我还是设置了。
char IDAP_comment[] = "This is my test plug-in";
char IDAP_help[] = "My plugin";
// 在Edit->Plugins 菜单中，插件的现实名称，
// 它能被用户的plugins.cfg文件改写
char IDAP_name[] = "My plugin";
// 启动插件的热键
char IDAP_hotkey[] = "Alt-X";
// 所有PLUGIN对象导出的重要属性。
plugin_t PLUGIN =
{
  IDP_INTERFACE_VERSION,     // IDA version plug-in is written for
  0,                         // Flags (see below)
  IDAP_init,                 // Initialisation function
  IDAP_term,                 // Clean-up function
  IDAP_run,                  // Main plug-in body
  IDAP_comment,              // Comment  - unused
  IDAP_help,                 // As above  - unused
  IDAP_name,                 // Plug-in name shown in
                             // Edit->Plugins menu
  IDAP_hotkey                // Hot key to run the plug-in
};
```

在PLUGIN结构中，您通常并不能逃避设置这些标志属性（从上数第二个标志），除非它是一个调试器模块，或者您想在Edit->Plugins菜单中隐藏点什么。您可能需要设置别的标志，请参看loader.hpp中的更多的有用信息。

上面的插件模板也可以从如下地址下载，
http://www.binarypool.com/idapluginwriting/template.cpp.

3.5 配置及运行插件

这是所有步骤中最简单的 -- 复制编译好的插件（确定它在Windows下的扩展名为.plw，Linux下为.plx）到IDA的plugins目录，然后IDA将自动加载它。

开始的时候，检查正确无误地检查您的编译环境（比如Linux下的LD_LIBRARY_PATH），确保您的插件能加载所有应该加载的DLL和共享库。您可以用-z20参数来启动IDA，这表示允许插件来进行调试。如果在加载的过程中，有什么错误的话，通常会显示出来。

如果您在IDAP_init()函数中添加了代码，这些代码将在IDA反汇编处理第一个文件时，得以执行，另外，如果您在IDAP_run()函数中添加了代码，它们将在用户按下热键组合，或通过选择Edit->Plugins菜单时，得以运行。

用户可以改写plugins.cfg文件中的一些PLUGIN的设置，比如插件名和热键，但是这对于您来说也没什么要紧的。在插件开始运行时，plugins.cfg文件也可以被用来传递参数到您的插件里面。

第四章 IDA插件原理

IDA SDK中有各式各样的类，数据结构和类型，其中有一些用得更广泛。这一章的目的就是向您介绍它们，因为它们提供了很棒的视角，来分析在一个反汇编文件中，IDA到底知道多少信息，而且，也可以让您思考，能用SDK做哪些可能的事情。

一些类和结构都非常大，都有很多的成员变量和方法或函数。在这一章，大多是介绍这些变量，而函数将在《第五章-函数》中介绍。有一些代码的注释是从SDK中获得的，还有一些是我自己写的注释，另外一些是兼有前两者。在一些情况下，#define预编译指令已经定义了一些成员，同样是在SDK中实现的。我囊括了它们，因为这样能更好的演示，成员变量可以设置什么样的有效值。

关于例子代码的重要事情：本章中的所有例子代码，都应该放到 3.4 章节中的插件模板的 IDAP_run()函数中，除非有特殊的情况。

4.1 重要的数据类型

贯穿 SDK 和这本手册，下面的数据类型都在被使用，因此，您能够非常清楚地明白，这些数据类型表示的是什么就非常重要了。

下面所有的类型，都是无符号长整型(unsigned long intergers)，和 64 位系统下的，无符号超长整型（unsigned long long integers)。它们被定义在 pro.h 头文件中。

类型	描述
ea_t	‘有效地址(Effective Address)’的意思，表示 IDA 中很多不同类型的地址（如内存，文件，limits 等等）
sel_t	段选择子，如，代码，栈和数据段选择子
uval_t	被用来表示无符号值
asize_t	通常用来表示某些东西的尺寸，例如一块内存。

下面这些是有符号长整型，和64为系统下的有符号超长整型。它们同样都被定义在pro.h头文件中。

类型	描述
sval_t	用来表示有符号值
adiff_t	表示两个地址间的不同处

最后，还有一些有意义的预定义值；其中之一是 BADADDR，它表示一个无效或不存在的地址，比如您会看到它被用来检测，一个可读地址区域或结构的末尾。同样，在字符缓冲区定义中的 MAXSTR 宏，它的值是 1024.

4.2 核心结构以及类

4.2.1 元数据信息(Meta Information)

idainfo 结构，事实上是存储在 IDA 数据库中（即 IDB 文件），注意，我提到的元数据信息是指，第一个文件被 IDA 加载反汇编后，以后无论再多的文件被加载，元数据信息都不会改变。下面是在 ida.hpp 头文件里，其中一些比较有意思的部分：

```
struct idainfo
{
...
char procName[8];    // IDA所运行的架构芯片
                     // (比如"metapc" = x86)
ushort filetype;     // 被反汇编的文件类型. 参看
                     // filetype_t 枚举 - 可以是 f_ELF,
                     // f_PE, 等等.
ea_t startSP;        // 程序开始运行时,
                     // [E]SP 寄存器的值
ea_t startIP;        // 程序开始运行之初,
                     // [E]IP 寄存器的值
ea_t beginEA;        // 程序入口点的线性地址,
                     // 一般和 startIP 相同
ea_t minEA;          // 程序的最小线性地址
ea_t maxEA;          // 程序的最大线性地址
                     // 不包含 maxEA
...
};
```

inf是上面这个结构的全局性实例。您会经常看见一个插件的初始化函数对inf.procName检查，以明确插件是为何种系统平台所编写。

比方说，如果您要编写的插件，仅处理x86平台下的PE和ELF二进制格式文件，您可以添加如下代码到您的插件的初始化函数（即3.4章节插件模板中的IDAP_init函数）。

```
// "metapc" 表示 x86 平台
if(strncmp(inf.procName, "metapc", 8) != 0
    || inf.filetype != f_ELF && inf.filetype != f_PE))
{
    error("Only PE and ELF binary type compiled for the x86 "
          "platform is supported, sorry.");
    return PLUGIN_SKIP; // 返回 PLUGIN_SKIP 意味着插件
                        // 不会被载入
}
return PLUGIN_KEEP;     // 继续此插件的加载
```

4.2.2 域的概念

在探究“高层”类的技术细节前，如段（segment），函数和指令的处理，我们应该了解一下两个关键概念：名为 areas(域)和 area control blocks（域控制块）。

4.2.2.1 area_t 结构

域以 area_t 结构表示，在 area.hpp 头文件中定义。基于这个头文件中的注释，严格来讲：

“域”由很多单独的 area_t 实例构成。域是一个连续的非空地址范围（由它的起始和结束地址指定，但不包括结束地址在内），还有地址范围的属性，也是域的内容。比如，一组段是一些域的集合。

请看下面摘录的一部分 area_t 结构的定义，它包括一个起始地址(startEA)和结束地址（endEA）成员。还有很多成员函数，比如判断一个域是否包括某一个地址，某一个域是否为空，以及返回一个域的尺寸。段是一个域，而函数也是，这就意味着，域可以包含另外的域。

```
struct area_t
{
    ...
    ea_t startEA;
    ea_t endEA; // 域不包括结束地址
    bool contains(ea_t ea) const { return startEA <= ea && endEA > ea; }
    bool empty(void) const { return startEA >= endEA; }
    asize_t size(void) const { return endEA - startEA; }
    ...
};
```

一般来说，函数和段都是域，这也表明，func_t和segment_t类继承了area_t结构。这就意味着，area_t结构中的成员变量和函数都能应用到func_t和segment_t(比如,segment_t.startEA和func_t.contains()都是合理的)。func_t和segment_t也扩展了area_t结构，还增加了自身的特殊成员变量和函数。稍后仍将介绍它们。

还有一些其他的基于area_t的继承，如下：

类型（和所在文件）	描述
hidden_area_t (bytes.hpp)	代码或数据被替换成的隐藏域，它以一个描述作为摘要，并能被展开查看隐藏的信息。
regvar_t (frame.hpp)	被用户自定义所替换的寄存器名称(寄存器变量)
memory_info_t (idd.hpp)	一块内存的相关信息(当使用调试器时)
segreg_t (srarea.hpp)	段寄存器(在x86平台是，CS,SS,等等）信息

4.2.2.2 areacb_t类

域控制块(Area Control Block)由areacb_t类表示，也定义于area.hpp头文件中。如下是对它的注释，稍微描述得简单，但事实上并不是非常需要。

“areacb_t”是一个基类，在IDA的很多地方都被用到。

域控制块的类很简单，由一些函数构成一个集合，这些函数可以被用来操作域。函数包括get_area_qty()，get_next_area()等等。您可能发觉自己可能并不需要直接使用这些函数，比如您反汇编分析二进制文件中的一个函数时，您可能更喜欢使用func_t的成员函数，或使用继承于area_t的其他的类。

有两个areacb_t类的全局实例，名为segs(segment.hpp中定义)和funcs(funcs.hpp中定义)，很明显，在当前的反汇编文件中，它们表示所有的段和函数。您可以使用下面的代码获取段和函数的数量。

```
#include <segment.hpp>
#include <funcs.hpp>

msg("Segments: %d, Functions: %d\n",
     segs.get_area_qty(),
     funcs.get_area_qty());
```

4.2.3 段和函数

前面提到过，segment_t和func_t类都是继承了或扩展了area_t结构,这意味着area_t的成员变量和函数都能在这些类中被使用,另外它们自身还实现了更多于area_t结构的功能。

4.2.3.1 段

segment_t类在segment.hpp中被定义。其中还有一些有趣的东西。

```
class segment_t : public area_t
{
public:
uchar perm;                 // 段访问权限（0表示没有内容）。可以是下面
                            // 的宏，或是宏的组合。
#define SEGPERM_EXEC 1      // 可执行
#define SEGPERM_WRITE 2     // 可写
#define SEGPERM_READ 4      // 可读
uchar type;                 // 段的类型。可以是下面的一个值。
#define SEG_NORM 0          // 未知类型，没有使用
#define SEG_XTRN 1          // 定义为‘extern’的段，
                            // 仅被允许包含非指令的内容
#define SEG_CODE 2          // 代码段
```

```
#define SEG_DATA 3          // 数据段
#define SEG_NULL 7          // 零长度的段
#define SEG_BSS 9           // 未初始化的段
...
}
```

SEG_XTRN 是一个特殊的（即非实际存在内容）段类型，由IDA建立，但是其他的类型，表示实际存在部分。对于一个被IDA载入的执行文件，例如.text区块的类型值为SEG_CODE，则perm成员的值为SEGPERM_EXEC | SEGPERM_READ。

在一个二进制文件中，要遍历所有的段，并在IDA的日志窗口中，打印段名和地址，您可以使用下面的代码。

```
#include <segment.hpp>
// 此代码只能在IDA 4.8中正常运行，因为get_segm_name()在4.9中已经改变
// 欲知详情，请阅读第五章
// get_segm_qty()返回加载文件中，段的总数量
for (int s = 0; s < get_segm_qty(); s++)
{
    // getnseg() 返回一个对应于提供的段序号的segment_t结构
    segment_t *curSeg = getnseg(s);
    // get_segm_name() 返回段名称
    // msg() 把信息打印到IDA的日志窗口
    msg("%s @ %a\n", get_segm_name(curSeg), curSeg->startEA);
}
```

4.2.3.2 函数

函数由 func_t 类来表示，该类被定义在 funcs.hpp 中，开始讨论 func_t 类的技术细节前，可能有必要弄清楚函数块(chunk)，函数源(parents)，以及函数尾(tail)的概念。

函数，是正在被分析的二进制文件中的，一些连续的代码块，通常被表示成一个单独的函数块。然而，很多时候，当优化性编译器移除一些冗余代码时，函数被分割成了很多含有代码的函数块，这是因为其他函数的隔离。这些被隔离的松散的函数块被叫做“函数尾(tail)”，还有一些函数块引用这些函数尾代码（由 JMP 或类似的指令引用），被称作“函数源(parents)”。有些容易混淆的是，所有的这些函数块，函数源，函数尾都是同一 func_t 类型，因此您需要检测 func_t 的成员 flags，以确定该 func_t 实例到底是函数尾还是函数源。

下面是 func_t 类的超级剪切版本，并附上一些 funcs.hpp 中的注释。

```
class func_t : public area_t
{
public:
...
    ushort flags;    // 表示函数类型的标志
```

```
                    // 下面是一些常用的标志:
#define FUNC_NORET      0x00000001L     // 函数并不返回
#define FUNC_LIB        0x00000004L     // 库函数
#define FUNC_HIDDEN     0x00000040L     // 一段隐藏函数块
#define FUNC_THUNK      0x00000080L     // 块(jump)函数
#define FUNC_TAIL       0x00008000L     // 一段函数尾
                                        // 其他标志都好理解（除了FUNC_HIDDEN）
    union // func_t要么表示整个函数块，要么表示一块函数尾
    {
        struct // 一个函数的整个块的属性
        {
            asize_t argsize;    // 返回之前，堆栈中要清除的字节数量
            ushort pntqty;      // 整个函数过程中，ESP寄存器被改变的次数
                                // (与PUSH, 等指令相关)
            int tailqty;        // 该函数自身含有的函数尾数量
            area_t *tails;      // 函数尾的数组，以ea排序
        }
        struct          // 函数尾的属性
        {
            ea_t owner; // 拥有该函数尾的main函数的地址
        }
...
}
```

因为函数同段一样，都是域，处理函数差不多和处理段一样。下面这段实例代码列出了所有函数名和它们在反汇编文件中的地址，并在 IDA 的日志窗口中显示结果。

```
#include <funcs.hpp>
// get_func_qty() 返回加载的文件中，函数的总数量。
for (int f = 0; f < get_func_qty(); f++)
{
    // getn_func() 返回由函数序号指定的func_t结构
    func_t *curFunc = getn_func(f);
    char funcName[MAXSTR];
    // get_func_name（）获取函数名，并存储到funcName
    get_func_name(curFunc->startEA,
                funcName,
                sizeof(funcName)-1);
    msg("%s:\t%a\n", funcName, curFunc->startEA);
}
```

4.2.4 代码的表示

通常，汇编语言指令由助记符（PUSH，SHR，CALL 等），以及操作数（EAX，[EBP+0xAh]，0x0Fh 等）组成。一些操作数可以有多种形式，而有一些指令则没有操作数。所有这些都在 IDA SDK 中清楚的写明了。

您可以从 insn_t 类型入手，它表示一整条指令，像“MOV EAX, 0x0A”这样的一整条指令。insn_t 由一些成员变量，6 个 op_t 变量（每一个对应指令中的一个操作数），而且每一个操作数，可以是一个特定的 optype_t 值（比如，通用寄存器，立即数，等等）。

现在，我们开始仔细地，探索其中的每一个部分，它们都定义在 ua.hpp。

4.2.4.1 操作数类型

optype_t 表示一条指令中的操作数类型。下面列举了一些常用的操作数类型。附带的一些描述取自 ua.hpp 中 optype_t 的定义。

操作数	描述	反汇编举例（操作数以粗体标出）
o_void	不含操作数	pusha
o_reg	通用寄存器	dec **eax**
o_mem	直接内存引用	mov eax, **ds:1001h**
o_phrase	间接内存引用[基址寄存器+偏移寄存器]	push dword ptr **[eax]**
o_displ	间接偏移内存引用[基址寄存器+偏移寄存器+偏移量]	push **[esp+8]**
o_imm	立即数	add **ebx, 10h**
o_near	立即近地址	call **_initterm**

4.2.4.2 操作数

op_t 表示一条指令中，某一个的操作数的相关信息。下面是 op_t 类的一部分剪切版。

```
class op_t
{
public:
    char n;          // 操作数序号或位置，(比如0,1,2)
    optype_t type;   // 操作数类型 (请看上一节的描述)
    ushort reg;      // 寄存器序号 (如果操作数类型是o_reg)
    uval_t value;    // 操作数的具体数值 (如果操作数类型是o_imm)
    ea_t addr;       // 指向操作数或被其使用的虚拟地址 (如果操作数类型是o_mem)
...
}
```

因此，举个例子来说，[esp+8]这样的操作数将返回 o_displ 这样的类型（即 type 成员的值为 o_displ），reg 成员的值为 4（正是 ESP 寄存器的序号）以及，addr 成员的值为 8，因为您正在访问堆栈指针指向的 8 字节，因此正是一个内存引用。您可以用下面的一段代码获取 IDA 中，您的光标所在的位置，那条指令的第一个操作数的 op_t 值：

```
#include <kernwin.hpp>
```

```
#include <ua.hpp>
// 反汇编当前光标所在位置的指令，
// 并使其存储到‘cmd’全局结构中。
ua_ana0(get_screen_ea());
// 显示第一个操作数的相关信息
msg("n = %d type = %d reg = %d value = %a addr = %a\n",
    cmd.Operands[0].n,
    cmd.Operands[0].type,
    cmd.Operands[0].reg,
    cmd.Operands[0].value,
    cmd.Operands[0].addr);
```

4.2.4.3 助记符

指令中的助记符（PUSH，MOV，等）由 insn_t 类（参考下一节）的 itype 成员表示。然而，itype 是一个整数，就目前而言，并不能在用户定义的数据结构中，直接显示指令的文本样式。但有一个 ua_mnem()函数可以实现上述功能，以后将在《第五章 - 函数》中介绍。

一个名为 instruc_t(allins.hpp)的枚举保存了所有的助记符(前缀为 NN_)。如果您知道您要寻找或测试的指令，您可以使用它，而不是使用文本表示的指令。比如，测试二进制文件中，某条指令的助记符是否为 PUSH，您可以这么做：

```
#include <ua.hpp>
#include <allins.hpp>
// 在入口点填充‘cmd’结构
ua_ana0(inf.startIP);
// 测试这条指令是否为PUSH指令
if (cmd.itype == NN_push)
    msg("First instruction is a PUSH");
else
    msg("First instruction isn't a PUSH");
return;
```

4.2.4.4 指令

insn_t 表示一整条指令。包括一个名叫 Operands 的 op_t 类型数组，表示指令中的所有操作数。当然，也有指令没有操作数的（象 PUSHA，CDQ，等指令），这种情况下，Operands[0]变量则为 optype_t 类型的 o_void 值（无操作数）。

```
class insn_t
{
public:
    ea_t cs;        // 代码段基址(in paragraphs)
    ea_t ip;        // 段中的偏移
    ea_t ea;        // 指令起始地址
    ushort itype;   // 助记符ID
```

```
    ushort size;    //  指令大小（字节）
#define UA_MAXOP 6
    op_t Operands[UA_MAXOP];
#define Op1 Operands[0] // 第一个操作数
#define Op2 Operands[1] // 第二个操作数
#define Op3 Operands[2] // ...
#define Op4 Operands[3]
#define Op5 Operands[4]
#define Op6 Operands[5]
};
```

有个 insn_t 结构类型的全局变量,名为 cmd，可由 ua_ana0()和 ua_code()函数填充。稍后将详细讨论，但目前，给出一个示例代码，它获取入口点的指令序号，地址和大小，并在 IDA 的日志窗口显示出来。

