目录

Xcode内置调试器：LLDB

- 最新版本： `v0.7`
- 更新时间： `20221026`

简介

介绍Xcode内置的调试器LLDB。先是LLDB概览；再详细介绍LLDB的命令，包括LLDB的命令概览和LLDB的各个命令；LLDB命令概览包括cheat sheet和help语法；LLDB常用命令包括image、register、expression，尤其是其中的p和po、memory、disassemble、thread、frame、breakpoint、watchpoint、以及调试控制相关的命令，包括run、continue、next和nexti、step和stepi、jump、finish、exit等最后再整理出相关心得，包括命令的缩写、Xcode中的lldb、iOS逆向、LLVM等等。最后给出相关的文档和资料。

源码+浏览+下载

本书的各种源码、在线浏览地址、多种格式文件下载如下：

HonKit源码

- crifan/xcode_debugger_lldb: Xcode内置调试器：LLDB

如何使用此HonKit源码去生成发布为电子书

详见：crifan/honkit_template: demo how to use crifan honkit template and demo

在线浏览

- Xcode内置调试器：LLDB book.crifan.org
- Xcode内置调试器：LLDB crifan.github.io

离线下载阅读

- Xcode内置调试器：LLDB PDF
- Xcode内置调试器：LLDB ePub
- Xcode内置调试器：LLDB Mobi

版权和用途说明

鸣谢

感谢我的老婆陈雪的包容理解和悉心照料，才使得我 `crifan` 有更多精力去专注技术专研和整理归纳出这些电子书和技术教程，特此鸣谢。

更多其他电子书

本人 `crifan` 还写了其他 `100+` 本电子书教程，感兴趣可移步至：

crifan/crifan_ebook_readme: Crifan的电子书的使用说明

LLDB概览

TODO：

- 【已解决】XCode和lldb调试常见用法和调试心得

背景

- 主流常见 `调试器` = `debugger`
  - GNU 的 `GDB`
  - (开源项目 `LLVM` 中的) `LLDB`
- Apple的 `Xcode` 的内置调试器
  - 之前： `GDB`
  - 现在(`Xcode 5+`)： `LLDB`

LLDB

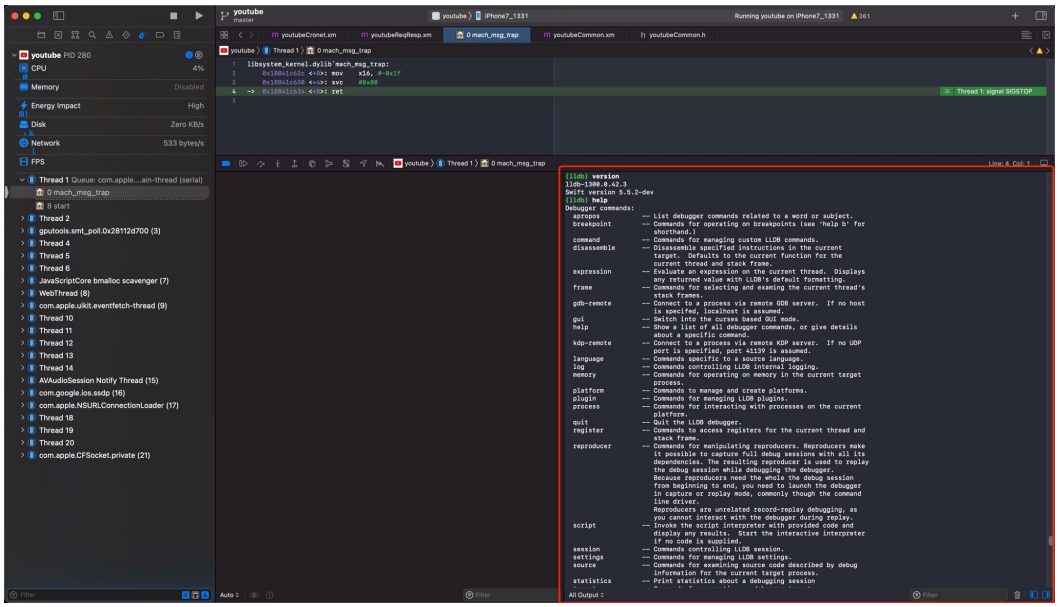
- `LLDB`
  - 名称：常写成小写的 `lldb`
  - 是什么：一个下一代的、高性能的 `开源调试器`
  - 说明
    - 和LLVM关系
      - 属于（更大的，开源的） `LLVM` 项目的 `一部分` =其中 `一个模块`
        - 所以LLDB也是开源的
      - 常搭配 `LLVM` 的其他模块一起使用
        - `expression parser` = 解释器 ： `Clang`
        - `disassembler` = 反汇编器 ： `LLVM disassembler`
    - 和Xcode关系
      - 是Xcode内置的调试器：之前是GDB，现在是LLDB
  - 特点
    - 支持调试语言
      - Xcode中的LLDB
        - 支持调试 `C` 、 `Objective-C` 、 `C++`
        - 支持运行平台：桌面端 `macOS` 、移动端 `iOS` （设备和模拟器）
    - 支持众多平台： `macOS` 、 `iOS` 、 `Linux` 、 `FreeBSD` 、 `NetBSD` 、 `Windows`

Features matrix [4]

Feature	FreeBSD	Linux	macOS	Windows
Backtracing	✓	✓	✓	✓
Breakpoints	✓	✓	✓	✓
C++11:	✓	✓	✓	?
Command-line lldb tool	✓	✓	✓	✓
Core file debugging	✓	✓	✓	✓
Debugserver (remote debugging)	Not ported	Not ported	✓	Not ported
Disassembly	✓	✓	✓	✓
Expression evaluation	?	Works with some bugs	✓	Works with some bugs
JIT debugging	?	Symbolic debugging only	Untested	✘
Objective-C 2.0:	?	N/A	✓	N/A

    - 支持 `REPL` 、 `C++` 和 `Python` 插件
        - 注： `REPL` = `Read-Eval-Print Loop` = 交互式解释器
- 此处
    - 主要使用场景
        - iOS逆向时，用 `LLDB` 调试 `ObjC` 的相关内容

lldb的位置和版本

- Mac中的lldb
    - 二进制
        - Mac自带的： `/usr/bin/lldb`
        - Xcode中的： `/Applications/Xcode.app/Contents/Developer/usr/bin/lldb`
    - 集成进XCode
        - 位置：内嵌在Xcode中的（一般是右下角的）调试区域的控制台
        - 

Mac自带的lldb

```
crifan@licrifandeMacBook-Pro  ~    which lldb
```

```
/usr/bin/lldb
 crifan@licrifandeMacBook-Pro  ~  ll /usr/bin/lldb
-rwxr-xr-x  1 root  wheel   134K  1  1  2020 /usr/bin/lldb

crifan@licrifandeMacBook-Pro  ~  /usr/bin/lldb --version
lldb-1300.0.42.3
Swift version 5.5.2-dev
```

Xcode中的lldb

```
 crifan@licrifandeMacBook-Pro  ~  ll /Applications/Xcode.app/Contents/Developer/usr/bin/lldb
-rwxr-xr-x  1 crifan  staff   828K 12 15  2021 /Applications/Xcode.app/Contents/Developer/usr/b
in/lldb

 crifan@licrifandeMacBook-Pro  ~  /Applications/Xcode.app/Contents/Developer/usr/bin/lldb --ver
sion
lldb-1300.0.42.3
Swift version 5.5.2-dev
```

LLDB命令

LLDB命令概览

TODO：

【整理】lldb的语法和用法

lldb的cheat sheet

lldb的 `cheat sheet` =小抄=手册：

lldb cheat sheet

Execution Commands

start lld (prefix with xcrun on os x)
```
>lldb [program.app]
>lldb -- program.app arg1
```

load program
```
>file program.app
```

run program
```
>process launch [-- args]
>run [args]
```

set arguments
```
>settings set target.run-args 1
```

launch process in new terminal
```
>process launch --tty -- <args>
```

set env variables
```
>settings set target.env-vars DEBUG=1
```

remove env variables
```
>settings remove target.env-vars DEBUG
```

show program arguments
```
>settings show target.run-args
```

set env variable and run
```
>process launch -v DEBUG=1
```

attach to process by PID
```
>process attach --pid 123
```

attach to process by name
```
>process attach --name a.out [--waitfor]
```

attach to remote gdb on eorgadd
```
>gdb-remote eorgadd:8000
```

attach to gdb server on localhost
```
>gdb-remote 8000
```

attach to remote Darwin kernel in kdp mode
```
>kdp-remote eorgadd
```

source level single step
```
>thread step-in
>step
>s
```

source level setop over
```
>thread step-over
>next
>n
```

instruction level single step
```
>thread step-inst
```

```
>si
```

instruction level single step over
```
>thread step-inst-over
>ni
```

step out of the currently selected frame
```
>thread step-out
>finish
```

Return from currently frame, with return value
```
>thread return [RETURN EXPRESSION]
```