```
// ua_ana0()函数在指定的地址，填充cmd结构
ua_ana0(inf.beginEA); // or inf.startIP
msg("instruction number: %d, at %a is %d bytes in size.\n",
            cmd.itype, cmd.ea, cmd.size);
```

4.2.5 交叉引用参考

IDA 的一个很方便的功能就是交叉引用参考功能，它有助于让您知道反汇编文件中，所有部分引用其他地址的情况。举例来说，在 IDA 里，您可以在反汇编窗口中高亮选中一个函数，然后按下‘x’键，然后就会在弹出窗口里，显示所有引用该函数的其他地址（比如，调用此函数的地址）。对于数据和局部变量也可以用同样的方式进行引用参考。

SDK 提供了一个简单接口，来访问这些交叉引用参考的信息，这些信息以 B-tree 数据结构存储，并可以通过 xrefblk_t 结构来访问。此外，也有更手动化的方式，来获取这些信息，但和下面列举的方法比起来，实在是太慢了。

应该记住的重要事情是，当一条指令后续又有一条指令，IDA 会潜在地看作第一条指令引用第二条指令，但这项功能可以用 xrefblk_t 结构的一些方法关掉，以后将在《第五章 - 函数》中介绍。

4.2.5.1 xrefblk_t 结构

交叉引用参考功能的核心是 xrefblk_t 结构，它被定义在 xref.hpp 中。此结构首先需要使用 first_from()或 first_to()成员函数来填充（这看您是要寻找一个地址的引用到‘reference to’，或引用于‘reference from’）,然后您遍历引用的时候，就用 next_from()或 next_to()成员函数来填充。

下面列出的这个结构的成员变量和大部分注释来自 xref.hpp 头文件。成员函数（first_from，first_to，next_from 和 next_to）省略了，但是会在《第五章-函数》里讨论。

```
struct xrefblk_t
{
```

```
ea_t from;      // 被引用地址(referencing address)
ea_t to;        // 引用的地址(referenced address)
uchar iscode;   // 1表示代码参考引用，0表示数据参考引用
uchar type;     // cref_t 或者dref_t 类型中之一(参看
                //  4.2.5.2章节、和 4.2.5.3章节)
...
};
```

如 iscode 成员变量表示的一样，xrefblk_t 能获取代码参考引用或者数据参考引用的信息，其中的每一个都有可能的参考引用类型，且以 type 成员变量表示其类型。这些代码和数据参考引用类型在接下来两节里被阐述。

下面的代码片段，将给您当前光标所在的位置，相关的交叉参考引用信息：

```
#include <kernwin.hpp>
#include <xref.hpp>

xrefblk_t xb;
// 获取当前光标所在地址
ea_t addr = get_screen_ea();
// 循环遍历所有交叉引用
for (bool res = xb.first_to(addr, XREF_FAR); res; res = xb.next_to()) {
    msg("From: %a, To: %a\n", xb.from, xb.to);
    msg("Type: %d, IsCode: %d\n", xb.type, xb.iscode);
}
```

4.2.5.2 代码

下面是 cref_t 枚举类型，剪去了一些不相关的内容。对于参考引用的类型，如果 xrefblk_t 的 iscode 成员变量为 1 的话，那么 type 成员变量将会是下面列出来的枚举值。下面的注释取自 xref.hpp 头文件。

```
enum cref_t
{
...
fl_CF = 16,     // 远调用(Call Far)
                // 该xref在引用的地方创建一个函数
fl_CN,          // 近调用(Call Near)
                // 该xref在引用的地方创建一个函数
fl_JF,          // 远跳转(Call Far)
fl_JN,          // 近跳转(Call Near)
fl_F,           // 选择跳转：用来表示下一条指令的执行流程。
...
};
```

下面的一个代码参考引用取自一个简单的二进制文件，712D9BFE 被 712D9BF6 引用，即它是一个近跳转引用类型。

```
.text:712D9BF6 jz short loc_712D9BFE    //近跳转引用类型
...
.text:712D9BFE loc_712D9BFE:
.text:712D9BFE lea ecx, [ebp+var_14]
```

4.2.5.3 数据

如果xrefblk_t的iscode成员被置为0，说明它是一个数据参考引用。下面是当您在处理数据参考引用时，一些可能的type成员变量的值。此枚举类型的注释取自xref.hpp头文件。

```
enum dref_t
{
...
dr_O,        // 参考是一个偏移(Offset)
             // 参考引用使用数据的‘偏移’而不是它的值
             //      或者
             // 参考引用的出现是因为指令的“OFFSET”标志被设置
             // 这时，就意味着此类型基于IDP(IDA Processor)。
dr_W,        // 写访问(Write acess)
dr_R,        // 读访问(Read acess)
...
};
```

请记住，当您看见如下代码时，您实际上看到的是数据引用，因此 712D9BD9 在引用 712C119C：

```
.idata:712C119C extrn wsprintfA:dword
...
.text:712D9BD9 call ds:wsprintfA
```

这种情况下，xrefblk_t的type成员将是典型的dr_R值，因为是简单地读取了ds:wsprintfA这一行的地址。另外一种数据参考引用如下，在712EABE2处的PUSH指令引用了位于712C255C的一串字符。

```
.text:712C255C aVersion:
.text:712C255C unicode 0, <Version>,0
...
.text:712EABE2 push offset aVersion
```

这种情况，xrefblk_t的type成员变量将会是dr_O，因为它以偏移访问该数据。

4.3 字节标志

对于反汇编文件的每一个字节，IDA录入了一个相应的四字节（32位）值，并保存在id1文件中。这些个四字节中，每半字节（四位）是一个标志，表示反汇编文件的某个地址中，该字节的一条信息。反汇编文件的地址中，四字节的最后一个字节才是真正内容。

比方，下面这个被反汇编的文件中，一条指令占据一个单独字节(0x55)：

```
.text:010060FA push ebp
```

文件中的上述地址被反汇编为IDA标志就是，0x00010755；0001007是标志成分，而55则是文件中该地址的字节标志。请记住，地址在标志中并没有实际意义，也不可能从一个地址或字节本身得到标志，您应该使用getFlags()来获取一个地址的标志（详见下）。

很明显，不是所有的指令都是一个字节的尺寸；拿下面的指令做为例子，它有三个字节（0x83 0xEC 0x14）。因此，该指令分成三个地址；0x010011DE,0x010011DF和0x010011E0：

```
.text:010011DE sub esp, 14h
.text:010011E1 ...
```

下面是该指令的每个字节对应的标志：

```
010011DE: 41010783
010011DF: 001003EC
010011E0: 00100314
```

因为这三个字节同属一条指令，故该指令的第一个字节被称作“指令头(head)”，然后剩下两个字节叫做“指令尾（tail)”。再说下，每个标志的最后一个字节对应于指令（0x83,0xEC,0x14)。

所有标志定义在 bytes.hpp 头文件，然后您可以检验，getFlags(ea_t ea)的返回的标志与适当的标志检测函数的测试结果，来确定该标志到底是哪个标志。下面是一些常用的标志及其封装函数。一些函数将在《第五章-函数》中讨论，剩下的你可以在 bytes.hpp 头文件中查阅。

标志名	标志值	含义	封装函数
FF_CODE	0x00000600L	该字节是代码吗？	isCode()
FF_DATA	0x00000400L	该字节是数据吗？	isData()
FF_TAIL	0x00000200L	该字节是一条指令的以部分（非指令头）吗？	isTail()
FF_UNK	0x00000000L	IDA 能分辨该字节吗？	isUnknown()
FF_COMM	0x00000800L	该字节被注释了吗？	has_cmt()

标志名	标志值	含义	封装函数
FF_REF	0x00001000L	该字节被别的地方引用吗?	hasRef()
FF_NAME	0x00004000L	该字节有名称吗?	has_name()
FF_FLOW	0x00010000L	上条指令是否流过这里?	isFlow()

回到开始的“push ebp”例子，如果我们用上面的两个标志来手动检测 getFlags(0x010060FA)的返回值，我们会得到下面的结果：

```
0x00010755 & 0x00000600 (FF_CODE) = 0x00000600. 由此，我们知道这是一条指令。
0x00010755 & 0x00000800 (FF_COMM) = 0x00000000. 我们知道这没有被注释。
```

上面的例子是纯粹的演示目的，请别再您的插件中用这样的方法。上面提到过，您可能经常要使用助手函数(helper function)，来检测一个标志到底是哪个。下面的代码将返回您的光标所在行，指定的头地址(head address)标志。

```
#include <bytes.hpp>
#include <kernwin.hpp>
msg("%08x\n", getFlags(get_screen_ea()));
```

4.4 调试器

IDA SDK 的一个很强大的新特性是，可以与 IDA 的调试器交互，而且除非您已经安装了您自己的调试器插件，那么，IDA 会使用一个自带的调试器插件。下面是 IDA 自带的调试器插件，也可以在您的 IDA plugins 目录中被找到：

插件文件名	描述
win32_user.plw	**Windows 本机调试器**
win32_stub.plw	**Windows 远程调试器**
linux_user.plw	**Linux 本机调试器**
linux_stub.plw	**Linux 远程调试器**

这些调试器被 IDA 自动加载，而且会在 Debugger->Run 菜单显示。打今儿起，“调试器”一词，将表示，您正在使用的调试器（IDA 将自动选择一个最适合您的）。

先前提到，为 IDA 写一个调试器模块是可以的，但这也不是说，就拒绝使用编写的插件模块来与调试器交互。下面描述的是插件的第二种类型。

此外，所有与调试器交互的接口函数，将在《第五章-函数》中被讨论，在深入讨论前，还有一些关键的数据结构和类需要去认识。

4.4.1 debugger_t 结构

定义在 idd.hpp 头文件中的 debugger_t 结构，导出了一个 dbg 指针，表示当前激活的调试器插件，并且，当调试器被加载时，该指针就是有效的了。（比如，在 IDA 启动时，而不仅仅是您抄起调试器的时候）。

```
struct debugger_t
{
  ...
  char *name; // 类似‘win32’或‘linux’的调试器短名称
#define DEBUGGER_ID_X86_IA32_WIN32_USER 0 // win32用户态进程
#define DEBUGGER_ID_X86_IA32_LINUX_USER 1 // linux用户态进程
  register_info_t *registers; // 寄存器数组
  int registers_size; // 寄存器个数
  ...
}
```

作为插件模块，可能您会需要访问 name 指针成员变量，来测试您的插件与哪个调试器交互。registers 指针和 registers_size 成员变量获取一些可用寄存器的时候，也很有用处（请看下集……）。

4.4.2 寄存器

当使用调试器的时候，通常的任务就是访问及操作寄存器值。IDA SDK 里，寄存器以 register_info_t 结构来描述，保存寄存器的值由 regval_t 结构表示。下面是部分摘取自 idd.hpp 头文件中的 register_info_t 结构定义。

```
struct register_info_t
{
  const char *name;                  // 寄存器全名（EBX，等）
  ulong flags;  // 寄存器特性
                // 可以是下面值的组合
#define REGISTER_READONLY 0x0001     // 用户不能修改该寄存器的当前值
#define REGISTER_IP 0x0002           // 指令指针
#define REGISTER_SP 0x0004           // 栈顶指针
#define REGISTER_FP 0x0008           // 栈帧指针
#define REGISTER_ADDRESS 0x0010      // 寄存器可以包含地址
...
}
```

这个结构的唯一实例，可以用*dbg（SDK 导出的一个 debugger_t 实例）的数组成员*register 来访问，因此知道您使用调试器时，在您的系统上，这些寄存器才是有效的。

要获取寄存器的值，最起码调试器得运行起来。读取或操作调试器值的函数将在《第五章-函数》中详细讨论，现今，您需要知道的事，就是要获取这些值，可以通过 regval_t 的成员 ival，或者，如果您正在处理浮点数，那么您可以使用 fval 成员。

下面是定义在 idd.hpp 头文件中的 regval_t 结构。

```
struct regval_t
{
```

```
    ulonglong ival;     // 整数值
    ushort    fval[6];  // 浮点数值
                        // 表示方法参看 ieee.h 头文件
};
```

ival/fval 将直接对应于一个寄存器里存储的东西，因此，如果 EBX 寄存器为 0xDEADBEEF，ival 成员(一旦用 get_reg_val()函数填充后)，同样将是 0xDEADBEEF。

下面的例子将循环遍历所有有效寄存器，并显示每个寄存器的值。但如果您没祭起调试器，而运行这段代码，那么值将会是 0xFFFFFFFF：

```
#include <dbg.hpp>
// 循环遍历所有寄存器
for (int i = 0; i < dbg->registers_size; i++) {
    regval_t val;
    // 获取寄存器中存储的值
    get_reg_val((dbg->registers+i)->name, &val);
    msg("%s: %08a\n", (dbg->registers+i)->name, val.ival);
}
```

4.4.3 断点

调试的一个最核心的部分就是断点，而且 IDA 用 dbg.hpp 中定义的 bpt_t 结构（见下），来表示不同的硬件和软件断点。硬件调试器是用 CPU 的调试寄存器（x86 的 DR0-DR3），但是软件调试器则是在需要中断的地址处，插入 INT 3 指令来达到目的。尽管这些都由 IDA 来为您处理好了，但知道其中的不同是有帮助的。在 x86 平台，您最多能设置 4 个硬件断点。

```
    struct bpt_t
    {
      // 只读属性：
      ea_t ea;                  // 断点的起始地址
      asize_t size;             // 断点的尺寸
                                // (若是软件断点，则是未定义的)
      bpttype_t type;           // 断点的类型:
    // 摘自 idd.hpp 中 bpttype_t 常量定义:
    // BPT_EXEC  =  0,             // 可执行的指令
    // BPT_WRITE =  1,             // 可写
    // BPT_RDWR  =  3,             // 可读写
    // BPT_SOFT  =  4;             // 软件断点
      // 可修改属性 (使用 update_bpt()函数修改):
      int pass_count;               // 执行流达到此断点消耗的时间
                                    // (如果未定义则为-1)
      int flags;
    #define BPT_BRK   0x01          // 调试器停在这个断点吗?
```

```
#define BPT_TRACE 0x02          // 当到达这个断点时，
                                // 调试器添加了跟踪信息吗？
  char condition[MAXSTR];       // 一个 IDC 表达式，
                                // 它将被用来表示一个中断条件，
                                // 或者当这个断点被激活时，要执行
                                // 的 IDC 命令。
};
```

因此，如果 bpt_t 的 type 成员是 0，1 或者 3，则表示硬件断点，但是 4 表示软件断点。

有许多的函数可以创建，操作，读取该结构，但目前，我给出一个很简单的例子，它遍历所有的断点，并且在 IDA 的 Log 窗口显示到底是一个软件或硬件断点。这些函数将在以后详细解释。

```
#include <dbg.hpp>
// get_bpt_qty() 获取断点数目
for (int i = 0; i < get_bpt_qty(); i++) {
    bpt_t brkpnt;
    // getn_bpt 基于给定的断点数目，
    // 并用断点信息填充 bpt_t 结构
    getn_bpt(i, &brkpnt);
    // BPT_SOFT 就表示软件断点
    if (brkpnt.type == BPT_SOFT)
        msg("Software breakpoint found at %a\n", brkpnt.ea);
    else
        msg("Hardware breakpoint found at %a\n", brkpnt.ea);
}
```

4.4.4 跟踪

IDA 中，有三种类型的跟踪可供您打开；函数跟踪，指令跟踪和断点（又被称作可读/可写/可执行）跟踪。编写插件时，还有另一个形式的跟踪是有用的；单步跟踪。单步跟踪是跟踪的一个底层形式，允许您在该形式之上建立自己的跟踪机制，再利用事件通知（参看 4.5 章节）驱动您的插件，这样，每一条指令可以被单步执行。这是基于 CPU 的跟踪能力，而非用断点。

跟踪时，一个“跟踪事件（trace evnet）”就会产生，并保存到一个缓冲区，而且，您允许的跟踪类型，决定了引发产生什么样的跟踪事件。但是，单步跟踪不产生跟踪事件，这样就有些麻烦，但也可以用事件通知（参见 4.5 章节）替代。下面的表格列出了所有不同的跟踪事件类型，和在 dbg.hpp 中，对应的 tev_type_t 枚举值定义。

跟踪类型	事件类型 (tev_type_t)	描述
函数调用和返回	tev_call 和 tev_ret	函数已经被调用或者已经返回

指令	tev_insn	**指令已经被执行（在 IDA 内核中，这建立于单步跟踪之上）**
断点	tev_bpt	**设置的断点被激活。也被称作可读/可写/可执行跟踪**

所有跟踪事件都被保存到一个循环缓冲区，所以它不会填满，但如果是这个缓冲区太小，那么原来的跟踪事件可能会被覆盖掉。每个跟踪事件被表示为 tev_info_t 结构，它被定义在 dbg.hpp 头文件中。

```
struct tev_info_t
{
  tev_type_t  type; // 跟踪事件类型（上述表格中之一，或者 tev_none）
  thread_id_t tid;  // 被记录的事件所在线程。
  ea_t        ea;   // 发生事件的地址。
};
```

在 4.4.3 章节基于 bpt_t 结构的描述，一个断点跟踪和普通的断点差不多，不过断点跟踪在 flags 成员中有一个 BPT_TRACE 的标志。还有一点，condition 缓冲区成员可以有一个 IDC 命令，在断点被激活时，就可以执行 IDC 命令。

跟踪信息在进程运行时被填充，但仍然能在进程刚终止的时候被访问，而且可以是您返回到静态反汇编模式的时候（除非在退出的时候，您使用的插件已经明显地清除了那块缓冲区）。您可以使用如下代码，枚举所有跟踪事件（供您在执行调试的时候枚举）：

```
#include <dbg.hpp>
// 遍历所有跟踪事件
for (int i = 0; i < get_tev_qty(); i++) {
    regval_t esp;
    tev_info_t tev;

    // 获取跟踪事件信息
    get_tev_info(i, &tev);
    switch (tev.type) {
           case tev_ret:
                msg("Function return at %a\n", tev.ea);
                break;
           case tev_call:
                msg("Function called at %a\n", tev.ea);
                break;
           case tev_insn:
                msg("Instruction executed at %a\n", tev.ea);
                break;
           case tev_bpt:
                msg("Breakpoint with tracing hit at %a\n", tev.ea);
```

```
                break;
            default:
                msg("Unknown trace type..\n");
    }
}
```

目前，无需讨论的是，给插件增加入口是不必要的，甚至是修改跟踪事件日志。

所有这些函数将在《第五章-函数》中详细讨论。

4.4.5 进程和线程

IDA 给运行在调试器中的进程和线程，保存了一些它们的信息。进程和线程 ID 分别用 process_id_t 和 thread_id_t 类型标识，这两个类型都是有符号整数型。所有这些类型在 idd.hpp 中定义。还有另一个关于进程的类型，即 process_info_t 类型，如下：

```
struct process_info_t
{
  process_id_t pid;     // 进程 ID
  char name[MAXSTR];    // 进程名称 (执行文件名)
};
```

它们只有当二进制文件在 IDA 下面被调试执行的时候，才有用（比方说，您不能在静态反汇编模式下使用它们)。下面的例子演示了 process_info_t 结构的使用方法。

```
#include <dbg.hpp>
// 获取调试中的有效进程的数量
// get_process_qty() 也可以初始化 IDA 的“进程快照（process sanpshot)”
if (get_process_qty() > 0) {
    process_info_t pif;
    get_process_info(0, &pif);
    msg("ID: %d, Name: %s\n", pif.pid, pif.name);
} else {
    msg("No process running!\n");
}
```

使用这些结构的函数将在《第五章-函数》中讨论。

4.5 事件通知

一般来说，插件是同步运行的，即由用户来决定执行，要么通过按下热键，要么通过 Edit->Plugins 菜单。但是，插件也可以异步运行，就是响应 IDA 或者用户产生的事件通知来达到这个目的。

在 IDA 下工作的过程中，您可能经常点击按钮，执行搜索，等等。所有这些

动作都是“事件（events）”，所以，IDA 做的事情就是当这些动作发生时，产生“事件通知（event notifications）”。如果您的插件被配置成接收这些通知（稍后解释），您就可以编程实现做一些动作。打个比方，这样的程序可以保存一些宏，然后插件就能产生事件，驱使 IDA 执行各种功能。

4.5.1 接收通知

在 IDA 中接收事件通知，插件必须用 hook_to_notification_point()函数，注册一个回调函数(call-back)。要产生事件通知，可以使用 callui()函数，这些将在《第五章-函数》中详细讨论。

用 hook_to_notification_point()函数注册了一个回调函数后，您可以使用三种事件类型其中之一，这要看您需要接收什么样的事件通知。这些类型在 loader.hpp 中定义为 hook_type_t 枚举：

类型	从模块中接收事件通知	事件通知类型的枚举
HT_IDP	处理器模块	idp_notify(不会讨论)
HT_UI	IDA 用户界面	ui_notification_t
HT_DBG	正在运行的 IDA 调试器	dbg_notification_t

因此，要接收所有适合调试器的事件通知，并发送它们到您的 dbg_callback（打个比方）回调函数，您可以将如下代码放到 IDAP_init()函数中：

```
hook_to_notification_point(HT_DBG, dbg_callback, NULL);
```

第三个参数一般设置为 NULL，除非在收到一个通知时，您需要传递数据到回调函数（可以是任何您选择的数据结构）。

提供给 hook_to_notification_point()的回调函数，必须符合下面的形式：

```
int idaapi mycallback (void *user_data, int notif_code, va_list va)
{
    ...
    return 0;
}
```

在处理事件通知时，mycallback 实际上由 IDA 负责调用，user_data 将指向您传递给回调函数的数据结构（在调用 hook_to_notification_point()时定义）。notif_code 将会是接收到的事件标志（接下来两章会列出），va 则是由 IDA 提供的，与事件相关的数据，可能是附加的信息。

如果回调函数允许事件通知被后来的处理者操作，那么它应该返回 0，或者该回调函数本身即最终的处理者，则应该返回任何其他值。

有一点值得注意的是，如果您在插件中使用 hook_to_notification_point()一旦您不需要接收通知了，或已经运行到 IDAP_term()函数了，那么您还应该使用 unhook_from_notification_point()函数。这将在退出 IDA 时避免发生不可预料的段访问错误。回到上面的例子代码，要卸载钩挂的事件通知，应该像下面

一样：

```
unhook_from_notification_point(HT_DBG, dbg_callback, NULL);
```

4.5.2 UI 事件通知

ui_notification_t 是定义在 kernwin.hpp 中的一个枚举，并包括了所有的，可以由 IDA 或者插件产生的用户界面事件通知。要注册获取这些事件通知，您必须将 hook_to_notification_point()的第一个函数设置为 HT_UI。

下面的两个列表给出了一些可以由插件接收或产生的事件通知。这些只是全部事件通知的一个子集；列出来的这些比较通用一点。

尽管这些通知可以由插件调用 callui()来产生，但还有一些助手函数(helper fuction)也有同样的功能，这意味着您无需使用 callui()，而调用助手函数也可以达到相同目的。

事件通知	描述	助手函数
ui_jumpto	移动光标到某地址	jumpto
ui_screenea	返回光标位置的当前地址	get_screen_ea
ui_refresh	刷新所有反汇编界面	refresh_idaview_anyway
ui_mbox	给用户显示一个消息框	vwarning, vinfo 等等
ui_msg	在 IDA 的日志窗口显示一些文本	deb, vmsg
ui_askyn	显示有 Yes 和 No 选项的消息框	askbuttons_cv
ui_askfile	提示用户输入文件名	askfile_cv
ui_askstr	提示用户输入一个单行的字符串	vaskstr
ui_asktext	提示用户输入一些文本	vasktext
ui_form	显示一个表格（很灵活）	AskUsingForm_cv
ui_open_url	在 web 浏览器中打开一个指定的 URL	open_url
ui_load_plugin	加载插件	load_plugin
ui_run_plugin	运行插件	run_plugin
ui_get_hwnd	获取 IDA 窗口的 HWND(窗口句柄)	暂无
ui_get_curline	获取着色的反汇编信息	get_curline
ui_get_cursor	获取当前光标位置的横纵坐标	get_cursor

下面的事件通知由插件接收，并交付给您的回调函数处理。

事件通知	描述
ui_saving 和 ui_saved	分别表示，IDA 正在保存或者已经保存完数据库
ui_term	IDA 已经关闭数据库

比如，下面的代码将产生一个 ui_screenea 事件通知，并用 ui_mbox 事件通知在

IDA 对话框中显示结果。

```
void IDAP_run(int arg)
{
    ea_t addr;
     va_list va;
     char buf[MAXSTR];

    // 获取当前光标的虚拟地址，并保存到 addr 变量
     callui(ui_screenea, &addr);
     qsnprintf(buf, sizeof(buf)-1, "Currently at: %a\n", addr);
    // 在消息框中显示相关信息
    callui(ui_mbox, mbox_inf,  buf,  va);
    return;
}
```

上面这种情况，您一般应该使用助手函数，这里使用 cullui()只不过是演示的目的。

4.5.3 调试器事件通知

调试器事件通知分为三种类型，底层型，高层型和函数返回事件通知；它们之间的不同点，将在接下来的章节解释清楚。上面说的的这些事件通知，都属于 dbg_notification_t 枚举，它定义在 dbg.hpp 头文件中。如果您传递给 hook_to_notification_point()一个 HT_DBG 参数，那么，在 IDA 调试进程的时候，下面这些事件通知将被传给您的插件。

4.5.3.1 底层型事件

下面这些事件摘自 dbg_notification_t，都表示底层型事件。底层事件通知都由调试器生成。

事件通知	描述
dbg_process_start	进程启动
dbg_process_exit	进程终止
dbg_library_load	加载了库文件
dbg_library_unload	卸载了库文件
dbg_exception	产生异常
dbg_breakpoint	非用户定义的断点被激活

您能够使用 debug_event_t 结构(idd.hpp)获取更多关于调试器事件通知的信息，通常是提供给 va 参数到您的回调函数（仅支持底层函数事件）。下面是 debug_evnet_t 结构的全貌。

```
struct debug_event_t
{
```

```
  event_id_t eid;           // 事件代码（常用于解释 ‘info’联合）
  process_id_t pid;         // 事件发生的进程
  thread_id_t tid;          // 事件发生的线程
  ea_t ea;                  // 事件发生的地址
  bool handled;             // 事件是否由调试器处理?
                            // (从系统的角度看)
  // 右边的注释说明了eid值
  // 与联合成员的设置有对应关系。
  union
  {
    module_info_t modinfo;  // PROCESS_START, PROCESS_ATTACH,
                            // LIBRARY_LOAD
    int exit_code;          // PROCESS_EXIT, THREAD_EXIT
    char info[MAXSTR];      // LIBRARY_UNLOAD (卸载的库文件名称)
                            // INFORMATION (若有信息，将被显示在消息窗口)
    e_breakpoint_t bpt;     // BREAKPOINT (非用户定义)
    e_exception_t exc;      // EXCEPTION
  };
};
```

比如，如果您的回调函数接收到dbg_library_load事件通知，您就能检查debug_event_t的modinfo成员，以确定加载了什么文件：

```
...
// 我们的回调函数要处理HT_DBG事件通知
static int idaapi dbg_callback(void *udata, int event_id, va_list va)
{
  // va包含了一个debug_event_t指针
  debug_event_t *evt = va_arg(va, debug_event_t *);
  // 若事件是dbg_library_load，我们就知道modinfo将被填充，
  // 而且，包含了加载的库文件名称
  if (event_id == dbg_library_load)
  msg("Loaded library, %s\n", evt->modinfo.name);

  return 0;
}

// 我们的init函数
int IDAP_init(void)
{
  // 注册通知点为我们的回调函数
  hook_to_notification_point(HT_DBG, dbg_callback, NULL);
...
```

4.5.3.2 高层型事件通知

下面的这些事件摘自dbg_notification_t，它们都是高层型事件通知，由IDA内核产生。

事件通知	描述
dbg_bpt	用户定义的断点被激活
dbg_trace	一条指令被执行（需要允许单步跟踪）
dbg_suspend_process	进程已经被暂停
dbg_request_error	请求的时候，产生一个错误（参看5.14章节）

每个这样的事件通知都有不同的参数，并复制给va参数，供您的回调函数使用。但没有提供debug_event_t，如底层型事件通知那样。

dbg_bpt 事件通知与受影响的线程的线程 ID(thread_id_t)，和被激活断点的地址，一起复制到 va 参数。下面的例子在用户断点被激活的时候，显示一些信息到 IDA 的日志窗口。

```
...
int idaapi dbg_callback(void *udata, int event_id, va_list va)
{
  // Only for the dbg_bpt event notification
  if (event_id == dbg_bpt)
    // 获取线程ID
    thread_id_t tid = va_arg(va, thread_id_t);
    // 获取被激活断点的地址
    ea_t addr = va_arg(va, ea_t);
    msg("Breakpoint hit at: %a, in Thread: %d\n", addr, tid);
    return 0;
}
int IDAP_init(void)
{
hook_to_notification_point(HT_DBG, dbg_callback, NULL);
...
```

4.5.3.3 函数返回型通知

最后一节里，将详细讨论同步和异步调试器函数；到目前为止，所有您需要知道的同步调试器函数只是一些普通函数 - 您调用它们，它们则完成一些事情，然后返回。而异步函数呢，在调用后就立即返回，高效地把请求放进队列，然后再后台运行。当任务完成，一个表示原来请求完成的事件通知就产生了。

下面都是函数返回型通知。

事件通知	描述
dbg_attach_process	调试器附加到进程(IDA 4.8)
dbg_detach_process	调试器离开于进程(IDA 4.9)
dbg_process_attach	调试器附加了一个进程(IDA 4.9)
dbg_process_detach	调试器离开了一个进程(IDA 4.9)

dbg_step_info	**调试器步入一个函数**
dbg_step_over	**调试器步过一个函数**
dbg_run_to	**调试器已经运行到用户的光标位置**
dbg_step_until_ret	**调试器已经返回到调用者的位置**

下面的例子在 IDAP_run()中让 IDA 附加到一个进程。一旦附加成功，IDA 产生一个事件通知，dbg_attach_process，并由 dbg_callback 回调函数所处理。

```
...
int idaapi dbg_callback(void *udata, int event_id, va_list va)
{
  // 获取被附加进程的ID
  process_id_t pid = va_arg(va, process_id_t);
  // 若您使用IDA 4.9,更改dbg_attach_process
  // 为dbg_process_attach
  if (event_id == dbg_attach_process)
  msg("Successfully attached to PID %d\n", pid);
  return 0;
}
void IDAP_run(int arg)
{
  int res;
  // 附加到一个进程，用法参看第五章
  attach_process(NO_PROCESS, res);
  return;
}
int IDAP_init(void) {
hook_to_notification_point(HT_DBG, dbg_callback, NULL);
...
```

4.6 字符串

我们可以用 SDK 来访问 IDA 中的字符串窗口，而每个二进制文件（被打开的文件）的字符串由 string_info_t 结构表示，该结构定义在 strlist.hpp 头文件中。下面是该结构的部分定义。

```
struct string_info_t
{
    ea_t ea;        // 字符串的地址
    int length;     // 字符串长度
    int type;       // 字符串类型 (0=C语言, 1=Pascal, 2=Pascal 2 字节
                    // 3=Unicode, etc.)
...
};
```

请注意，上面的结构并不含字符串。要获取字符串，您需要使用 get_bytes()或者 get_many_bytes()来从二进制文件中提取。要得到所有有效的字符串，可以使用如下代码：