Backtrace and disassemble every time you stop
```
>target stop-hook add
>bt
>disassemble --pc
>DONE
```

run until line 12 or end of frame
```
>thread until 12
```

Breakpoint Commands

set breakpoint at all functions named main
```
>breakpoint set --name main
>br s -n main
>b main
```

set breakpoint in file test.c line 12
```
>breakpoint set --file test.c --line 12
>br s -f test.c -l 12
>b test.c:12
```

set breakpoint at all C++ methods with name main
```
>breakpoint set --method main
>br s -M main
```

set breakpoint at ObjC function
```
>breakpoint set --name "-[NSString stringWithFormat:]"
>b -[NSString stringWithFormat:]
```

set breakpoint at all ObjC functions whose selector is count
```
>breakpoint set --selector count
>br s -S count
```

set breakpoint by regular expression function name
```
>breakpoint set --func-regex print.*
```

ensure that breakpoints by file and line work (c/cpp/objc)
```
>settings set target.inline-breakpoint-strategy always
```

```
>br s -f foo.c -l 12
```

set a breakpoint by regular expression on source file contents
```
>breakpoint set --source-pattern regular-expression --file SourceFile
>br s -p regular-expression -f file
```

set conditional breakpoint
```
>breakpoint set --name foo --condition '(int)strcmp(y,"hello") == 0'
>br s -n foo -c '(int)strcmp(y,"hello") == 0'
```

list breakpoints
```
>breakpoint list
>br l
```

delete a breakpoint
```
>breakpoint delete 1
>br del 1
```

Watchpoint Commands

set watchpoint on variable when written to
```
>watchpoint set variable global_var
>wa s v global_var
```

set watchpoint on memory of pointer size
```
>watchpoint set expression --
0x123456
>wa s e -- 0x123456
```

set watchpoint on memory of custom size
```
> watchpoint set expression -x
byte_size -- 0x123456
> wa s e -x byte_size -- 0x123456
```

set a condition on a watchpoint
```
>watch set var global
>watchpoint modify -c '(global==5)'
```

list watchpoints
```
>watchpoint list
>watch l
```

delete a watchpoint
```
>watchpoint delete 1
>watch del 1
```

Examining Variables

show arguments and local variables
```
>frame variable
>fr v
```

show local variables
```
>frame variable --no-args
>fr v -a
```

show contents of variable bar
```
>frame variable bar
>fr v bar
>p bar
```

show contents of var bar formatted as hex
```
>frame variable --format x bar
>fr v -f x bar
```

show contents of global variable baz
```
>target variable baz
>ta v baz
```

show global/static variables in current file
```
>target variable
>ta v
```

show argc and argv every time you stop
```
>target stop-hook add --one-liner "frame variable argc argv"
>ta st a -o "fr v argc argv"
>display argc
>display argv
```

display argc and argv when stopping in main
```
>target stop-hook add --name main --one-liner "frame variable argc argv"
>ta st a -n main -o "fr v argc argv"
```

display *this when in class MyClass
```
>target stop-hook add --classname MyClass --one-liner "frame variable *this"
>ta st a -c MyClass -o "fr v *this"
```

Evaluating Expressions

evaluate expression (print alias possible as well)
```
>expr (int) printf ("Print nine: %d.", 4 + 5)
>print (int) printf ("Print nine: %d.", 4 + 5)
```

using a convenience variable
```
>expr unsigned int $foo = 5
```

print the ObjC description of an object
```
>expr -o -- [SomeClass returnAnObject]
>po [SomeClass returnAnObject]
```

print dynamic type of expression result
```
>expr -d 1 -- [SomeClass returnAnObject]
>expr -d 1 –someCPPObjectPtrOrReference
```

set dynamic type printing as default
```
>settings set target.prefer-dynamic run-target
```

calling a function with a breakpoint
```
>expr -i 0 --function_with_a_breakpoint()
```

calling a function that crashes
```
expr -u 0 -- function_which_crashes()
```

Examining Thread State

show backtrace (current thread)
```
>thread backtrace
>bt
```

show backtrace for all threads
```
>thread backtrace all
>bt all
```

backtrace the first 5 frames of current thread
```
>thread backtrace -c 5
>bt 5 (lldb-169 and later)
>bt –c 5 (lldb-168 and later)
```

select a different stack frame by index
```
>frame select 12
>fr s 12
>f 12
```

show frame information
```
>frame info
```

select stack frame the called current frame
```
>up
>frame select --relative=1
```

select stack frame that is called by current frame
```
>down
>frame select --relative=-1
>fr s -r-1
```

select different frame using relative offset
```
>frame select --relative 2
>fr s -r2
>frame select --relative -3
>fr s -r-3
```

show general purpose registers
```
>register read
```

write 123 to register rax
```
>register write rax 123
```

skip 8 bytes using with program counter
```
>register write pc `$pc+8`
```

show general purpose registers as signed decimal
```
>register read --format i
>re r -f i
>register read/d
```