```
// 遍历所有字符串
for (int i = 0; i < get_strlist_qty(); i++) {
  char string[MAXSTR];
  string_info_t si;
  // 获取字符串项目
  get_strlist_item(i, &si);
  if (si.length < sizeof(string)) {
  // 从二进制文件中得到字符串
  get_many_bytes(si.ea, string, si.length);
  if (si.type == 0) // C 字符串
  msg("String %d: %s\n", i, string);
  if (si.type == 3) // Unicode
  msg("String %d: %S\n", i, string);
  }
}
```

上面的一些函数将在第五章《函数》中，详细讨论。

第五章 函数

这一章，将以导出的IDA SDK函数的不同作用，来分类进行描述。从最简单的开始，然后使用更复杂一些的函数。还将提供使用这些函数的简单例子，而且在第六章《实例代码》中，将提供更多的内容。显然，这里并不是一个完整的介绍（更完整的请参阅SDK中的头文件），但总的来看，仍然是一些很有用的函数。

例子的重要事项：下面所有的函数，都可以在IDAP_run()，IDAP_init()或IDAP_term()函数中使用，除非有特殊情况。下面的任何一个例子，都可以粘贴到3.4章节中插件模板的IDAP_run()函数中，而且应该能正常运行。每个函数或例子需要的头文件都会写明。

5.1 常用函数的替代

IDA提供了很多常用C标准库函数的替代调用。推荐您用下面这些替代函数，而不是C标准库函数。在IDA 4.9中，许多C标准库函数都不再有效，您应该使用IDA的替代函数。

C 库函数	IDA 替代函数	定义在
fopen, fread, fwrite, fseek, fclose	qfopen, qfread, qfwrite, qfseek, qfclose	fpro.h
fputc, fgetc, fputs,fgets	qfputc, qfgetc, qfputs, qfgets	fpro.h
vfprintf, vfscanf, vprintf	qfprintf, qfscanf, qvprintf	fpro.h
strcpy, strncpy, strcat, strncat	qstrncpy, qstrncat	pro.h
sprintf, snprintf, wsprintf	qsnprintf	pro.h
open, close, read, write, seek	qopen, qclose, qread, qwrite, qseek	pro.h
mkdir, isdir, filesize	qmkdir, qisdir, qfilesize	pro.h
exit, atexit	qexit, qatexit	pro.h
malloc, calloc, realloc, strdup, free	qalloc, qcalloc, qrealloc, qstrdup, qfree	pro.h

强烈推荐您使用上面这些替代函数，但是如果您在写一个老版本上面的插件，而且因为一些原因要用到C标准库函数，您可以用-DUSE_DANGEROUS_FUNCTIONS 或-DUSE_STANDARD_FILE_FUNCTIONS预定义来进行编译。

5.2 消息框

您可能会在写插件的时候，用一些常见的函数；但并不是因为它们是最有用的，只是因为它们的用法简单，而且在调试插件的时候很有帮助。象你从定义中知道的一样，这些函数都是内联型，而且和printf的参数格式一致。它们定义在kernwin.hpp。

5.2.1 msg

定义	inline int msg(const char *format,...)
含义	在IDA的日志窗口中显示一段文本(静态反汇编模式下，屏幕的底端，或者动态调试模式下，屏幕的顶端)。
示例	msg(“Starting analysis at: %a\n”, inf.startIP);

5.2.2 info

定义	inline int info(const char *format,...)
含义	以“info”形式图标，在一个弹出的对话框中，显示一段文本。
示例	info(“My plug-in v1.202 loaded.”);

5.2.3 warning

定义	inline int warning(const char *format,...)
含义	以”warning”形式图标，在一个弹出对话框中，显示文本。
示例	warning(“Please beware this could crash IDA!\n”);

5.2.3 error

定义	inline int error(const char *format,...)
含义	以 “error”图表形式，在一个弹出对话框中，显示文本。当用户单击OK后，关闭IDA（非正常）。
示例	error(“There was a critical error, exiting IDA.\n”);

5.3 UI浏览

下面这些函数专门用来与IDA GUI进行交互。其中一些用callui()产生事件通知给IDA。所有这些函数定义在kernwin.hpp。

5.3.1 get_screen_ea

定义	inline ea_t get_screen_ea(void)
含义	返回用户的光标所在地址。

示例	#include <kernwein.hpp>
	msg(“Cursor position is %a\n”,get_screen_ea());

5.3.2 jumpto

定义	inline bool jumpto(ea_t ea, int opnum=-1)
含义	移动用户的光标到由ea指定的地址。opnum表示光标要移动到的横坐标位置，opnum为-1则表示不改变横坐标原来的位置。返回true表示成功，false为失败。

示例	#include <kernwin.hpp>
	
	// 跳到入口点+8字节偏移的位置，但不改变
	// 光标以前的横坐标
	jumpto(inf.startIP+8);

5.3.3 get_cursor

定义	inline bool get_cursor(int *x, int *y)
含义	获取光标的横纵坐标到x,y

示例	#include <kernwin.hpp>
	
	int x, y;
	// 保存光标横纵坐标到x, y,
	// 显示结果到日志窗口
	get_cursor(&x, &y);
	msg("X: %d, Y: %d\n", x, y);

5.3.4 get_curline

定义	inline char * get_curline(void)
含义	返回指针，它指向光标位置的一行文本。该函数将返回在那一行中的所有东西：地址，代码和注释。而且也是着色代码，您可以使用tag_remove()清除着色。（参阅5.20.1节）

示例	#include <kernwin.hpp>
	
	// 在日志窗口显示光标所在的一行的文本。
	msg("%s\n", get_curline());

5.3.5 read_selection

定义	inline bool read_selection(ea_t *ea1, ea_t *ea2)
含义	把用户选定的区域的开始和结束地址，分别填充到*ea1和*ea2。

示例	#include <kernwin.hpp>
	
	ea_t saddr, eaddr;
	// 获取选定的地址范围，或者没有选定，
	// 则返回false。
	int selected = read_selection(&saddr, &eaddr);
	if (selected) {
	msg("Selected range: %a -> %a\n", saddr, eaddr);
	} else {
	msg("No selection.\n"
	}

5.3.6 callui

定义	idaman callui_t ida_export_data (idaapi*callui)(ui_notification_t what,...)
含义	用户界面转发函数。允许您调用4.5.2章节列出的事件，以及其它一些ui_notification_t枚举值。callui()的第一个参数，通常是传一个ui_notification_t类型（ui_jumpto,ui_banner,等等），后面跟一些各个事件的参数。

示例	#include <windows.hpp> // 需要HWND的定义
	#include <kernwin.hpp>
	
	// 对于ui_get_hwnd,callui_t的*vptr指针指向结果
	// 我们需要转换该结果，因为vptr是一个void指针
	HWND hwnd = (HWND)callui(ui_get_hwnd).vptr;
	
	// 若hwnd为NULL，表示我们工作在IDA控制台模式之下。

5.3.7 askaddr

定义	inline int askaddr(ea_t *addr,const char *format,...)
含义	显示一个对话框，询问用户提供一个地址。开始以*addr作为默认值，然后当点击OK后，填充addr为用户提供的地址。*format为显示在对话框中的printf输出格式。

示例	#include <kernwin.hpp>
	
	// 设置默认值为文件的入口点
	ea_t addr = inf.startIP;
	// 当用户输入一个地址
	askaddr(&addr, "Please supply an address to jump to.");
	// 移动光标到用户输入的那个地址(参看5.3.2章节)

	jumpto(addr);

5.3.8 AskUsingForm_c

定义	inline int AskUsingForm_c(const char *form,...)
含义	为用户显示一个界面，在这里讨论有些麻烦，但是在kernwin.hpp里有更详细的解释。该接口允许您设计自己的用户界面，包括按钮，文本区域，单选按钮和格式化文本。

示例	#include <kernwin.hpp>
	
	// 第一个 \n 之前的文本为标题，接下来的为首个输入字段，//（用<>表示），以及后来的第二个输入字段。
	// 输入字段的格式为：
	// <label:field type:maximum chars:field length:help
	// identifier>
	// 返回结果分别保存到result1和result2。
	// 要获取输入字段的信息，参看kernwin.hpp的
	// AskUsingForm_c函数解释。
	
	char form[] = “My Title\n<Please enter some text “
	“here:A:20:30::>\n<And
	here:A:20:30::>\n”;
	char result1[MAXSTR] = “”;
	char result2[MAXSTR] = “”;
	AskUsingForm_c(form, result1, result2);
	msg(“User entered text: %s and %s\n”, result1,
	result2);

5.4 入口点

下面的函数可以分析二进制文件的入口点（执行开始之处）。可以在entry.hpp中找到它们。

5.4.1 get_entry_qty

定义	idaman size_t ida_export get_entry_qty(void)
含义	返回当前反汇编文件的入口点数目。通常是返回1，除非是DLL，就会返回更多。

示例	#include <entry.hpp>
	msg(“Number of entry points: %d\n”, get_entry_qty());

5.4.2 get_entry_ordinal

定义	idaman uval_t ida_export get_entry_ordinal(size_t idx)
含义	返回入口点的序数，由idx提供索引参数。您需要这个序数是因为get_entry()和get_entry_name()要使用它。

示例	#include <entry.hpp>
	
	// 显示所有入口点的序数
	for (int e = 0; e < get_entry_qty(); e++)
	msg("Ord # for %d is %d\n", e,
	get_entry_ordinal(e));

5.4.3 get_entry

定义	idaman ea_t ida_export get_entry(uval_t ord);
含义	返回ord参数对应入口点的地址，使用get_entry_ordinal()获取入口点的序数，参看5.4.2章节

示例	#include <entry.hpp>
	
	// 循环搜索每个入口点
	for (int e = 0; e < get_entry_qty(); e++)
	msg("Entry point found at:
	%a\n",get_entry(get_entry_ordinal(e)));

5.4.4 get_entry_name

定义	idaman char * ida_export get_entry_name(uval_t ord)
含义	返回指向入口点地址名称的指针（如,start）

示例	#include <entry.hpp>
	
	// 循环遍历每个入口点
	for (int e = 0; e < get_entry_qty(); e++) {
	int ord = get_entry_ordinal(e);
	// 显示入口点地址和名称
	msg("Entry point %a: %s\n",
	get_entry(ord),
	get_entry_name(ord));
	}

5.5 域

下面这些函数可以分析域和域控制块，分别在4.2.2和4.2.3章节描述过。与以前描述的函数有所不同的是，这些函数是areacb_t类的成员函数，所以只能在这个类的实例中使用这些函数。areacb_t的两个实例是funcs和segs，表示当前反汇编文件中，所有的函数和段。

尽管您应该使用“段处理”(segment-specific)函数来处理段，还有“函数处理”(function-specific)函数来处理函数，使用域会直接给您更多的处理函数和段的手段。

下面这些都定义在area.hpp。

5.5.1 get_area

定义	area_t * get_area(ea_t ea)
含义	返回指向属于ea的area_t结构的指针。

示例	#inlcude <kernwin.hpp>
	#include <funcs.hpp>
	#include <area.hpp>
	
	ea_t addr;
	
	// 询问用户输入一个地址
	askaddr(&addr, “Find the function owner of address:”);
	
	// 获取含有该地址的函数
	// 您应该使用segs.get_area(addr)
	// 来获取包含该地址的段
	areat *area = funcs.get_area(addr);
	msg(“Area holding %a starts at %a, ends at %a\n”,
	addr,
	area->startEA,
	area->endEA);

5.5.2 get_area_qty

定义	uint get_area_qty(void)
含义	获取当前域控制块的域数目。
示例	#include <funcs.hpp> #include <setment.hpp> #include <area.hpp> msg(“%d Functions, and %d Segments”, funcs.get_area_qty(), segs.get_area_qty());

5.5.3 getn_area

定义	area_t * getn_area(unsigned int n)

含义	返回由域序数n指定的，指向area_t结构的指针。

示例	#include <funcs.hpp>
	#include <setments.hpp>
	#include <area.hpp>
	
	// funcs表示所有的函数，因此获取第一个
	// 函数域，使用0作为序数
	
	area_t *firstFunc = funcs.getn_area(0);
	msg(“First func starts: %a, ends: %a\n”,
	firstFunc->startEA,
	firstFunc->endEA);
	
	// segs表示所有的段，因此获取第一个
	// 段域，也使用0作为序数
	area_t *firstSeg = segs.getn_area(0);
	msg(“First seg starts: %a, ends: %a\n”,
	firstSeg->startEA,
	firstSeg->endEA);

5.5.4 get_next_area

定义	int get_next_area(ea_t ea)
含义	返回包含地址ea的域的下一个域的序数。

示例	#include <funcs.hpp>
	#include <area.hpp>
	
	// 循环遍历所有函数的域
	int i = 0;
	for (area_t *func = funcs.getn_area(0);
	i < funcs.get_area_qty();
	i++)
	{
	msg (“Area start: %a, end: %a\n”,
	func->startEA,
	func->endEA);
	int funcNo = funcs.get_next_area(func->startEA);
	funcs = funcs.getn_area(funcNo);
	}

5.5.5 get_prev_area

定义	int get_prev_area(ea_t ea)
含义	返回包含地址ea的域的上一个域的序数

示例	#include <setment.hpp>
	#include <area.hpp>
	
	// 循环遍历所有的段
	int i = segs.get_area_qty();
	for (area_t *seg = segs.getn_area(0); i > 0; i--) {
	  msg (“Area start: %a, end: %a\n”,
	  seg->startEA,
	  seg->endEA);
	  int segNo = segs.get_next_area(seg->startEA);
	  seg = segs.getn_area(segNo);
	}

5.6 段

下面的函数用于分析段（.text，.idata，等），定义在segment.hpp。大部分都是简单封装segs变量的areacb_t成员函数。

5.6.1 get_segm_qty

定义	inline int get_segm_qty(void)
含义	返回当前反汇编文件中，段的总数。可以见调用segs.get_area_qty()
示例	#include <segment.hpp> msg(“%d segments in disassemble file(s).\n” get_segm_qty());

5.6.2 getnseg

定义	inline segment_t * getnseg(int n)
含义	返回由n参数指定的segment_t结构指针。

示例	#include <segment.hpp>
	
	// 获取第一个段的地址
	segment_t *firstSeg = getnseg(0);
	msg(“Addres of the first segment is %a\n”,
	  firstSeg->startEA);

5.6.3 get_segm_by_name

定义	idaman segment_t *ida_export get_segm_by_name(const char *name)

含义	返回一个指向segment_t结构的指针，该结构对应于*name参数所指定的段。如果不存在这个段，则返回NULL。如果多个段有相同的名称，将返回第一个。
示例	#include <segment.hpp> // 获取对应于.text段的segment_t结构 segment_t *textSeg = get_segm_by_name(“.text”); msg(“Text segment is at %a\n”,textSeg->startEA);

5.6.4 getseg

定义	inline segment_t * getseg(ea_t ea)
含义	返回指向对应于段的segment_t结构的指针，该段包含地址ea。这个韩素是segs.get_area()的一个封装形式。

示例	#include <kernwin.hpp>
	#include <segment.hpp>
	
	// 获取用户光标位置的地址
	// 参见5.2.1章节的get_screen_ea()函数
	ea_t addr = get_screen_ea();
	
	// 获取包含该地址的段
	area_t *area = segs.get_area(addr);
	msg(“Segment holding %a starts at %a, ends at %a\n”,
	addr,
	area->startEA,
	area->endEA);

5.6.5 get_segm_name(IDA 4.8)

定义	idaman char *ida_export get_segm_name(const segment_t *s)
含义	返回区段*s的名称(“_text,”,“_idata”,等)

示例	#include <segment.hpp>
	
	// 循环遍历所有的区段，并显示它们的名称
	for (int i = 0; i < get_segm_qty(); i++) {
	segment_t *seg = getnseg(i);
	msg(“Segment %d at %a is named %s\n”,
	i,
	seg->startEA,
	get_segm_name(seg));
	}

5.6.6 get_segm_name(IDA 4.9)

定义	idaman ssize_t ida_export get_segm_name(const segment_t *s, char *buf, size_t bufsize)
含义	用区段*s的名称(“_text”,“_idata”,etc)填充*buf，名称长度限定为bufsize。返回区段名称长度，s为NULL则返回-1。

示例	#include <segment.hpp>
	
	// 循环遍历所有区段，并显示它们的名称
	for (int i = 0; i < get_segm_qty(); i++) {
	  char segName[MAXSTR];
	  segment_t *seg = getnseg(i);
	  get_segm_name(seg, segName, sizeof(segName)-1);
	  msg(“Segment %d at %a is named %s\n”,
	  i,
	  seg->startEA,
	  segName);
	}

5.7 函数

下面介绍处理IDA反汇编文件中的函数的方法。和区段一样，函数也是域。下面这些函数简单地封装了areacb_t的成员方法，它们都定义在funcs.hpp。

5.7.1 get_func_qty

定义	idaman size_t ida_export get_func_qty(void)
含义	返回当前反汇编文件中的函数个数。
示例	#include <funcs.hpp> msg(“%d functions in disassembled file(s).\n”, get_func_qty());

5.7.2 get_func

定义	idaman func_t *ida_export get_func(ea_t ea)
含义	返回指向func_t结构的指针，func_t结构表示包含地址ea的函数.如果ea并没有在该函数内，返回NULL。只返回函数的入口块（参看4.2.3.2章节关于函数块和函数尾的信息）。
示例	#include <kernwin.hpp> #include <funcs.hpp>

	// 获取用户光标所在位置的地址
	ea_t addr = get_screen_ea();
	func_t *func = get_func(addr);
	if (func != NULL) {
	msg("Current function starts at %a\n",
	func->startEA);
	} else {
	msg("Not inside a function!\n");
	}

5.7.3 getn_func

定义	idaman func_t *ida_export getn_func(size_t n)
含义	返回一个指针，它指向以func_t结构表示的函数，参数n表示函数序号。如果n是不存在的函数序号，将返回NULL。同样，它也只返回函数的入口块。

示例	#include <funcs.hpp>
	
	// 循环遍历所有函数
	for (int i = 0; i < get_func_qty(); i++) {
	func_t *curFunc = getn_func(i);
	msg("Function at: %a\n", curFunc->startEA);
	}

5.7.4 get_func_name

定义	idaman char *char_export get_func_name(ea_t ea, char *buf, size_t bufsize)
含义	获取拥有地址ea的函数的名称，并把名称保存到*buf，长度限定为bufsize。返回*buf指针，函数没有名称就返回NULL。

示例	#include <kernwin.hpp>
	#include <funcs.hpp>
	
	// 获取用户的光标所在位置的地址
	ea_t addr = get_screen_ea();
	func_t *func = get_func(addr);
	if (func != NULL) {
	// 保存函数名的缓冲区
	char funcName[MAXSTR];
	if (get_func_name(func->startEA, funcName, MAXSTR)
	!= NULL) {
	msg("Current function %a, named %s\n",
	func->startEA,
	funcName);

	} }

5.7.5 get_next_func

定义	idaman func_t * ida_export get_next_func(ea_t ea)
含义	返回一个指针，它指向以func_t结构表示的，拥有地址ea的函数的下一个函数。若没有下一个函数，则返回NULL。

示例	#include <kernwin.hpp>
	#include <funcs.hpp>
	
	ea_t addr = get_screen_ea();
	// 获取在包含地址的函数的后一个函数。
	func_t *nextFunc = get_next_func(addr);
	
	if (nextFunc != NULL)
	msg("Next function starts at %a\n",
	nextFunc->startEA);

5.7.6 get_prev_func

定义	idaman func_t * ida_export get_prev_func(ea_t ea)
含义	返回一个指针，它指向以func_t结构表示的，拥有地址ea的函数的上一个函数。若没有上一个函数，则返回NULL。

示例	#include <kernwin.hpp>
	#include <funcs.hpp>
	
	ea_t addr = get_screen_ea();
	// 获取在包含地址的函数的后一个函数。
	
	func_t *prevFunc = get_prev_func(addr);
	if (prevFunc != NULL)
	msg("Previous function starts at %a\n",
	prevFunc->startEA);

5.7.7 get_func_comment

定义	inline char * get_func_comment(func_t *fn, bool repeatable)
含义	返回用户添加到由*fn表示的函数的注释。如果repeatable为true，将包括重复性注释。如果该函数没有注释，则返回NULL。

示例	#include <funcs.hpp>
	
	// 循环遍历所有函数，显示它们的注释，
	// 包括重复性注释

	for (int i = 0; i < get_func_qty(); i++) {
	  func_t *curFunc = get_func(i);
	  msg("%a: %s\n",
	  curFunc->startEA,
	  get_func_comment(curFunc, false));
	}

5.8 指令

下面这些函数用来分析反汇编文件中的指令。它们定义在ua.hpp，但除了定义在lines.hpp中的generate_disasm_line()。

5.8.1 generate_disasm_line

定义	idaman bool ida_export generate_disasm_line(ea_t ea, char *buf, size_t bufsize, int flags=0)
含义	把地址ea的反汇编代码填充到*buf，长度限定为bufsize。这些反汇编代码是着过色的，所以您需要使用tag_remove()来得到可以正常打印的文本(参看5.20.1章节)。

示例	#include <kernwin.hpp>
	#include <lines.hpp>
	
	ea_t ea = get_screen_ea();
	// 将保存反汇编文本的缓冲区
	char buf[MAXSTR];
	
	// 保存反汇编文本
	generate_disasm_line(ea, buf, sizeof(buf)-1);
	
	// 显示着色文本的反汇编代码（在IDA的日志
	// 窗口可能不太好读）
	msg("Current line: %s\n", buf);

5.8.2 ua_ana0

定义	idaman int ida_export ua_ana0(ea_t ea)
含义	反汇编地址ea。返回指令的字节数长度，并且把该指令的信息填充到全局cmd结构。如果地址ea处没有指令，返回0。这是只读函数，也不能修改IDA数据库。
示例	#include <kernwin.hpp> #include <ua.hpp>

	ea_t ea = get_screen_ea();
	
	if (ua_ana0(ea) > 0)
	msg("Instruction size: %d bytes\n", cmd.size);
	else
	msg("Not at an instruction.\n");

5.8.3 ua_code

定义	idaman int ida export ua_code(ea_t ea)
含义	反汇编地址ea。返回指令的字节数长度，并用指令的信息填充全局cmd结构，而且将最后的结果更新IDA的数据库。如果ea地址处没有指令，返回0。

示例	#include <kernwin.hpp>
	#include <ua.hpp>
	
	ea_t saddr, eaddr;
	ea_t addr;
	
	// 获取用户的选择范围
	int selected = read_selection(&saddr, %eaddr);
	if (selected) {
	// 重新分析选择的地址区域
	for (addr = saddr; addr <= eaddr; addr++) {
	ua_code(addr);
	}
	} else {
	msg("No selection.\n");
	}

5.8.4 ua_outop

定义	idaman bool ida_export ua_outop(ea_t ea, char *buf, size_t bufsize, int n)
含义	用ea处指令的操作数序号n位置的文本，来填充*buf，长度限定为bufsize，而且如果指令没有被定义，则更新IDA数据库。如果操作数n不存在，则返回false。 返回到*buf中的文本是着过色的，所以您需要使用tag_remove()来获得可正常打印的文本（参看5.20.1章节）
示例	#include <ua.hpp> // 获取入口点地址

	ea_t addr = inf.startIP;
	
	// 把入口点的指令信息填充到cmd。
	
	ua_ana0(addr);
	// 循环遍历每个操作数（直到碰到一个o_void类型），
	// 并显示操作数文本
	for (int i = 0; cmd.Operands[i].type != o_void; i++) {
	    char op[MAXSTR];
	    ua_outop(addr, op, sizeof(op)-1, i);
	    msg("Operand %d: %s\n", i, op);
	}

5.8.5 ua_mnem

定义	idaman const char *ida_export ua_mnem(ea_t ea, char *buf, size_t bufsize)
含义	把ea处指令的助记符填充到*buf，长度限定为bufsize，如果指令没有被定义好，就更新IDA数据库。返回*buf指针，或者ea处没有指令则返回NULL。

示例	#include <segment.hpp>
	#include <ua.hpp>
	
	// 循环遍历每个可执行区段，并显示
	// 每条指令的助记符
	for (int s = 0; s < get_segm_qty(); s++) {
	    segment_t *seg = getnseg(s);
	    is (seg->type == SEG_CODE) {
	        int bytes = 0;
	
	        // a should always be the address of an
	        // instruction, which is why bytes is dynamic
	        // depending on the result of ua_mnem()
	        for (ea_t a = seg->startEA;
	                a < seg->endEA; a += bytes) {
	            char mnem[MAXSTR];
	            const char *res;
	
	        // 获取地址a的指令助记符，并保存到mnem
	        res = ua_mnem(a, mnem, sizeof(mnem)-1);
	
	        // 如果是一条指令，则显示助记符，
	        // 并设置字节数到cmd.size，
	        // 因此由ua_mnem()处理的下一地址，
	        // 即为下一条指令。

```
            if (res != NULL) {
                msg ("Mnemonic at %a: %s\n:", a, mnem);
                bytes = cmd.size;
            } else {
                msg ("No code\n");
                // 如果该地址处没有指令代码，
                // 就把字节数加一，所有ua_mnem()
                // 不会继续处理下一个地址。
                bytes = 1;
            }
        }
    }
}
```

5.9 交叉引用

下面四个函数是xrefblk_t结构的一部分，定义在xref.hpp。它们用来填充和枚举一个地址的交叉引用。所有这些函数以标志作为一个参数，标志可以是如下的一个，同样取自在xref.hpp：