show all registers in all register threads
```
>register read --all
>re r -a
```

show registers rax, rsp, rbp
```
register read rax rsp rbp
```

show register rax with binary format
```
>register read --format binary rax
```

read memory from 0xbffff3c0 and show 4 hex uint32_t values
```
>memory read --size 4 --format x --count 4 0xbffff3c0
>me r -s4 -fx -c4 0xbffff3c0
>x -s4 -fx -c4 0xbffff3c0
>memory read/4xw 0xbffff3c0
>x/4xw 0xbffff3c0
>memory read --gdb-format 4xw 0xbffff3c0
```

read memory starting at the expression "argv[0]"
```
>memory read `argv[0]`
>memory read --size `sizeof(int)` `argv[0]`
```

read 512 bytes from address 0xbffff3c0 and save results to a local file
```
>memory read --outfile /tmp/mem.txt --count 512 0xbffff3c0
>me r -o/tmp/mem.txt -c512 0xbffff3c0
>x/512bx -o/tmp/mem.txt 0xbffff3c0
```

save binary memory data starting at 0x1000 and ending at 0x2000 to file
```
>memory read --outfile /tmp/mem.bin --binary 0x1000 0x2000
>me r -o /tmp/mem.bin -b 0x1000 0x2000
```

get information about specific heap allocation (Mac OS X only)
```
>command script import lldb.macosx.heap
>process launch --environment MallocStackLogging=1 -- [ARGS]
>malloc_info --stack-history 0x10010d680
```

get information about specific heap allocation and cast result to dynamic type that can be deduced (Mac OS X only)
```
>command script import lldb.macosx.heap
>malloc_info --type 0x10010d680
```

find all heap blocks that contain pointer specified by an expression EXPR (Mac OS X only)
```
>command script import lldb.macosx.heap
>ptr_refs EXPR
```

find all heap blocks that contain a C string anywhere in the block (Mac OS X only)
```
>command script import lldb.macosx.heap
>cstr_refs CSTRING
```

disassemble current function for current frame
```
>disassemble –frame
>di -f
```

disassemble any functions named main
```
>disassemble --name main
>di -n main
```

disassemble address range
```
>disassemble --start-address 0x1eb8 --end-address 0x1ec3
>di -s 0x1eb8 -e 0x1ec3
```

disassemble 20 instructions from start address
```
>disassemble --start-address 0x1eb8 --count 20
>di -s 0x1eb8 -c 20
```

show mixed source and disassembly for the current function
```
>disassemble --frame –mixed
>di -f -m
```

disassemble the current function for the current frame and show the opcode bytes
```
>disassemble --frame –bytes
>di -f -b
```

disassemble the current source line for the current frame
```
>disassemble --line
>di -l
```

Executable and Shared Library Query Commands

list the main executable and all dependent shared libraries
```
>image list
```

look up information for a raw address in the executable or any shared libraries
```
>image lookup --address 0x1ec4
>im loo -a 0x1ec4
```

look up functions matching a regular expression in a binary
```
>image lookup -r -n <FUNC_REGEX> (debug symbols)
>image lookup -r -s <FUNC_REGEX> (non-debug syms)
```

find full source line information
```
>image lookup -v --address 0x1ec4 (Look for entryline)
```

look up information for an address in a.out only
```
>image lookup --address 0x1ec4 a.out
>im loo -a 0x1ec4 a.out
```

look up information for a type Pointer by name
```
>image lookup --type Point
>im loo -t Point
```

dump all sections from the main executable and any shared libraries
```
>image dump sections
```

dump all sections in the a.out module
```
>image dump sections a.out
```

dump all symbols from the main executable and any shared libraries
```
>image dump symtab
```

dump all symbols in a.out and liba.so
```
>image dump symtab a.out liba.so
```

Miscellaneous

echo text to the screen
```
>script print "Here is some text"
```

remap source file pathnames for the debug session (e.g. if program was built on another PC)
```
>settings set target.source-map /buildbot/path /my/path
```

lldb的help

lldb的help：显示所有命令的帮助信息

```
(lldb) help
Debugger commands:
  apropos           -- List debugger commands related to a word or subject.
  breakpoint        -- Commands for operating on breakpoints (see 'help b' for
                       shorthand.)
  command           -- Commands for managing custom LLDB commands.
  disassemble       -- Disassemble specified instructions in the current
                       target.  Defaults to the current function for the
                       current thread and stack frame.
  expression        -- Evaluate an expression on the current thread.  Displays
                       any returned value with LLDB's default formatting.
  frame             -- Commands for selecting and examing the current thread's
                       stack frames.
  gdb-remote        -- Connect to a process via remote GDB server.  If no host
                       is specifed, localhost is assumed.
  gui               -- Switch into the curses based GUI mode.
  help              -- Show a list of all debugger commands, or give details
                       about a specific command.
  kdp-remote        -- Connect to a process via remote KDP server.  If no UDP
                       port is specified, port 41139 is assumed.
  language          -- Commands specific to a source language.
  log               -- Commands controlling LLDB internal logging.
  memory            -- Commands for operating on memory in the current target
                       process.
  platform          -- Commands to manage and create platforms.
  plugin            -- Commands for managing LLDB plugins.
  process           -- Commands for interacting with processes on the current
                       platform.
  quit              -- Quit the LLDB debugger.
  register          -- Commands to access registers for the current thread and
                       stack frame.
  reproducer        -- Commands for manipulating reproducers. Reproducers make
                       it possible to capture full debug sessions with all its
                       dependencies. The resulting reproducer is used to replay
                       the debug session while debugging the debugger.
                       Because reproducers need the whole the debug session
                       from beginning to end, you need to launch the debugger
                       in capture or replay mode, commonly though the command
                       line driver.
                       Reproducers are unrelated record-replay debugging, as
                       you cannot interact with the debugger during replay.
  script            -- Invoke the script interpreter with provided code and
                       display any results.  Start the interactive interpreter
                       if no code is supplied.
  session           -- Commands controlling LLDB session.
  settings          -- Commands for managing LLDB settings.
  source            -- Commands for examining source code described by debug
                       information for the current target process.
  statistics        -- Print statistics about a debugging session
```

```
  target            -- Commands for operating on debugger targets.
  thread            -- Commands for operating on one or more threads in the
                       current process.
  trace             -- Commands for loading and using processor trace
                       information.
  type              -- Commands for operating on the type system.
  version           -- Show the LLDB debugger version.
  watchpoint        -- Commands for operating on watchpoints.
Current command abbreviations (type 'help command alias' for more info):
  add-dsym  -- Add a debug symbol file to one of the target's current modules
               by specifying a path to a debug symbols file or by using the
               options to specify a module.
  attach    -- Attach to process by ID or name.
  b         -- Set a breakpoint using one of several shorthand formats.
  bt        -- Show the current thread's call stack.  Any numeric argument
               displays at most that many frames.  The argument 'all' displays
               all threads.  Use 'settings set frame-format' to customize the
               printing of individual frames and 'settings set thread-format'
               to customize the thread header.
  c         -- Continue execution of all threads in the current process.
  call      -- Evaluate an expression on the current thread.  Displays any
               returned value with LLDB's default formatting.
  continue  -- Continue execution of all threads in the current process.
  detach    -- Detach from the current target process.
  di        -- Disassemble specified instructions in the current target.
               Defaults to the current function for the current thread and
               stack frame.
  dis       -- Disassemble specified instructions in the current target.
               Defaults to the current function for the current thread and
               stack frame.
  display   -- Evaluate an expression at every stop (see 'help target
               stop-hook'.)
  down      -- Select a newer stack frame.  Defaults to moving one frame, a
               numeric argument can specify an arbitrary number.
  env       -- Shorthand for viewing and setting environment variables.
  exit      -- Quit the LLDB debugger.
  f         -- Select the current stack frame by index from within the current
               thread (see 'thread backtrace'.)
  file      -- Create a target using the argument as the main executable.
  finish    -- Finish executing the current stack frame and stop after
               returning.  Defaults to current thread unless specified.
  history   -- Dump the history of commands in this session.
               Commands in the history list can be run again using "!<INDEX>".
               "!-<OFFSET>" will re-run the command that is <OFFSET> commands
               from the end of the list (counting the current command).
  image     -- Commands for accessing information for one or more target
               modules.
  j         -- Set the program counter to a new address.
  jump      -- Set the program counter to a new address.
  kill      -- Terminate the current target process.
  l         -- List relevant source code using one of several shorthand formats.