```
#define XREF_ALL        0x00        // 返回所有引用
#define XREF_FAR        0x01        // 不返回普通流程引用
#define XREF_DATA       0x02        // 只返回数据引用
```

普通流程(ordinary flow)指从一条指令直接执行下一条指令，并不通过CALL或JMP（或等效跳转指令）。如果您只对代码交叉引用（忽略普通流程）感兴趣，那么您应该使用XREF_ALL，并在任何情况下都检查xrefblk_t的isCode成员是否为true。如果您只对数据引用感兴趣，就使用XREF_DATA。

5.9.1 first_from

定义	bool first_from(ea_t from, int flags)
含义	用来自from地址的首个交叉引用，来填充xrefblk_t结构。flags参数表示您感兴趣的交叉引用。

示例	#include <kernwin.hpp>
	#include <xref.hpp>
	
	ea_t addr = get_screen_ea();
	xrefblk_t xb;
	if (xb.first_from(addr, XREF_ALL)) {
	    // xb开始被填充
	    msg("First reference FROM %a is %a\n", xb.from,

	xb.to); }

5.9.2 first_to

定义	bool first_to(ea_t to,int flags)
含义	用引用到to地址的首个交叉引用，来填充xrefblk_t结构。flags表示您感兴趣的交叉引用。如果没有to没有引用，则返回NULL。

示例	#include <kernwin.hpp> // For get_screen_ea() definition
	#include <xref.hpp>
	ea_t addr = get_screen_ea();
	xrefblk_t xb;
	if (xb.first_to(addr, XREF_ALL)) {
	  // xb is now populated
	  msg("First reference TO %a is %a\n", xb.to,
	  xb.from);
	}

5.9.3 next_from

定义	bool next_from(void)
含义	用from地址的下一个参考引用，填充xrefblk_t结构。如果没有其它的参考引用，则返回false

示例	#include <kernwin.hpp> // For get_screen_ea() definition
	#include <lines.hpp> // For tag_remove() and
	// generate_disasm_line()
	#include <xref.hpp>
	xrefblk_t xb;
	ea_t addr = get_screen_ea();
	// Replicate IDA 'x' keyword functionality
	for (bool res = xb.first_to(addr, XREF_FAR); res;
	res = xb.next_to()) {
	  char buf[MAXSTR];
	  char clean_buf[MAXSTR];
	
	  // Get the disassembly text for the referencing addr
	  generate_disasm_line(xb.from, buf, sizeof(buf)-1);
	
	  // Clean out any format or colour codes
	  tag_remove(buf, clean_buf, sizeof(clean_buf)-1);
	  msg("%a: %s\n", xb.from, clean_buf);

	}

5.9.4 next_to

定义	bool next_to(void)
含义	用引用到to地址的交叉引用信息，填充xrefblk_t结构。如果没有其它的交叉引用，则返回false

示例	#include <kernwin.hpp> // For get_screen_ea() definition
	#include <xref.hpp>
	xrefblk_t xb;
	ea_t addr = get_screen_ea();
	// Get the first cross reference to addr
	if (xb.first_to(addr, XREF_FAR)) {
	  if (xb.next_to())
	  msg("There are multiple references to %a\n",
	  addr);
	  else
	  msg("The only reference to %a is at %a\n",
	  addr, xb.from);
	}

5.10 名称

下面的函数处理有IDA或用户所设置的函数(sub_*)，位置(loc_*)和变量(arg_*,var_*)的名称。定义在name.hpp。而寄存器名称不能被这些函数识别。

5.10.1 get_name

定义	idaman char *ida_export get_name(ea_t from, ea_t ea, char *buf, size_t bufsize)
含义	用ea的非着色名称填充*buf，长度限定为bufsize。如果ea有一个名称，就返回*buf指针，否则返回NULL。如果您在一个名称后面是一个函数的位置，from也应该位于同样的函数内，或者它不可见。如果您不是在一个位置名称后面，from应该为BADADDR。

示例	#include <name.hpp>
	char name[MAXSTR];
	// Get the name of the entry point, should be start
	// in most cases.
	char *res = get_name(BADADDR,
	inf.startIP, // Entry point
	name,
	sizeof(name)-1);
	if (res != NULL)

	msg("Name: %s\n", name);
	else
	msg("No name for %a\n", inf.startIP);

5.10.2 get_name_ea

定义	idaman ea_t ida_export get_name_ea(ea_t from, const char *name)
含义	返回由定义的*name指定的名称的地址。如果您在一个函数的位置名称之后，from应该也在同一个函数内，或者是不可见的。如果您不是在一个位置名称之后，from应该为BADADDR。

示例	#include <kernwin.hpp> // For askstr and get_screen_ea
	#include <name.hpp>
	// Get the cursor address
	ea_t addr = get_screen_ea();
	// Ask the user for a string (see kernwin.hpp), which
	// will be the name we search for.
	char *name = askstr(HIST_IDENT, // History identifier
	"start", // Default value
	"Please enter a name"); // Prompt
	// Display the address that the name represents. You will
	// get FFFFFFFF for stack variables and nonexistent
	// names.
	msg("Address: %a\n", get_name_ea(addr, name));

5.10.3 get_name_value

定义	idaman int ida_export get_name_value(ea_t from, const char *name, uval_t *value)
含义	返回一个值到*value，它表示对应于地址from的名称*name。*value将包含义个栈偏移或线性地址。 如果您在一个函数的位置名称之后，from应该也位于同一函数内，或者是不可见的。如果您不在一个位置名称后，from应该为BADADDR。返回值是如下的其中之一，表示名称的类型。摘自name.hpp： #define NT_NONE 0 // name doesn't exist or has no value #define NT_BYTE 1 // name is byte name (regular name) #define NT_LOCAL 2 // name is local label #define NT_STKVAR 3 // name is stack variable name #define NT_ENUM 4 // name is symbolic constant #define NT_ABS 5 // name is absolute symbol // (SEG_ABSSYM) #define NT_SEG 6 // name is segment or segment register // name

	#define NT_STROFF 7 // name is structure member #define NT_BMASK 8 // name is a bit group mask name
示例	#define NT_NONE 0 // name doesn't exist or has no value #define NT_BYTE 1 // name is byte name (regular name) #define NT_LOCAL 2 // name is local label #define NT_STKVAR 3 // name is stack variable name #define NT_ENUM 4 // name is symbolic constant #define NT_ABS 5 // name is absolute symbol // (SEG_ABSSYM) #define NT_SEG 6 // name is segment or segment register // name #define NT_STROFF 7 // name is structure member #define NT_BMASK 8 // name is a bit group mask name

5.11 搜索

如下函数用来，在反汇编文件中做一些简单的搜索工作，定义在search.hpp。同样也还有一些其它的搜索函数，它们可以做一些特殊的搜索类型（错误搜索，等），同样也定义在search.hpp。搜索函数有一些标志参数，表示搜索的方式，和搜索的内容，等等。如下为搜索标志，摘自search.hpp：