  list      -- List relevant source code using one of several shorthand formats.
  n         -- Source level single step, stepping over calls.  Defaults to
               current thread unless specified.
  next      -- Source level single step, stepping over calls.  Defaults to
               current thread unless specified.
```

```
nexti     -- Instruction level single step, stepping over calls.  Defaults to
             current thread unless specified.
ni        -- Instruction level single step, stepping over calls.  Defaults to
             current thread unless specified.
p         -- Evaluate an expression on the current thread.  Displays any
             returned value with LLDB's default formatting.
parray    -- parray <COUNT> <EXPRESSION> -- lldb will evaluate EXPRESSION to
             get a typed-pointer-to-an-array in memory, and will display
             COUNT elements of that type from the array.
po        -- Evaluate an expression on the current thread.  Displays any
             returned value with formatting controlled by the type's author.
poarray   -- poarray <COUNT> <EXPRESSION> -- lldb will evaluate EXPRESSION to
             get the address of an array of COUNT objects in memory, and will
             call po on them.
print     -- Evaluate an expression on the current thread.  Displays any
             returned value with LLDB's default formatting.
q         -- Quit the LLDB debugger.
r         -- Launch the executable in the debugger.
rbreak    -- Sets a breakpoint or set of breakpoints in the executable.
re        -- Commands to access registers for the current thread and stack
             frame.
repl      -- Evaluate an expression on the current thread.  Displays any
             returned value with LLDB's default formatting.
run       -- Launch the executable in the debugger.
s         -- Source level single step, stepping into calls.  Defaults to
             current thread unless specified.
shell     -- Run a shell command on the host.
si        -- Instruction level single step, stepping into calls.  Defaults to
             current thread unless specified.
sif       -- Step through the current block, stopping if you step directly
             into a function whose name matches the TargetFunctionName.
step      -- Source level single step, stepping into calls.  Defaults to
             current thread unless specified.
stepi     -- Instruction level single step, stepping into calls.  Defaults to
             current thread unless specified.
t         -- Change the currently selected thread.
tbreak    -- Set a one-shot breakpoint using one of several shorthand formats.
undisplay -- Stop displaying expression at every stop (specified by stop-hook
             index.)
up        -- Select an older stack frame.  Defaults to moving one frame, a
             numeric argument can specify an arbitrary number.
v         -- Show variables for the current stack frame. Defaults to all
             arguments and local variables in scope. Names of argument,
             local, file static and file global variables can be specified.
             Children of aggregate variables can be specified such as
             'var- child.x'.  The -> and [] operators in 'frame variable' do
             not invoke operator overloads if they exist, but directly access
             the specified element.  If you want to trigger operator
             overloads use the expression command to print the variable
             instead.
             It is worth noting that except for overloaded operators, when
             printing local variables 'expr local_var' and 'frame var
             local_var' produce the same results.  However, 'frame variable'
             is more efficient, since it uses debug information and memory
             reads directly, rather than parsing and evaluating an
             expression, which may even involve JITing and running code in
```

```
                the target program.
  var         -- Show variables for the current stack frame. Defaults to all
                arguments and local variables in scope. Names of argument,
                local, file static and file global variables can be specified.
                Children of aggregate variables can be specified such as
                'var- child.x'.  The -> and [] operators in 'frame variable' do
                not invoke operator overloads if they exist, but directly access
                the specified element.  If you want to trigger operator
                overloads use the expression command to print the variable
                instead.
                It is worth noting that except for overloaded operators, when
                printing local variables 'expr local_var' and 'frame var
                local_var' produce the same results.  However, 'frame variable'
                is more efficient, since it uses debug information and memory
                reads directly, rather than parsing and evaluating an
                expression, which may even involve JITing and running code in
                the target program.
  vo          -- Show variables for the current stack frame. Defaults to all
                arguments and local variables in scope. Names of argument,
                local, file static and file global variables can be specified.
                Children of aggregate variables can be specified such as
                'var- child.x'.  The -> and [] operators in 'frame variable' do
                not invoke operator overloads if they exist, but directly access
                the specified element.  If you want to trigger operator
                overloads use the expression command to print the variable
                instead.
                It is worth noting that except for overloaded operators, when
                printing local variables 'expr local_var' and 'frame var
                local_var' produce the same results.  However, 'frame variable'
                is more efficient, since it uses debug information and memory
                reads directly, rather than parsing and evaluating an
                expression, which may even involve JITing and running code in
                the target program.
  x           -- Read from the memory of the current target process.
For more information on any command, type 'help <command-name>'.
```