```
#define SEARCH_UP 0x000 // only one of SEARCH_UP or
// SEARCH_DOWN can be specified
#define SEARCH_DOWN 0x001
#define SEARCH_NEXT 0x002 // Search for the next occurrence
#define SEARCH_CASE 0x004 // Make the search case-sensitive
#define SEARCH_REGEX 0x008 // Use the regular expression parser
#define SEARCH_NOBRK 0x010 // don't test ctrl-break
#define SEARCH_NOSHOW 0x020 // don't display the search progress
#define SEARCH_UNICODE 0x040 // treat strings as unicode
#define SEARCH_IDENT 0x080 // search for an identifier
// it means that the characters before
// and after the pattern can not be
// is_visible_char()
#define SEARCH_BRK 0x100 // return BADADDR if break is
// pressed during find_imm()
```

一般来说，您只要用SEARCH_DOWN，就可以从文件的头部到尾部，进行大小写敏感搜索。

5.11.1 find_text(仅支持IDA 4.9)

定义	idaman ea_t ida_export find_text(ea_t startEA, int y, intx, const char *ustr, int sflag);

含义	从StartEA地址和x、y坐标（都可以为0）开始，搜索当前反汇编文件的*ustr文本。sflag可以是上述的任何一个标志。

示例	#include <kernwin.hpp> // For askstr() definition
	#include <search.hpp>
	char *s = askstr(0, "", "String to search for", NULL);
	// Find the first occurrence of the string
	ea_t foundAt = find_text(inf.minEA, 0, 0, s, SEARCH_DOWN);
	while (foundAt != BADADDR) {
	msg("%s was found at %a\n", s, foundAt);
	}

5.11.2 find_binary

定义	idaman ea_t ida_export find_binary(ea_t startea, ea_t endea, const char *ubinstr, int radix, int sflag)
含义	在startea和endea之间搜索*ubinstr字符串。radix是进制（假如您要搜索数字），它可以是8进制，10进制或16进制。sflag可以是前面提到的任意标志。 注意，该函数并不搜索您在IDA中看到的反汇编文本，它只搜索二进制文本。 您进行的搜索类型不同，则*ubinstr的内容也不同。对于字符串，字符串自身应该被包含在双引号(“)内，对于单个字符，应该包括在单引号（‘）内。问号(?)表示单个字符通配符。

示例	#include <kernwin.hpp> // for askstr() and jumpto()
	#include <search.hpp>
	// Ask the user for a search string
	char *name = askstr(HIST_SRCH, "",
	"Please enter a string");
	char searchstring[MAXSTR];
	// Encapsulate the search string in quotes
	qsnprintf(searchstring, sizeof(searchstring)-1,
	"\"%s\"", name);
	ea_t res = find_binary(inf.minEA, // Top of the file
	inf.maxEA, // Bottom of the file
	searchstring,
	0, // radix not applicable
	SEARCH_DOWN);
	if (res != NULL) {
	msg("Match found at %a\n", res);
	// Move the cursor to the address

	jumpto(res);
	} else {
	msg("No match found.\n");
	}

5.12 IDB

下面的函数能操作IDA数据库(IDB)文件，也可以在loader.hpp中找到。尽管兵灭有实际linput_t类的定义，您仍然需要调用open_linput() (diskio.hpp)函数创建这个类的一个实例，有些函数用此实例作为参数。您也可以用make_linput()转换一个FILE指针为linput_t实例；参阅loader.hpp的更多详情。

5.12.1 open_linput

定义	idaman linput_t *ida_export open_linput(const char *file, bool remote)
含义	为文件名路径*file，创建一个linput_t类的实例。如果此文件在远程机器，设置remote参数为true。返回NULL表示打开文件失败。

示例	#include <kernwin.hpp> // For askfile_cv definition
	#include <diskio.hpp>
	// Prompt the user for a file
	char *file = askfile_cv(0, "", "File to open", NULL);
	// Open the file
	linput_t *myfile = open_linput(file, false);
	if (myfile == NULL)
	msg("Failed to open or corrupt file.\n");
	else
	// Return the size of the opened file.
	msg("File size: %d\n", qlsize(myfile));

5.12.2 close_linput

定义	idaman void ida_export close_linput(linput_t *li)
含义	关闭由linput_t示例表示的文件，*li，由open_linput()创建。
示例	#include <loader.hpp> linput_t *myfile = open_linput("C:\\temp\\myfile.exe", false); close_linput(myfile);

5.12.3 load_loader_module

定义	idaman int ida_export load_loader_module(linput_t *li, const char *lname, const

	char *fname, bool is_remote)
含义	加载一个文件到当前的IDB，文件要么是linput_t的实例，*li，要么是*fname的文件路径名，使用加载模块*lname。若*li为NULL，则必须提供*fname，反之亦然。成功返回1，失败返回0。

示例	#include <kernwin.hpp> // For askfile_cv()
	#include <loader.hpp>
	// Prompt the user for a file to open.
	char *file = askfile_cv(0, "", "DLL file..", NULL);
	// Load it into the IDB using the PE loader module
	int res = load_loader_module(NULL, "pe", file, false)
	if (res < 1)
	msg("Failed to load %s as a PE file.\n", file);

5.12.4 load_binary_file

定义	idaman bool ida_export load_binary_file(const char *filename, linput_t *li, ushort _neflags, long fileoff, ea_t basepara, ea_t binoff, ulong nbytes);
含义	加载名为*filename的二进制文件*li，偏移起始于fileoff。_nflags可以是loader.hpp中的任何一个NEF_标志。nbytes指定从文件中加载的字节数，0表示整个文件。 basepara为将加载的二进制文件的基址，而binoff则是这个段的偏移。您可以安全地将basepara设置为，要加载的文件中的地址，以及置binoff为0。 加载失败则返回false。 此函数并不能加载DLL或者执行文件到IDB。但load_loader_module()可以做到。

示例	#include <kernwin.hpp> // For askfile_cv()
	#include <diskio.hpp> // For open_linput()
	#include <loader.hpp>
	// Ask the user for a filename
	char *file = askfile_cv(0, "", "DLL file..", NULL);
	// Create a linput_t instance for that file
	linput_t *li = open_linput(file, false);
	// Load the file at the end of the currently loaded
	// file (inf.maxEA).
	bool status = load_binary_file(file,
	li,
	NEF_SEGS,
	0,
	inf.maxEA,

	0,
	0);
	if (status)
	msg("Successfully loaded %s at %a\n", file,
	inf.maxEA);
	else
	msg("Failed to load file.\n");

5.12.5 gen_file

定义	idaman int ida_export gen_file(ofile_type_t otype, FILE *fp, ea_t ea1, ea_t ea2, int flags)

含义	生成一个输出文件，*fp，基于当前打开的IDB文件。ea1和ea2分别表示开始和结束地址，但是在某些情况下，可以忽略这两个参数。otype必须为下面的标志中的一个，摘自loader.hpp：
	
	OFILE_MAP = 0, // MAP 文件
	OFILE_EXE = 1, // 执行文件
	OFILE_IDC = 2, // IDC脚本文件
	OFILE_LST = 3, // 反汇编列表文件
	OFILE_ASM = 4, // 汇编
	OFILE_DIF = 5; // 不同处
	
	flags参数可以是如下宏的组合，同样摘自loader.hpp：
	
	#define GENFLG_MAPSEG 0x0001 // map: generate map
	// of segments
	#define GENFLG_MAPNAME 0x0002 // map: include dummy names
	#define GENFLG_MAPDMNG 0x0004 // map: demangle names
	#define GENFLG_MAPLOC 0x0008 // map: include local names
	#define GENFLG_IDCTYPE 0x0008 // idc: gen only
	// information about types
	#define GENFLG_ASMTYPE 0x0010 // asm&lst: gen
	// information about
	// types too
	#define GENFLG_GENHTML 0x0020 // asm&lst: generate html
	// (ui_genfile_callback
	// will be used)
	#define GENFLG_ASMINC 0x0040 // asm&lst: gen information
	// only about types
	
	函数返回-1表示有一个错误，如果函数运行成功，则返回生成文件的行数。对于OFILE_EXE文件，返回0表示失败，1表示成功

示例	#include <loader.hpp>

	// Open the output file
	FILE *fp = qfopen("C:\\output.idc", "w");
	// Generate an IDC output file
	gen_file(OFILE_IDC, fp, inf.minEA, inf.maxEA, 0);
	// Close the output file
	qfclose(fp);

5.12.6 save_database

定义	idaman void ida_export save_database(const char *outfile, bool delete_unpacked)
含义	保存数据库到文件，*output。如果delete_unpacked为false，则临时的未压缩文件不会被删除。而且该函数不会返回任何值，所以没有办法检查是否保存陈宫，除非当函数调用完成后，检测那个文件是否存在。

示例	#include <loader.hpp>
	msg("Saving database...");
	char *outfile = "c:\\myidb.idb";
	save_database(outfile, false);
	// There was an error if the filesize is <= 0
	if (qfilesize(outfile) <= 0)
	msg("failed.\n");
	else
	msg("ok\n");

5.13 标志

如下函数可以检测反汇编文件中字节的特定标志是否被设置。它们定义在bytes.hpp。

5.13.1 getFlags

定义	idaman flags_t ida_export getFlags(ea_t ea)
含义	返回地址ea处设置的标志。您可能需要用它来获取地址的标志，然后将结果用于isHead()，isCoder()等函数。
示例	#include <kernwin.hpp> // For get_screen_ea() definition #include <bytes.hpp> msg("Flags for %a are %08x\n", get_screen_ea(), getFlags(get_screen_ea()));

5.13.2 isEnabled

定义	idaman bool ida_export isEnabled(ea_t ea)
含义	当前反汇编文件中，地址ea是否存在。

示例	#include <kernwin.hpp> // For askaddr() definition
	#include <bytes.hpp>
	ea_t addr;
	askaddr(&addr, "Address to look for:");
	if (isEnabled(addr))
	msg("%a found within the currently opened file(s).",
	addr);
	else
	msg("%a was not found.\n");

5.13.3 isHead

定义	inline bool idaapi isHead(flags_t F)
含义	标志F是否为代码或数据的开头?

示例	#include <kernwin.hpp> // For get_screen_ea()
	definition
	#include <bytes.hpp>
	ea_t addr = get_screen_ea();
	// Cycle through 20 bytes from the cursor position
	// printing a message if the byte is a head byte.
	for (int i = 0; i < 20; i++) {
	flags_t flags = getFlags(addr);
	if (isHead(flags))
	msg("%a is a head (flags = %08x).\n",
	addr, flags);
	addr++;
	}

5.13.4 isCode

定义	inline bool idaapi isCode(flags_t F)
含义	标志F是否表示指令的开头？和isHead()相同，但对于代码则是返回true，数据则不会。所以，如果用在不是头字节的代码字节，将返回false。

示例	#include <segment.hpp> // For segment functions
	#include <bytes.hpp>
	for (int i = 0; i < get_segm_qty(); i++) {
	segment_t *seg = getnseg(i);
	if (seg->type == SEG_CODE) {
	// Look for any bytes in the code segment that
	// aren't code.
	for (ea_t a = seg->startEA; a < seg->endEA; a++) {

	flags_t flags = getFlags(a);
	if (isHead(flags) && !isCode(flags))
	msg("Non-code at %a in segment: %s.\n",
	a,
	get_segm_name(seg));
	}
	}
	}

5.13.5 isData

定义	inline bool idaapi isData(flags_t F)
含义	标志F是否表示一些数据的开头？和isHead()相同，但对于数据则是返回true，代码则不会。所以，如果用在不是头字节的数据字节，将返回false。

示例	#include <segment.hpp> // For segment functions
	#include <bytes.hpp>
	for (int i = 0; i < get_segm_qty(); i++) {
	segment_t *seg = getnseg(i);
	if (seg->type == SEG_DATA) {
	// Look for any bytes in the data segment that
	// aren't data (possibly code).
	for (ea_t a = seg->startEA; a < seg->endEA; a++) {
	flags_t flags = getFlags(a);
	if (isHead(flags) && !isData(flags))
	msg("Non-data at %a in segment: %s.\n",
	a,
	get_segm_name(seg));
	}
	}
	}

5.13.6 isUnknown

定义	inline bool idaapi isUnknown(flags_t F)
含义	标志F是否为IDA没有成功分析出的一个字节？

示例	#include <segment.hpp> // For segment functions
	#include <bytes.hpp>
	// Loop through every segment
	for (int i = 0; i < get_segm_qty(); i++) {
	segment_t *seg = getnseg(i);
	// Look for any unexplored bytes in this segment
	for (ea_t a = seg->startEA; a < seg->endEA; a++) {

	flags_t flags = getFlags(a);
	if (isUnknown(flags))
	msg("Unknown bytes at %a in segment: %s.\n",
	a,
	get_segm_name(seg));
	}
	}

5.14 数据

当分析反汇编文件时，绕过反汇编器直接在二进制文件中，访问其中的字节，非常有用。要做到这些，IDA提供如下函数（下面附带的）。所有下面这些函数都定义在bytes.hpp。这些函数对字节进行操作，而且也有一些函数对word，long,qword进行操作（get_word(),patch_word()等等），也定义在bytes.hpp中。在二进制文件自身中，可以使用这些函数读取数据，也可以在调试器下执行进程的时候，读取进程的内存。详情请参考Debugger函数章节。

5.14.1 get_byte

定义	idaman uchar ida_export get_byte(ea_t ea)
含义	在IDA中打开的当前反汇编文件中，返回地址ea处的字节。如果ea并不存在，则返回BADADDR。要操作更大的数据块，可以使用get_word(),get_long()还有get_qword()。操作多字节数据块，使用get_many_bytes()。

示例	#include <kernwin.hpp> // For get_screen_ea()
	definition
	#include <bytes.hpp>
	// Display the byte value for the current cursor
	// position. The values returned should correspond
	// to those in your IDA Hex view.
	msg("%x\n", get_byte(get_screen_ea()));

5.14.2 get_many_bytes

定义	idaman bool ida_export get_many_bytes(ea_t ea, void *buf, ssize_t size)
含义	在ea地址的开始处，获取size个字节的数据，并保存到*buf。
示例	#include <kernwin.hpp> // For get_screen_ea() definition #include <bytes.hpp> char string[MAXSTR]; flags_t flags = getFlags(get_screen_ea());

	// Only get a string if we're at actual data.
	if (isData(flags)) {
	// Get a string from the binary
	get_many_bytes(get_screen_ea(),
	string,
	sizeof(string)-2);
	// NULL terminate the string, if not already
	// terminated in the binary (so strlen doesn't barf)
	string[MAXSTR-1] = '\0';
	msg("String length: %d\n", strlen(string));
	}

5.14.3 patch_byte

定义	idaman void ida_export patch_byte(ea_t ea, ulong x)
含义	用x替换地址ea处的字节。原始字节保存在IDA数据库，也可以使用get_original_byte()（参阅bytes.hpp）获取原始字节。不保存原始字节，使用put_byte(ea_t ea, ulong x)代替。要操作更大的数据块，可以使用put_word()，put_long()还有put_qword()。操作多字节数据块，使用put_many_bytes()。

示例	#include <kernwin.hpp> // For get_screen_ea()
	#include <bytes.hpp>
	// Get the flags for the byte at the cursor position.
	flags_t flags = getFlags(get_screen_ea());
	// Replace the instruction at the cursor position with
	// a NOP instruction (0x90).
	// Unless used carefully, your executable will probably
	// not work correctly after this :-)
	if (isCode(flags))
	patch_byte(get_screen_ea(), 0x90);

5.14.4 patch_many_bytes

定义	idaman void ida_export patch_many_bytes(ea_t ea, const void *buf, size_t size)
含义	用*buf的内容替换地址ea处的size个字节内容。

示例	#include <kernwin.hpp> // For get_screen_ea() et al
	#include <bytes.hpp>
	// Prompt the user for an address, then a string
	ea_t addr = get_screen_ea();
	askaddr(&addr, "Address to put string:");
	char *string = askstr(0, "", "Please enter a string");
	// Write the user supplied string to the address
	// the user specified.
	patch_many_bytes(addr, string, strlen(string));

5.15 I/O

如5.1节提到的，许多I/O的标准C库函数，都有IDA SDK版本，而且也推荐您使用这些IDA SDK版本。它们定义在diskio.hpp。

5.15.1 fopenWT

定义	idaman FILE *ida_export fopenWT(const char *file)
含义	以写模式，打开*file文件，返回FILE指针，如果打开文件失败则返回NULL。要以读模式打开文件，使用fopenRT()，而且对于二进制文件，用W替换R。要同时读写，使用fopenM()。
示例	#include <diskio.hpp> FILE *fp = fopenWT("c:\\temp\\txtfile.txt"); if (fp == NULL) warning("Failed to open output file.");

5.15.2 openR

定义	idaman FILE *ida_export openR(const char *file)
含义	以只读模式，打开二进制文件*file，并返回FILE指针，如果打开文件失败，则直接退出（显示一个错误框，然后关闭IDA）。以只读模式打开text文件，失败就推出，使用openRT()，要同时读写，使用openM()。
示例	#include <diskio.hpp> FILE *fp = openR("c:\\temp\\binfile.exe");

5.15.3 ecreate

定义	idaman FILE *ida_export ecreate(const char *file)
含义	建立二进制文件*file，以只写模式，返回文件的FILE指针。如果无法建立文件，则显示一个错误框，并直接退出。要建立text文件，使用ecreateT()。
示例	#include <diskio.hpp> FILE *fp = ecreate("c:\\temp\\newbinfile.exe");

5.15.4 eclose

定义	idaman void ida_export eclose(FILE *fp)
含义	关闭FILE指针*fp表示的文件。如果无法关闭文件，则显示一个

	错误框，并直接退出。

示例	#include <diskio.hpp>
	// Open the file first.
	FILE *fp = openR("c:\\temp\\binfile.exe");
	// Close it
	eclose(fp);

5.15.5 eread

定义	idaman void ida_export eread(FILE *fp, void *buf, sszie_t size)
含义	读取FILE指针*fp所表示文件的size个字节数据，保存到缓冲区*buf。如果读取不成功，将显示一个错误框，然后退出IDA。

示例	#include <diskio.hpp>
	char buf[MAXSTR];
	// Open the text file
	FILE *fp = openRT("c:\\temp\\txtfile.txt");
	// Read MAXSTR bytes from the start of the file.
	eread(fp, buf, MAXSTR-1);
	eclose(fp);

5.15.6 ewrite

定义	idaman void ida_export ewrite(FILE *fp, const void *buf, ssize_t size)
含义	写入*buf的size个字节到FILE指针*fp所表示文件中。如果写入不成功，将显示一个错误框，然后退出IDA。

示例	#include <kernwin.hpp> // For read_selection()
	#include <bytes.hpp> // For get_many_bytes()
	#include <diskio.hpp>
	char buf[MAXSTR];
	ea_t saddr, eaddr;
	// Create the binary dump file
	FILE *fp = ecreate("c:\\bindump");
	// Get the address range selected, or return false if
	// there was no selection
	if (read_selection(&saddr, &eaddr)) {
	int size = eaddr - saddr;
	// Dump the selected address range to a binary file
	get_many_bytes(saddr, buf, size);
	ewrite(fp, buf, size);
	}
	eclose(fp);

5.16 调试函数

和以前介绍的一些函数有所不同，下面三章节介绍，如何在执行时操作二进制文件。本章节将着重介绍在二进制文件/进程中的高层操作(如进程和线程的控制)。调试和跟踪将在后续两章节介绍。所有下面的这些函数定义在dbg.hpp，但另外两个invalidate_dbg_contents()和invalidate_dbg_config()函数定义在bytes.hpp。要有效使用下面的示例代码，您应该在二进制文件被IDA动态调试的时候，调用您的插件。

您可能注意到了所有这些函数都没有ida_export前缀。事实上，它们并不需要，因为它们都是封装了callui()的内联函数。

5.16.0 请求（Request）中的注意事项

和SDK中的一些函数有所不同，许多调试函数（也还有一些跟踪函数）有两种模式；普通异步模式，比如示例run_to()，和同步模式或请求(request)模式，比如request_run_to()。函数的所有模式都有相同的形式参数，但在实现的方式上，这两者却各不相同。

同步模式的函数（request_前缀）会把函数推入到一个队列，当您调用run_requests()的时候，事实上是由IDA来执行该函数。而异步模式则直接执行，和普通函数没两样。

当您需要让某些函数由IDA等待执行，同步模式的函数就很方便。5.17.5就是这样一个好例子，当使用del_bpt()删除一系列的断点时会失败，除非使用同步模式,这样在您使用getn_bpt()获取下一个断点时，它ID号会被重新组织。需要注意的是,在一个处理调试器事件通知的函数中,您必须使用同步模式的函数。

在5.16,5.17和5.18章节中的所有函数，也可以作为请求模式的会在函数名后面加*号。

5.16.1 run_requests

定义	bool idaapi run_request(void)
含义	执行队列中的任何请求（即同步函数）。

示例	`#include <dbg.hpp>`
	`// Run to the entry point of the binary`
	`request_run_to(inf.startIP);`
	`// Enable function tracing`
	`request_enable_func_trace();`
	`// Run the above requests`
	`run_requests();`

5.16.2 get_process_state

定义	int idaapi get_process_state(void)
示例	返回当前正在被调试进程的状态。如果该进程是暂停的，就返

	回-1，如果该进程正在运行，则返回1，如果没有进程运行在调试器中，就返回0。

示例	#include <dbg.hpp>
	switch (get_process_state()) {
	case 0:
	msg("No process running.\n");
	break;
	case -1:
	msg("Process is suspended.\n");
	break;
	case 1:
	msg("Process is running.\n");
	break;
	default:
	msg("Unknown status.\n");
	}

5.16.3 get_process_qty

定义	int idaapi get_process_qty(void)
含义	返回在IDA中正在运行进程数量。
示例	#include <dbg.hpp> msg("There are %d processes running.\n", get_process_qty());

5.16.4 get_process_info

定义	process_id_t idaapi get_process_info(int n, process_info_t *process_info);
含义	获取进程序数为n（n并非PID）的*process_info相关信息，并返回该进程的ID。如果*process_info为NULL，则只会返回PID。

示例	#include <dbg.hpp>
	// Only get the info if a process is actually running..
	if (get_process_qty() > 0) {
	process_info_t pif;
	// Populate pif
	get_process_info(0, &pif);
	msg("ID: %d, Name: %s\n", pif.pid, pif.name);
	} else {
	msg("No process running!\n");
	}

5.16.5 start_process *

定义	int idaapi start_process(const char *path = NULL, const char *args = NULL, const char *sdir = NULL);
含义	在*sdir目录下，使用*args参数，开始调试*path表示的程序。如果任何一个参数为NULL，也可以将参数重新添加到Debugger->Process Options...菜单中。

示例	#include <kernwin.hpp> // For askstr()
	#include <dbg.hpp>
	// Ask the user for arguments to supply.
	char *args = askstr(HIST_IDENT, "", "Arguments");
	// Run the process with those arguments
	start_process(NULL, args, NULL);

5.16.6 continue_process*

定义	bool idaapi continue_process(void)
含义	继续执行进程。如果继续执行进程失败，则返回false。同样的，也可以在进程处于暂停状态时，在IDA中按下F9。

示例	#include <dbg.hpp>
	// Continue running the process when the user
	// involkes this plug-in.
	if (continue_process())
	msg("Continuing process..\n");
	else
	msg("Failed to continue process execution.\n");

5.16.7 suspend_process*

定义	bool idaapi suspend_process(void)
含义	暂停当前被调试的进程。如果暂停该进程失败，则返回false。同样也可以在IDA中按‘Pause Process’按钮。

示例	#include <dbg.hpp>
	// Suspend the process being debugged.
	if (suspend_process())
	msg("Suspended process.\n");
	else
	msg("Failed to suspend process.\n");

5.16.8 attach_process*

定义	int idaapi attach_process(process_id_t pid=NO_PROCESS, int event_id=-1)

含义	附加到进程ID为pid参数的进程。附加的进程必须与当前被反汇编的可执行文件映像相一致。如果pid参数为NO_PROCESS，用户将会看到一些要附加的进程列表。返回代码如下(摘自dbg.hpp) // -2 - 无法找到一个兼容的进程 // -1 - 无法附加到指定进程（进程已终止，权限不够，调试器插件不支持，...） // 0 - 用户取消了附加到进程 // 1 - 调试器正确附加到进程

示例	#include <dbg.hpp>
	// Present the user with a list of processes to
	// attach to. If there is no executable running that
	// matches what's open in IDA, no dialog box will
	// be presented.
	int err;
	if ((err = attach_process(NO_PROCESS)) == 1)
	msg("Successfully attached to process.\n");
	else
	msg("Unable to attach, error: %d\n", err);

5.16.9 detach_process*

定义	bool idaapi detach_process(void)
含义	拆分当前被调试的进程。可以是被附加的或者通过IDA运行的进程。如果无法拆离则返回false。拆分进程只在XP SP2和2003下被支持。

示例	#include <dbg.hpp>
	// Detach from the debugged process.
	if (detach_process())
	msg("Successfully detached from process.\n");
	else
	msg("Failed to detach.\n");

5.16.10 exit_process*

定义	bool idaapi exit_process(void)
示例	终止当前被调试的进程。如果无法终止进程则返回false。

示例	#include <dbg.hpp>
	// Terminate the debugged process.
	if (exit_process())
	msg("Successfully terminated the process.\n");
	else
	msg("Failed to terminate the proces.\n");

5.16.11 get_thread_qty

定义	int idaapi get_thread_qty(void)
示例	返回被调试的进程中，存在的线程数量。

示例	#include <dbg.hpp>
	// Only display if there is a process being debugged.
	if (get_process_qty() > 0)
	msg("Threads running: %d\n", get_thread_qty());

5.16.12 get_reg_val

定义	bool idaapi get_reg_val(const char *regname, regval_t *regval)
示例	获取寄存器为*regname的值，并保存到*regval。如果无法获取该寄存器的值，则返回false。寄存器名称大小写敏感。

示例	#include <dbg.hpp>
	// Process needs to be suspended for this to work.
	regval_t eax;
	regval_t eax_upper;
	char *regname = "eax";
	char *regname_upper = "EAX";
	// Prooving that the register name is case insenstive
	if (get_reg_val(regname, &eax))
	msg("eax = %08a\n", eax.ival);
	if (get_reg_val(regname_upper, &eax_upper))
	msg("EAX = %08a\n", eax_upper.ival);

5.16.13 set_reg_val*

定义	bool idaapi set_reg_val(const char *regname, const regval_t *regval)
含义	在同一线程内，设置寄存器*regname的值为*regval。如果设置失败，返回false。和get_reg_val()一样，*regname也是大小写敏感。与其它异步函数不同的是，在调试事件通知例程中调用很安全。

示例	#include <kernwin.hpp> // For get_screen_ea() definition
	#include <dbg.hpp>
	// Suspend the currently executing process.
	suspend_process();
	// Continue execution from the user's cursor position.
	ea_t addr = get_screen_ea();
	char *regname = "EIP";
	if (set_reg_val(regname, addr)) {
	msg("Continuing execution from %a\n", addr);

	continue_process(); }

5.16.14 invalidate_dbgmem_contents

定义	idaman void ida_export invalidate_dbgmem_contents(ea_t ea, asize_t size)
含义	使size字节的内存无效，从ea开始。如果您要无效整个进程内存，设置ea为BADADDR，以及size为0。 使内存空间无效，实质上是刷新IDA内核分给进程的内存缓存，该缓存即您最后访问的进程内存。您应该在进程被暂停后，调用此函数，或者如果您怀疑内存空间已经发生改变。

示例	#include <dbg.hpp>
	#include <bytes.hpp>
	// Process must be suspended for this to work
	// Get the address stored in the ESP register
	regval_t esp;
	get_reg_val("ESP", &esp);
	// Get the value at the address stored in the ESP reg.
	uchar before = get_byte(esp.ival);
	// Invalidate memory contents
	invalidate_dbgmem_contents(BADADDR, 0);
	// Re-fetch contents of the address stored in ESP
	uchar after = get_byte(esp.ival);
	msg("%08a: Before: %a, After: %a\n",
	esp.ival, before, after);

5.16.15 invalidate_dbgmem_config

定义	idaman void ida_export invalidate_dbgmem_config(void)
含义	与invalidate_dbgmem_contents()类似，您使用此函数来确保IDA正在处理最小内存配置。如果被调试的进程已经分配好或清理好内存，当进程暂停时，您需要运行此函数。该函数也会刷新IDA内存缓存，但是比invalidate_dbgmem_contents()要慢。

示例	#include <dbg.hpp>
	#include <bytes.hpp>
	regval_t esp;
	// Get ESP before invalidate config
	get_reg_val("ESP", &esp);
	uchar before = get_byte(esp.ival);
	// Invalidate memory config
	invalidate_dbgmem_config();
	// After invalidate
	uchar after = get_byte(esp.ival);

	msg("%08a Before: %a, After: %a\n", esp.ival, before, after);

5.16.16 run_to *

定义	bool idaapi run_to(ea_t ea)
含义	运行进程，直到地址ea就暂停。如果没有进程在运行，那么当前被反汇编的文件将被执行。如果无法执行进程，则返回false。

示例	#include <kernwin.hpp> // For get_screen_ea() definition
	#include <dbg.hpp>
	// Replicate F4 functionality
	if (!run_to(get_screen_ea()))
	msg("Failed to run to %a\n", get_screen_ea());

5.16.17 step_into*

定义	bool idaapi step_into(void)
含义	执行被调试进程中当前线程的一条指令。这与IDA的F7热键相同。如果无法单步执行指令，就返回false。

示例	#include <dbg.hpp>
	// Go to the entry point (queued)
	request_run_to(inf.startIP);
	// Run 20 instructions (queued)
	for (int i = 0; i < 20; i ++)
	request_step_into();
	// Run through the queue
	run_requests();

5.16.18 step_over*

定义	bool idaapi step_over(void)
含义	执行被调试进程中，当前线程的一条指令，但不会跟进到函数内部，而是把函数视为一条指令。这与IDA的F8热键一致。如果无法步过执行指令，则返回false。

示例	#include <dbg.hpp>
	// This can only run when the process is suspended
	// Step over 5 instructions. This needs to be done as
	// a request, otherwise only one step will execute.
	for (int i = 0; i < 5; i ++)
	request_step_over();
	run_requests();

5.16.19 step_until_ret*

定义	bool idaapi

	step_until_ret(void)
含义	执行被调试进程中，当前线程的每一条指令，直到该函数返回。这与IDA的CTRL-F7热键相同。

示例	#include <dbg.hpp>
	// Get the address of where the function named
	// 'myfunc' is.
	ea_t addr = get_name_ea(BADADDR, "myfunc");
	if (addr != BADADDR) {
	    // Run until execution hits myfunc (queued)
	    request_run_to(addr);
	    // Step into the function (queued)
	    request_step_into();
	    // Continue executing until myfunc returns (queued)
	    request_step_until_ret();
	    // Run through the queue
	    run_requests();
	}

5.17 断点

调试的一个重要部分就是，可以设置并操作断点，可以是进程内存空间的任意地址，或者硬件、软件断点。下面的函数用于操作断点，定义在dbg.hpp。

5.17.1 get_bpt_qty

定义	int idaapi get_bpt_qty(void)
含义	返回已存在的断点数量（不管它们是否打开或关闭）。
示例	#include <dbg.hpp> msg("There are currently %d breakpoints set.\n", get_bpt_qty());

5.17.2 getn_bpt

定义	bool idaapi getn_bpt(int n, bpt_t *bpt)
含义	把序号为n断点的相关信息填充到*bpt参数。如果没有该断点序号，则返回false。

示例	#include <dbg.hpp>
	// Go through all breakpoints, displaying the address
	// of where they are set.
	for (int i = 0; i < get_bpt_qty(); i++) {
	    bpt_t bpt;
	    if (getn_bpt(i, &bpt))
	        msg("Breakpoint found at %a\n", bpt.ea);
	}

5.17.3 get_bpt

定义	bool idaapi get_bpt(ea_t ea, bpt_t *bpt)
含义	把地址ea处的断点相关信息填充到*bpt参数。如果ea地址并没有断点，则返回false。如果*bpt为NULL，此函数就简单返回true，或者在ea地址设置了断点，就返回false。

示例	#include <kernwin.hpp> // For get_screen_ea() definition
	#include <dbg.hpp>
	if (get_bpt(get_screen_ea(), NULL))
	msg("Breakpoint is set at %a.\n", get_screen_ea());
	else
	msg("No breakpoint set at %a.\n", get_screen_ea());

5.17.4 add_bpt*

定义	bool idaapi add_bpt(ea_t ea, asize_t size = 0, bpttype_t type = BPT_SOFT)
含义	在ea地址添加type类型，size大小的断点。如果无法设置断点，则返回false。查阅4.4.2节的关于不同断点类型的解释。在设置软件断点，size就可以忽略。

示例	#include <kernwin.hpp> // For get_screen_ea() definition
	#include <dbg.hpp>
	// Add a software breakpoint at the cursor position
	if (add_bpt(get_screen_ea(), 0, BPT_SOFT))
	msg("Successfully set software breakpoint at %a\n",
	get_screen_ea());

5.17.5 del_bpt*

定义	bool idaapi del_bpt(ea_t ea)
含义	删除ea地址处的断点。如果没有断点，则返回false。

示例	#include <dbg.hpp>
	// Go through all breakpoints, deleting each one.
	for (int i = 0; i < get_bpt_qty(); i++) {
	bpt_t bpt;
	if (getn_bpt(i, &bpt)) {
	// Because we are performing many delete
	// operations, queue the request, otherwise the
	// getn_bpt call will fail when the id
	// numbers change after the delete operation.
	if (request_del_bpt(bpt.ea))
	msg("Queued deleting breakpoint at %a\n",
	bpt.ea);

	}
	}
	// Run through request queue
	run_requests();
	// Make sure there are no breakpoints left over
	if (get_bpt_qty() > 0)
	msg("Failed to delete all breakpoints.\n");

5.17.6 update_bpt

定义	bool idaapi update_bpt(const bpt_t *bpt)
含义	更新由*bpt表示的断点的可修改属性。如果修改不成功，则返回false。

示例	#include <dbg.hpp>
	// Loop through all breakpoints
	for (int i = 0; i < get_bpt_qty(); i++) {
	bpt_t bpt;
	if (getn_bpt(i, &bpt)) {
	// Change the breakpoint to not pause
	// execution when it's hit
	bpt.flags ^= BPT_BRK;
	// Change the breakpoint to a trace breakpoint
	bpt.flags |= BPT_TRACE;
	// Run a little IDC every time it's hit
	qstrncpy(bpt.condition,
	"Message(\"Trace hit!\")",
	sizeof(bpt.condition));
	// Update the breakpoint
	if (!update_bpt(&bpt))
	msg("Failed to update breakpoint at %a\n",
	bpt.ea);
	}
	}

5.17.7 enable_bpt*

定义	bool idaapi enalbe_bpt(ea_t ea, bool enable = true)
含义	打开，或关闭ea地址处的断点。如果没有断点，或者在打开、关闭断点时有错误，则返回false。如果enable参数置为false，断点就是关闭的。
示例	#include <kernwin.hpp> // For get_screen_ea() definition #include <dbg.hpp> bpt_t bpt; // If a breakpoint exists at the user's cursor, disable

	// it.
	if (get_bpt(get_screen_ea(), &bpt)) {
	    if (enable_bpt(get_screen_ea(), false))
	        msg("Disabled breakpoint.\n");
	}

5.18 跟踪

可用于跟踪的函数，大多会仔细检查，一个指定的跟踪类型是否设置了，例如，打开或关闭一个跟踪类型，还有获取跟踪事件。下面的函数定义在dbg.hpp。

5.18.1 set_trace_size

定义	bool idaapi set_trace_size(int size)
含义	设置跟踪缓冲区为size大小。如果分配失败，则返回false。置size为0，则缓冲区为无限大小（危险）。如果您设置的size比当前跟踪事件的数量更少，size事件被删除。
示例	#include <dbg.hpp> // 1000 trace events allowed if (set_trace_size(1000)) msg("Successfully set the trace buffer to 1000\n");

5.18.2 clear_trace*

定义	void idaapi clear_trace(void)
含义	清楚跟踪缓冲区

示例	#include <dbg.hpp>
	// Start our plug-in with a clean slate
	clear_trace();

5.18.3 is_step_trace_enabled

定义	bool idaapi is_step_trace_enabled(void)
含义	如果当前的单步跟踪是打开的，则返回true.
示例	#include <dbg.hpp> if (is_step_trace_enabled()) msg("Step tracing is enabled.\n");

5.18.4 enable_step_trace*

定义	bool idaapi enable_step_trace(int enable = true)
含义	打开但不跟踪。如果enable参数置为false，单步跟踪被关闭。
示例	#include <dbg.hpp> // Toggle step tracing

	if (is_step_trace_enabled())
	    enable_step_trace(false);
	else
	    enable_step_trace();

5.18.5 is_insn_trace_enabled

定义	bool idaapi is_insn_trace_enabled(void)
含义	如果指令跟踪被打开，返回true.
示例	#include <dbg.hpp> if (is_insn_trace_enabled()) msg("Instruction tracing is enabled.\n");

5.18.6 enable_insn_trace*

定义	bool idaapi enalbe_insn_trace(int enalbe = true)
含义	打开指令跟踪。如果enalbe置为false，指令跟踪就被关闭。

示例	#include <dbg.hpp>
	// Toggle instruction tracing
	if (is_insn_trace_enabled())
	    enable_insn_trace(false);
	else
	    enable_insn_trace();

5.18.7 is_func_trace_enalbed

定义	bool idaapi is_func_trace_enabled(void)
含义	如果函数跟踪是打开的，则返回true。
示例	#include <dbg.hpp> if (is_func_trace_enabled()) msg("Function tracing is enabled.\n");

5.18.8 enable_func_trace*

定义	bool idaapi enable_func_trace(int enable = true)
含义	打开函数跟踪，如果enable置为false，函数跟踪就被关闭。

示例	#include <dbg.hpp>
	// Toggle function tracing
	if (is_func_trace_enabled())
	    enable_func_trace(false);
	else
	    enable_func_trace();

5.18.9 get_tev_qty

定义	int idaapi get_tev_qty(void)
含义	返回在跟踪缓冲区中，跟踪事件的数量。
示例	#include <dbg.hpp> msg("There are %d trace events in the trace buffer.\n", get_tev_qty());

5.18.10 get_tev_info

定义	bool idaapi get_tev_info(int n, tev_info_t *tev_info)
含义	把跟踪缓冲区中，序数为n的相关信息填充到*tev_info。如果没有该跟踪事件序数n，则返回false。

示例	#include <dbg.hpp>
	// Loop through all trace events
	for (int i = 0; i < get_tev_qty(); i++) {
	tev_info_t tev;
	// Get the trace event information
	get_tev_info(i, &tev);
	// Display the address the event took place
	msg("Trace event occurred at %a\n", tev.ea);
	}

5.18.11 get_insn_tev_reg_val

定义	bool idaapi get_insn_tev_reg_val(int n, const char *regname, regval_t *regval)
含义	在执行某指令前，当序数为n的指令跟踪事件发生时，保存寄存器*regname的值到*regval。如果该事件不是指令跟踪事件，则返回false。 要在执行后获取寄存器，参阅get_insn_tev_reg_result()。

示例	#include <dbg.hpp>
	// Loop through all trace events
	for (int i = 0; i < get_tev_qty(); i++) {
	regval_t esp;
	tev_info_t tev;
	
	// Get the trace event information
	get_tev_info(i, &tev);
	// If it's an instruction trace event...
	if (tev.type == tev_insn) {
	// Get ESP, store into &esp

	if (get_insn_tev_reg_val(i, "ESP", &esp))
	// Display the value of ESP
	msg("TEV #%d before exec: %a\n", i, esp.ival);
	else
	msg("No ESP change for TEV #%d\n", i);
	}
	}

5.18.12 get_insn_tev_reg_result

定义	bool idaapi get_insn_tev_reg_result(int n, const char *regname, regval_t *regval)
含义	在执行某指令前，当序数为n的指令跟踪事件发生时，保存寄存器*regname的值到*regval。如果该事件不是指令跟踪事件，则返回false。 要在执行前获取寄存器，参阅get_insn_tev_reg_val()。

示例	#include <dbg.hpp>
	// Loop through all trace events
	for (int i = 0; i < get_tev_qty(); i++) {
	regval_t esp;
	tev_info_t tev;
	
	// Get the trace event information
	get_tev_info(i, &tev);
	// If it's an instruction trace event...
	if (tev.type == tev_insn) {
	// Get ESP, store into &esp
	if (get_insn_tev_reg_result(i, "ESP", &esp))
	// Display the value of ESP
	msg("TEV #%d after exec: %a\n", i, esp.ival);
	else
	msg("No ESP change for TEV #%d\n", i);
	}
	}

5.18.13 get_call_tev_callee

定义	ea_t idaapi get_call_tev_callee(int n)
含义	返回调用了序数为n函数跟踪事件的函数地址。如果没有该函数跟踪事件序数n，则返回BADADDR。函数跟踪事件的类型必须为tev_call。
示例	#include <dbg.hpp>

	// Loop through all trace events
	for (int i = 0; i < get_tev_qty(); i++) {
	    regval_t esp;
	    tev_info_t tev;
	
	    // Get the trace event information
	    get_tev_info(i, &tev);
	    // If it's an function call trace event...
	    if (tev.type == tev_call) {
	        ea_t addr;
	        // Get ESP, store into &esp
	        if ((addr = get_call_tev_callee(i)) != BADADDR)
	            msg("Function at %a was called\n", addr);
	    }
	}

5.18.14 get_ret_tev_return

定义	ea_t idaapi get_ret_tev_return(int n)
含义	返回被序数为n的函数事件通知调用的函数地址。如果没有该函数事件通知序数n，则返回BADADDR。函数事件通知的类型必须为tev_ret。

示例	#include <dbg.hpp>
	// Loop through all trace events
	for (int i = 0; i < get_tev_qty(); i++) {
	    tev_info_t tev;
	
	    // Get the trace event information
	    get_tev_info(i, &tev);
	    // If it's an function return trace event...
	    if (tev.type == tev_ret) {
	        ea_t addr;
	        if ((addr = get_ret_tev_return(i)) != BADADDR)
	            msg("Function returned to %a\n", addr);
	    }
	}

5.18.15 get_bpt_tev_ea

定义	ea_t idaapi get_bpt_tev_ea(int n)
含义	返回序数为n的读/写/执行事件通知的地址。如果该事件不是读/写/执行事件，则返回false。
示例	#include <dbg.hpp> // Loop through all trace events

	for (int i = 0; i < get_tev_qty(); i++) {
	    tev_info_t tev;
	
	    // Get the trace event information
	    get_tev_info(i, &tev);
	    // If it's an breakpoint trace event...
	    if (tev.type == tev_bpt) {
	        ea_t addr;
	        if ((addr = get_bpt_tev_ea(i)) != BADADDR)
	            msg("Breakpoint trace hit at %a\n", addr);
	    }
	}

5.19 字符串

下面的函数用来读取IDA的Strings窗口的字符串列表，该列表由当前的反汇编文件生成。如下函数定义在strlist.hpp。

5.19.1 refresh_strlist

定义	idaman void ida_export refresh_strlist(ea_t ea1, ea_t ea2)
含义	刷新IDA的Strings窗口的字符串列表。刷新的范围在当前反汇编文件中的ea1和ea2地址之间。

示例	#include <strlist.hpp>
	// Refresh the string list.
	refresh_strlist();

5.19.2 get_strlist_qty

定义	idaman size_t ida_export get_strlist_qty(void)
含义	返回在当前反汇编文件中的字符串总数。
示例	#include <strlist.hpp> msg("%d strings were found in the currently open file(s)", get_strlist_qty());

5.19.3 get_strlist_item

定义	idaman bool ida_export get_strlist_item(int n, string_info_t *si)
含义	用*si的内容填充到序数为n的字符串。如果没有该字符串，则返回false。

示例	#include <strlist.hpp>
	int largest = 0;
	// Loop through all strings, finding the largest one.
	for (int i = 0; i < get_strlist_qty(); i++) {

	string_info_t si;
	get_strlist_item(i, &si);
	if (si.length > largest)
	largest = si.length;
	}
	msg("Largest string is %d characters long.\n", largest);

5.20 其它

下面的函数没有特定的类别。它们的头文件会在每一处提及。

5.20.1 tag_remove

定义	idaman int ida_export tag_remove(const char *instr, char *buf, int bufsize)
含义	删除*instr中的任何颜色标记，并将结果保存到*buf，限定长度为bufsize。在bufsize为0的情况下，指定*instr和*buf为同一指针也是允许的。该函数定义在ines.hpp。

示例	#include <ua.hpp> // For ua_ functions
	#include <lines.hpp>
	// Get the entry point address
	ea_t addr = inf.startIP;
	// Fill cmd with information about the instruction
	// at the entry point
	ua_ana0(addr);
	// Loop through each operand (until one of o_void type
	// is reached), displaying the operand text.
	for (int i = 0; cmd.Operands[i].type != o_void; i++) {
	char op[MAXSTR];
	ua_outop(addr, op, sizeof(op)-1, i);
	// Strip the colour tags off
	tag_remove(op, op, 0);
	msg("Operand %d: %s\n", i, op);
	}

5.20.2 open_url

定义	inline void open_url(const char *url)
含义	用系统的默认网页浏览器打开*url。该函数定义在kernwin.hpp。

示例	#include <kernwin.hpp>
	open_url("http://www.binarypool.com/idapluginwriting/");
	);

5.20.3 call_system

定义	idaman int ida_export call_system(const char *command)
含义	从系统命令行中运行*command命令。该函数定义在diskio.hpp。

示例	#include <diskio.hpp>
	// Run notepad
	call_system("notepad.exe");

5.20.4 idadir

定义	idaman const char *ida_export idadir(const dir *subdir)
含义	如果*subdir为NULL，则返回IDA根路径。如果不为NULL，则返回IDA的子目录。下面是一些可能的子目录，摘自diskio.hpp： #define CFG_SUBDIR "cfg" #define IDC_SUBDIR "idc" #define IDS_SUBDIR "ids" #define IDP_SUBDIR "procs" #define LDR_SUBDIR "loaders" #define SIG_SUBDIR "sig" #define TIL_SUBDIR "til" 此函数定义在diskio.hpp

示例	#include <diskio.hpp>
	msg("IDA directory is %s\n", idadir(NULL));

5.20.5 getdspace

定义	idaman ulonglong ida_export getdspace(const char *path)
含义	返回*path所处硬盘分区的可用剩余空间。该函数定义在diskio.hpp。

示例	#include <diskio.hpp>
	// Get the disk space on the disk with IDA installed on
	// it.
	if (getdspace(idadir(NULL)) < 100*1024*1024)
	msg("You need at least 100 MB free to run this.");

5.20.6 str2ea

定义	idaman bool ida_export str2ea(const char *p, ea_t *ea, ea_t screenEA)
含义	把字符串*p转换成保存它的地址*ea，成功则返回true。该函数定义在kernwin.hpp。

示例	#include <kernwin.hpp>
	// Just some random address
	char *addr_s = "010100F0";
	ea_t addr;
	// If 010100F0 is in the binary, print the address
	if (str2ea(addr_s, &addr, 0))
	    msg("Address: %a\n", addr);

5.20.7 ea2str

定义	idaman char *ida_export ea2str(ea_t ea, char *buf, int bufsize)
含义	转换地址ea，为字符串，并保存到*buf，限定长度为bufsize。字符串的保存格式为，segmentname:address，所以，比方说，在.text段中提供地址0100102A，就是.text.0100102A。此函数定义在kernwin.hpp。

示例	#include <kernwin.hpp>
	ea_t addr = get_screen_ea();
	char addr_s[MAXSTR];
	// Convert addr into addr_s
	ea2str(addr, addr_s, sizeof(addr_s)-1);
	msg("Address: %s\n", addr_s);

5.20.8 get_nice_colored_name

定义	idaman ssize_t ida_export get_nice_colored_name(ea_t ea, char *buf, size_t bufsize, int flags=0);
含义	获取ea的格式化名称，保存到*buf，限定长度为bufsize。如果标志设为GNCN_NOCOLOR，那么无颜色的代码将会被包含在名称中。如果ea并没有名称，则会返回以一种“可读性”格式的地址，象start+56或者.txet:01002010。该函数定义在name.hpp。

示例	#include <kernwin.hpp> // For get_screen_ea() definition
	#include <name.hpp>
	char buf[MAXSTR];
	// Get the nicely formatted name/address of the
	// current cursor position. No colour codes will
	// be included.
	get_nice_colored_name(get_screen_ea(),
	buf,
	sizeof(buf)-1,
	GNCN_NOCOLOR);
	msg("Name at cursor position: %s\n", buf);

第六章 示例

下面的代码已经包括了，本手册的结构和函数的一些内容。所有的都是详细注释，并且可以用如下方法编译，比方，不需要任何修改，或包含别的头文件，等等。和前面的示例一样。

所有代码也可以在http://www.binarypool.com/idapluginwriting/获得。

6.1 搜索sprintf,strcpy,和sscanf的调用

在审计二进制代码时，下面的代码示例会搜索“low hanging fruit”（直译为低处的果实，寓意为更容易实现的目标）。它通过搜索经常被误用的函数，如，sprintf,strcpy和sscanf（可以自行添加您的更多选择）来完成该项任务。首先，它会搜索这些函数的全局定义地址，然后用IDA的交叉参考引用功能，寻找二进制代码中，引用那些全局定义的地址。