lldb的help的用法解释

单个命令=子命令

单个命令的语法，可以用：

```
help <command name>
```

举例：

- `help register`

```
(lldb) help register
Commands to access registers for the current thread and stack frame.

Syntax: register [read write] ...
```

```
The following subcommands are supported:

      read  -- Dump the contents of one or more register values from the
               current frame.  If no register is specified, dumps them all.
      write -- Modify a single register value.

For more help on any particular subcommand, type 'help <command> <subcommand>'.
```

- `help memory`

```
(lldb) help memory
Commands for operating on memory in the current target process.

Syntax: memory <subcommand> [<subcommand-options>]

The following subcommands are supported:

      find    -- Find a value in the memory of the current target process.
      history -- Print recorded stack traces for allocation/deallocation events
                 associated with an address.
      read    -- Read from the memory of the current target process.
      region  -- Get information on the memory region containing an address in
                 the current target process.
      write   -- Write to the memory of the current target process.

For more help on any particular subcommand, type 'help <command> <subcommand>'.
```

单个命令的子命令=单个命令的参数

而命令的子命令的语法，也是前面加上help：

```
help <command> <subcommand>
```

举例：

- `help memory read`

```
(lldb) help memory read
Read from the memory of the current target process.

Syntax: memory read <cmd-options> <address-expression> [<address-expression>]

Command Options Usage:
  memory read [-drd] [-f <format>] [-c <count>] [-G <gdb-format>] [-s <byte-size>] [-l <number-
per-line>] [-o <filename>] <address-expression> [<address-expression>]
  memory read [-dbrd] [-f <format>] [-c <count>] [-s <byte-size>] [-o <filename>] <address-expr
ession> [<address-expression>]
  memory read [-AFLORTdrd] -t <name> [-f <format>] [-c <count>] [-G <gdb-format>] [-E <count>] [
-o <filename>] [-d <none>] [-S <boolean>] [-D <count>] [-P <count>] [-Y[<count>]] [-V <boolean>]
 [-Z <count>] <address-expression> [<address-expression>]
  memory read -t <name> [-x <source-language>] <address-expression> [<address-expression>]

       -A ( --show-all-children )
```

```
     Ignore the upper bound on the number of children to show.

-D <count> ( --depth <count> )
     Set the max recurse depth when dumping aggregate types (default is
     infinity).

-E <count> ( --offset <count> )
     How many elements of the specified type to skip before starting to
     display data.

-F ( --flat )
     Display results in a flat format that uses expression paths for
     each variable or member.

-G <gdb-format> ( --gdb-format <gdb-format> )
     Specify a format using a GDB format specifier string.

-L ( --location )
     Show variable location information.

-O ( --object-description )
     Display using a language-specific description API, if possible.

-P <count> ( --ptr-depth <count> )
     The number of pointers to be traversed when dumping values (default
     is zero).

-R ( --raw-output )
     Don't use formatting options.

-S <boolean> ( --synthetic-type <boolean> )
     Show the object obeying its synthetic provider, if available.

-T ( --show-types )
     Show variable types when dumping values.

-V <boolean> ( --validate <boolean> )
     Show results of type validators.

-Y[<count>] ( --no-summary-depth=[<count>] )
     Set the depth at which omitting summary information stops (default
     is 1).

-Z <count> ( --element-count <count> )
     Treat the result of the expression as if its type is an array of
     this many values.

-b ( --binary )
     If true, memory will be saved as binary. If false, the memory is
     saved save as an ASCII dump that uses the format, size, count and
     number per line settings.

-c <count> ( --count <count> )
     The number of total items to display.

-d <none> ( --dynamic-type <none> )
```

```
            Show the object as its full dynamic type, not its static type, if
            available.
            Values: no-dynamic-values | run-target | no-run-target

        -f <format> ( --format <format> )
            Specify a format to be used for display.

        -l <number-per-line> ( --num-per-line <number-per-line> )
            The number of items per line to display.

        -o <filename> ( --outfile <filename> )
            Specify a path for capturing command output.

        -r ( --force )
            Necessary if reading over target.max-memory-read-size bytes.

        -s <byte-size> ( --size <byte-size> )
            The size in bytes to use when displaying with the selected format.

        -t <name> ( --type <name> )
            The name of a type to view memory as.

        -x <source-language> ( --language <source-language> )
            The language of the type to view memory as.

        -d ( --append-outfile )
            Append to the file specified with '--outfile <path>'.

    This command takes options and free-form arguments.  If your arguments
    resemble option specifiers (i.e., they start with a - or --), you must use
    ' -- ' between the end of the command options and the beginning of the
    arguments.
```

常用命令

image

TODO：

- 【记录】lldb命令使用心得：image
- 【已解决】lldb命令使用心得：image
  - 【已解决】lldb命令使用心得：image lookup

register

TODO：

【记录】lldb命令使用心得：register

expression

TODO：

- 【记录】lldb命令使用心得：expression
- 【记录】lldb命令使用心得：p和po
  - 【整理】Xcode中lldb命令对比：po和p

p

po

memory

TODO：

【记录】lldb命令使用心得：memory

disassemble

TODO：

【记录】lldb命令使用心得：disassemble

thread

TODO：

【记录】lldb命令使用心得：thread

frame

TODO：

【记录】lldb命令使用心得：frame

breakpoint

TODO：

【记录】lldb命令使用心得：breakpoint

watchpoint

TODO：

- 【已解决】Xcode中lldb中如何给watchpoint加上条件判断过滤
- 【已解决】Xcode中lldb的条件watchpoint报错：error user expression indirection requires pointer operand long invalid
-
- 【已解决】Xcode的lldb中如何监控结构体变量值的变化
- 【未解决】研究YouTube逻辑：监控NSArray的_allTrackRenderers值被改动
- 【未解决】YouTube的HAMPlayerInternal的playerLoop中监控_currentTime变量值变化