```
//
// unsafefunc.cpp
//
#include <ida.hpp>
#include <idp.hpp>
#include <loader.hpp>
#include <lines.hpp>
#include <name.hpp>
int IDAP_init(void)
{
  if(inf.filetype != f_ELF && inf.filetype != f_PE)    {
    error("Executable format must be PE or ELF, sorry.");
    return PLUGIN_SKIP;
  }
  return PLUGIN_KEEP;
}
void IDAP_term(void)
{
  return;
}
void IDAP_run(int arg)
{
  // The functions we're interested in.
  char *funcs[] = { "sprintf", "strcpy", "sscanf", 0 };
  // Loop through all segments
  for (int i = 0; i < get_segm_qty(); i++) {
    segment_t *seg = getnseg(i);
```

```
    // We are only interested in the pseudo segment created by
    // IDA, which is of type SEG_XTRN. This segment holds all
    // function 'extern' definitions.
    if (seg->type == SEG_XTRN) {
// Loop through each of the functions we're interested in.
      for (int i = 0; funcs[i] != 0; i++) {
        // Get the address of the function by its name
        ea_t loc = get_name_ea(seg->startEA, funcs[i]);
        // If the function was found, loop through it's
        // referrers.
        if (loc != BADADDR) {
          msg("Finding callers to %s (%a)\n", funcs[i], loc);
          xrefblk_t xb;
          // Loop through all the TO xrefs to our function.
          for (bool ok = xb.first_to(loc, XREF_DATA);
ok;
ok = xb.next_to()) {
            // Get the instruction (as text) at that address.
            char instr[MAXSTR];
            char instr_clean[MAXSTR];
            generate_disasm_line(xb.from, instr, sizeof(instr)-1);
            // Remove the colour coding and format characters
            tag_remove(instr, instr_clean, sizeof(instr_clean)-1);
            msg("Caller to %s: %a [%s]\n",
funcs[i],
xb.from,
instr_clean);
          }
        }
      }
    }
  }
  return;
}
char IDAP_comment[] = "Insecure Function Finder";
char IDAP_help[] = "Searches for all instances"
" of strcpy(), sprintf() and sscanf().\n";

char IDAP_name[] = "Insecure Function Finder";
char IDAP_hotkey[] = "Alt-I";
plugin_t PLUGIN =
{
  IDP_INTERFACE_VERSION,
  0,
```

```
  IDAP_init,
  IDAP_term,
  IDAP_run,
  IDAP_comment,
  IDAP_help,
  IDAP_name,
  IDAP_hotkey
};
```

6.2 输出含有MOVS指令的函数

当寻找使用了一些漏洞函数的代码时，诸如strcpy之类，相对于简单地使用函数来说，您可能需要更进一步做相关处理，还要识别一些使用了movs族（movsb,movsd,等）指令的函数。这份插件会遍历所有函数，并搜索任何一条类movs指令。

```
//
// movsfinder.cpp
//
#include <ida.hpp>
#include <idp.hpp>
#include <loader.hpp>
#include <allins.hpp>
int IDAP_init(void)
{
  // Only support x86 architecture
  if(strncmp(inf.procName, "metapc", 8) != 0)     {
    error("Only x86 binary type supported, sorry.");
    return PLUGIN_SKIP;
  }
  return PLUGIN_KEEP;
}
void IDAP_term(void)
{
  return;
}
void IDAP_run(int arg)
{
  // Instructions we're interested in. NN_movs covers movsd,
  // movsw, etc.
  int movinstrs[] = { NN_movsx, NN_movsd, NN_movs, 0 };
  // Loop through all segments
  for (int s = 0; s < get_segm_qty(); s++) {
    segment_t *seg = getnseg(s);
```

```
    // We are only interested in segments containing code.
    if (seg->type == SEG_CODE) {

      // Loop through each function
      for (int x = 0; x < get_func_qty(); x++) {
        func_t *f = getn_func(x);
        char funcName[MAXSTR];
        // Get the function name
        get_func_name(f->startEA, funcName, sizeof(funcName)-1);

        // Loop through the instructions in each function
        for (ea_t addr = f->startEA; addr < f->endEA; addr++) {
          // Get the flags for this address
          flags_t flags = getFlags(addr);

          // Only look at the address if it's a head byte, i.e.
          // the start of an instruction and is code.
          if (isHead(flags) && isCode(flags)) {
            char mnem[MAXSTR];

            // Fill the cmd structure with the disassembly of
            // the current address and get the mnemonic text.
            ua_mnem(addr, mnem, sizeof(mnem)-1);

            // Check the mnemonic of the address against all
            // mnemonics we're interested in.
            for (int i = 0; movinstrs[i] != 0; i++) {
              if (cmd.itype == movinstrs[i])
                msg("%s: found %s at %a!\n", funcName, mnem, addr);
            }
          }
        }
      }
    }
  }
  return;
}
char IDAP_comment[] = "MOVSx Instruction Finder";
char IDAP_help[] =
        "Searches for all MOVS-like instructions.\n"
        "\n"
        "This will display a list of all functions along with\n"
        "the movs instruction used within.";
char IDAP_name[] = "MOVSx Instruction Finder";
```

```
char IDAP_hotkey[] = "Alt-M";
plugin_t PLUGIN =
{
  IDP_INTERFACE_VERSION,
  0,
  IDAP_init,
  IDAP_term,
  IDAP_run,
  IDAP_comment,
  IDAP_help,
  IDAP_name,
  IDAP_hotkey
};
```

6.3 自动加载动态链接库到IDA数据库

很多软件包会以它们的功能，划分开它们成多个文件爱你（动态链接库），动态加载它们可以用LoadLibrary。在这些情况下，让IDA自动加载这些动态链接库到IDB就十分有用了。这份插件会在二进制文件中，搜索包含.dll的字符串。对于搜索的字符串，它会假设这些是二进制文件要加载的，还会提示用户输入动态链接库的全路径，然后再加载到IDB。

```
//
// loadlib.cpp
//
#include <ida.hpp>
#include <idp.hpp>
#include <loader.hpp>
#include <strlist.hpp>
// Maximum number of library files to load into the IDB
#define MAXLIBS 5
int IDAP_init(void)
{
  if (inf.filetype != f_PE) {
    error("Only PE executable file format supported.\n");
    return PLUGIN_SKIP;
  }
  return PLUGIN_KEEP;
}
void IDAP_term(void)
{
  return;
}
void IDAP_run(int arg)
{
```

```
    char loadLibs[MAXLIBS][MAXSTR];
    int libno = 0, i;
    // Loop through all strings to find any string that contains
    // .dll. This will eventuall be our list of DLLs to load.
    for (i = 0; i < get_strlist_qty(); i++) {
      char string[MAXSTR];
      string_info_t si;
      // Get the string item
      get_strlist_item(i, &si);
      if (si.length < sizeof(string)) {
        // Retrieve the string from the binary
        get_many_bytes(si.ea, string, si.length);
        // We're only interested in C strings.
        if (si.type == 0) {
          // .. and if the string contains .dll
          if (stristr(string, ".dll") && libno < MAXLIBS) {
            // Add the string to the list of DLLs to load later on.
            strncpy(loadLibs[libno++], string, MAXSTR-1);
          }
        }
      }
    }
    // Now go through the list of libraries found and load them.
    msg("Loading the first %d libraries found...\n", MAXLIBS);
    for (i = 0; i < MAXLIBS; i++) {
      msg("Lib: %s\n", loadLibs[i]);
      // Ask the user for the full path to the DLL (the executable will
      // only have the file name).
      char *file = askfile_cv(0, loadLibs[i], "File path...\n", NULL);
      // Load the DLL using the pe loader module.
      if (load_loader_module(NULL, "pe", file, 0)) {
        msg("Successfully loaded %s\n", loadLibs[i]);
      } else {
        msg("Failed to load %s\n", loadLibs[i]);
      }
    }
}
char IDAP_comment[] = "DLL Auto-Loader";
char IDAP_help[] = "Loads the first 5 DLLs"
" mentioned in a binary file\n";
char IDAP_name[] = "DLL Auto-Loader";
char IDAP_hotkey[] = "Alt-D";
plugin_t PLUGIN =
{
```

```
    IDP_INTERFACE_VERSION,
    0,
    IDAP_init,
    IDAP_term,
    IDAP_run,
    IDAP_comment,
    IDAP_help,
    IDAP_name,
    IDAP_hotkey
};
```

6.4 断点设置器，记录器

这份专用插件让您可以将当前设置的断点的信息，保存到一个文件，还可以，从文件中读取一列地址，并在相应位置设置断点。为了使插件简洁易读，输入文件要求是正常格式，否则将会出错。您还要修改您的plugins.cfg文件，这样就可以使用一个插件完成多个功能（设置和保存），示例如下：

```
//
// bulkbpt.cpp
//
#include <ida.hpp>
#include <idp.hpp>
#include <loader.hpp>
#include <diskio.hpp>
#include <dbg.hpp>
// Maximum number of breakpoints that can be set
#define  MAX_BPT     100
// Insert the following two lines into your plugins.cfg file
// Replace pluginname with the filename of your plugin minus
// the extension
//
// Write_Breakpoints    pluginname     Alt-D   0
// Read_Breakpoints     pluginname     Alt-E   1
//
void read_breakpoints() {
  char c, ea[9];
  int x = 0, b = 0;
  ea_t ea_list[MAX_BPT];
// Ask the user for the file containing the breakpoints
char *file = askfile_cv(0, "", "Breakpoint list file...", NULL);
// Open the file in read-only mode
FILE *fp = fopenRT(file);
if (fp == NULL) {
```

```
  warning("Unable to open breakpoint list file, %s\n", file);
  return;
}
// Grab 8-byte chunks from the file
while ((c = qfgetc(fp)) != EOF && b < MAX_BPT) {
  if (isalnum(c)) {
    ea[x++] = c;
    if (x == 8) {
      // NULL terminate the string
      ea[x] = 0;
      x = 0;

      // Convert the 8 character string to an address
      str2ea(ea, &ea_list[b], 0);
      msg("Adding breakpoint at %a\n", ea_list[b]);
      // Add the breakpoint as a software breakpoint
      add_bpt(ea_list[b], 0, BPT_SOFT);
      b++;
      }
    }
  }

  // Close the file handle
  qfclose(fp);
}
void write_breakpoints() {
  char c, ea[9];
  int x = 0, b = 0;
  ea_t ea_list[MAX_BPT];

  // Ask the user for the file to save the breakpoints to
  char *file = askstr(0, "", "Breakpoint list file...", NULL);

  // Open the file in write-only mode
  FILE *fp = ecreateT(file);

  for (int i = 0; i < get_bpt_qty(); i++) {
bpt_t bpt;
    char buf[MAXSTR];
    getn_bpt(i, &bpt);

    qsnprintf(buf, sizeof(buf)-1, "%08a\n", bpt.ea);
    ewrite(fp, buf, strlen(buf));
  }
```

```
  // Close the file handle
  eclose(fp);
}
void IDAP_run(int arg)
{
  // Depending on the argument supplied,
  // read the breakpoint list from a file and
  // apply it, or write the current breakpoints
  // to a file.
  switch (arg) {
    case 0:
      write_breakpoints();
      break;
    case 1:
    default:
      read_breakpoints();
      break;
  }
}
int IDAP_init(void)
{
  return PLUGIN_KEEP;
}

void IDAP_term(void)
{
  return;
}
// These are irrelevant because they will be overridden by
// plugins.cfg.
char IDAP_comment[] = "Bulk Breakpoint Setter and Recorder";
char IDAP_help[] =
        "Sets breakpoints at a list of addresses in a text file"
        " or saves the current breakpoints to file.\n"
        "The read list must have one address per line.\n";
char IDAP_name[] = "Bulk Breakpoint Setter and Recorder";
char IDAP_hotkey[] = "Alt-B";
plugin_t PLUGIN =
{
  IDP_INTERFACE_VERSION,
  0,
  IDAP_init,
  IDAP_term,
```

```
  IDAP_run,
  IDAP_comment,
  IDAP_help,
  IDAP_name,
  IDAP_hotkey
};
```

6.5 可选式跟踪(方法一)

这份插件让您可以在指定地址范围内，打开指令跟踪。原理是，在运行到开始地址，打开指令跟踪，运行到结束地址，最后关掉指令跟踪。方法二则演示了更灵活的方式，利用单步跟踪。

```
//
// snaptrace.cpp
//
#include <ida.hpp>
#include <idp.hpp>
#include <loader.hpp>
#include <dbg.hpp>
int IDAP_init(void)
{
  return PLUGIN_KEEP;
}
void IDAP_term(void)
{
  return;
}
void IDAP_run(int arg)
{
  // Set the default start address to the user cursur position
  ea_t eaddr, saddr = get_screen_ea();
  // Allow the user to specify a start address
  askaddr(&saddr, "Address to start tracing at");
  // Set the end address to the end of the current function
  func_t *func = get_func(saddr);
  eaddr = func->endEA;
  // Allow the user to specify an end address
  askaddr(&eaddr, "Address to end tracing at");
  // Queue the following
  // Run to the start address
  request_run_to(saddr);
  // Then enable tracing
  request_enable_insn_trace();
```

```
  // Run to the end address, tracing all stops in between
  request_run_to(eaddr);
  // Turn off tracing once we've hit the end address
  request_disable_insn_trace();
  // Stop the process once we have what we want
  request_exit_process();
  // Run the above queued requests  run_requests();
}
// These are actually pointless because we'll be overriding them
// in plugins.cfg
char IDAP_comment[] = "Snap Tracer";
char IDAP_help[] = "Allow tracing only between user "
                   "specified addresses\n";
char IDAP_name[] = "Snap Tracer";
char IDAP_hotkey[] = "Alt-T";
plugin_t PLUGIN =
{
  IDP_INTERFACE_VERSION,
  0,
  IDAP_init,
  IDAP_term,
  IDAP_run,
  IDAP_comment,
  IDAP_help,
  IDAP_name,
  IDAP_hotkey
};
```

6.6 可选式跟踪（方法二）

利用单步跟踪，这份插件设置了一个调试事件通知处理函数，来处理跟踪事件（执行了一条指令）。使用这个处理函数，可以检查EIP是否位于用于定义的范围内，如果是的话，就显示ESP。很明显，您可以用这类函数来完成许多有趣的事情，比如警示寄存器和/或内存的内容。

```
//
// snaptrace2.cpp
//
#include <ida.hpp>
#include <idp.hpp>
#include <loader.hpp>
#include <dbg.hpp>
ea_t start_ea = 0;
ea_t end_ea = 0;
// Handler for HT_DBG events
```

```
int idaapi trace_handler(void *udata, int dbg_event_id, va_list va)
{
  regval_t esp, eip;
  // Get ESP register value
  get_reg_val("esp", &esp);
  // Get EIP register value
  get_reg_val("eip", &eip);
  // We'll also receive debug events unrelated to tracing,
  // make sure those are filtered out
  if (dbg_event_id == dbg_trace) {
    // Make sure EIP is between the user-specified range
    if (eip.ival > start_ea && eip.ival < end_ea)
msg("ESP = %a\n", esp.ival);
  }
  return 0;
}
int IDAP_init(void)
{
  // Receive debug event notifications
  hook_to_notification_point(HT_DBG, trace_handler, NULL);
  return PLUGIN_KEEP;
}
void IDAP_term(void)
{
  // Unhook from the notification point on exit
  unhook_from_notification_point(HT_DBG, trace_handler, NULL);
  return;
}
void IDAP_run(int arg) {
  // Ask the user for a start and end address
  askaddr(&start_ea, "Start Address:");
  askaddr(&end_ea, "End Address:");

  // Queue the following

  // Run to the binary entry point
  request_run_to(inf.startIP);
  // Enable step tracing
  request_enable_step_trace();
  // Run queued requests
  run_requests();
}
// These are actually pointless because we'll be overriding them
// in plugins.cfg
```

```
char IDAP_comment[] = "Snap Tracer 2";
char IDAP_help[] = "Allow tracing only between user "
                   "specified addresses\n";
char IDAP_name[] = "Snap Tracer 2";
char IDAP_hotkey[] = "Alt-I";
plugin_t PLUGIN =
{
  IDP_INTERFACE_VERSION,
  0,
  IDAP_init,
  IDAP_term,
  IDAP_run,
  IDAP_comment,
  IDAP_help,
  IDAP_name,
  IDAP_hotkey
};
```

6.7 二进制代码拷贝&粘贴

看到IDA中并没有二进制代码拷贝和粘贴的功能，这份插件将实现拷贝和粘贴的功能，这些操作可以让您从一个地方获取一块二进制代码，而在另一个地方填充它。因为这是一个多功能插件，如果需要一个调用拷贝，另一个粘贴，那么您就需要修改您的plugins.cfg文件。很显然，在IDA中，它仅仅是支持拷贝和粘贴，但是也可以扩展其中功能。

```
//
// copypaste.cpp
//
#include <ida.hpp>
#include <idp.hpp>
#include <loader.hpp>
#define  MAX_COPYPASTE  1024
// This will hold our copied buffer for pasting
char data[MAX_COPYPASTE];
// Bytes copied into the above buffer
ssize_t filled = 0;
// Insert the following two lines into your plugins.cfg file
// Replace pluginname with the filename of your plugin minus
// the extension.
//
//  Copy_Buffer    pluginname    Alt-C  0
//  Paste_Buffer    pluginname    Alt-V  1
//
```

```
int IDAP_init(void)
{
  return PLUGIN_KEEP;
}
void IDAP_term(void)
{
  return;
}
void copy_buffer() {
  ea_t saddr, eaddr;
  ssize_t size;
  // Get the boundaries of the user selection
  if (read_selection(&saddr, &eaddr)) {
    // Work out the size, make sure it doesn't exceed the buffer
    // we have allocated.
    size = eaddr - saddr;
    if (size > MAX_COPYPASTE) {
      warning("You can only copy a max of %d bytes\n", MAX_COPYPASTE);
return;
    }
    // Get the bytes from the file, store it in our buffer
    if (get_many_bytes(saddr, data, size)) {
      filled = size;
      msg("Successfully copied %d bytes from %a into memory.\n",
size,
saddr);
    } else {
      filled = 0;
    }
  } else {
    warning("No bytes selected!\n");
    return;
  }
}
void paste_buffer() {
  // Get the cursor position. This is where we will paste to
  ea_t curpos = get_screen_ea();
  // Make sure the buffer has been filled with a Copy operation first.
  if (filled) {
    // Patch the binary (paste)
    patch_many_bytes(curpos, data, filled);
    msg("Patched %d bytes at %a.\n", filled, curpos);
  } else {
    warning("No data to paste!\n");
```

```
    return;
  }
}
void IDAP_run(int arg) {
  // Based on the argument supplied in plugins.cfg,
  // we can use the one plug-in for both the copy
  // and paste operations.
  switch(arg) {
    case 0:
      copy_buffer();
      break;
    case 1:
      paste_buffer();
      break;
    default:
      warning("Invalid usage!\n");
      return;
  }
}
// These are actually pointless because we'll be overriding them
// in plugins.cfg
char IDAP_comment[] = "Binary Copy and Paster";
char IDAP_help[] = "Allows the user to copy and paste binary\n";
char IDAP_name[] = "Binary Copy and Paster";char IDAP_hotkey[] = "Alt-I";
plugin_t PLUGIN =
{
  IDP_INTERFACE_VERSION,
  0,
  IDAP_init,
  IDAP_term,
  IDAP_run,
  IDAP_comment,
  IDAP_help,
  IDAP_name,
  IDAP_hotkey
};
```