---

help语法：

```
(lldb) help watchpoint
Commands for operating on watchpoints.

Syntax: watchpoint <subcommand> [<command-options>]

The following subcommands are supported:

      command -- Commands for adding, removing and examining LLDB commands
                 executed when the watchpoint is hit (watchpoint 'commands').
      delete  -- Delete the specified watchpoint(s).  If no watchpoints are
                 specified, delete them all.
      disable -- Disable the specified watchpoint(s) without removing it/them.
                 If no watchpoints are specified, disable them all.
      enable  -- Enable the specified disabled watchpoint(s). If no watchpoints
                 are specified, enable all of them.
      ignore  -- Set ignore count on the specified watchpoint(s).  If no
                 watchpoints are specified, set them all.
      list    -- List all watchpoints at configurable levels of detail.
      modify  -- Modify the options on a watchpoint or set of watchpoints in
                 the executable.  If no watchpoint is specified, act on the
                 last created watchpoint.  Passing an empty argument clears the
                 modification.
      set     -- Commands for setting a watchpoint.

For more help on any particular subcommand, type 'help <command> <subcommand>'.
```

用法举例：

```
(lldb) watchpoint set expr 0x000000011ceb5818
Watchpoint created: Watchpoint 1: addr = 0x11ceb5818 size = 8 state = enabled type = w
    new value: 0
```

触发时打印：

```
Watchpoint 1 hit:
old value: 0
new value: 0
```

关闭所有：

```
(lldb) watchpoint disable
All watchpoints disabled. (4 watchpoints)
```

打开所有：

```
(lldb) watchpoint enable
All watchpoints enabled. (4 watchpoints)
```

调试控制

run

next

nexti

step

stepi

jump

finish

exit

LLDB心得

命令缩写

lldb命令的缩写：所有命令都支持任意前缀字符的缩写，只要不产生混淆

lldb中的命令，可以缩写

比如：

```
print
```

常见缩写是：

```
p
```

但其实底层逻辑是：

从第一个字母 `p` 到最后一个字母 `t` ，缩写到任意位置都是可以的

前提是，只要不（和其他命令的前缀）产生混淆

举例

print

- `print`
  - `p`
    - 是 `lldb` 专门为 `print` 保留的 `p` ，所以可以用 `p`
      - 否则按道理，也会和其他 `process` 等命令产生冲突，也不能把 `print` 缩写为 `p`
  - `pr`
    - 和 `process` 的 `pr` 是一样的前缀字符
      - `lldb` 无法确定是哪个，所以就属于会产生冲突、混淆
      - 所以不能用 `pr`
  - `pri`
    - 可以
  - `prin`
    - 可以
  - `print`
    - 可以 所以总体结论就是：
- `print`
  - 可以用特定的缩写： `p`
  - 也可以用其他普通的，不产生冲突的缩写： `pri` 、 `prin` 、 `print`

breakpoint

断点de的命令

```
breakpoint
```

可以缩写/简写为：

```
breakpoin
breakpoi
breakpo
breakp
break
brea
bre
br
```

而不能用：

- `b`
  - 特殊：属于lldb中专门保留的特定的缩写
    - 含义是：以某种特定的格式去添加断点

-》有了上面的缩写逻辑，则普通的：

```
breakpoint list
```

就可以写为：

```
breakpoin list
breakpoi list
breakpo list
breakp list
break list
brea list
bre list
br list
```

都是可以的，都是等价的

子命令也支持缩写

当然命令的子命令，参数，也是同样支持缩写

比如此处

```
br list
```

的 `list` 也可以缩写：

```
br lis
br li
br l
```

只要不产生冲突即可

此处就是 `breakpoint` 的子命令中，上述缩写不会冲突混淆即可。

注：

此处可以用 `help breakpoint` 去查看， `breakpoint` 有哪些子命令

```
(lldb) help breakpoint
Commands for operating on breakpoints (see 'help b' for shorthand.)

Syntax: breakpoint <subcommand> [<command-options>]

The following subcommands are supported:

      clear   -- Delete or disable breakpoints matching the specified source
                 file and line.
      command -- Commands for adding, removing and listing LLDB commands
                 executed when a breakpoint is hit.
      delete  -- Delete the specified breakpoint(s).  If no breakpoints are
                 specified, delete them all.
      disable -- Disable the specified breakpoint(s) without deleting them.  If
                 none are specified, disable all breakpoints.
      enable  -- Enable the specified disabled breakpoint(s). If no breakpoints
                 are specified, enable all of them.
      list    -- List some or all breakpoints at configurable levels of detail.
      modify  -- Modify the options on a breakpoint or set of breakpoints in
                 the executable.  If no breakpoint is specified, acts on the
                 last created breakpoint.  With the exception of -e, -d and -i,
                 passing an empty argument clears the modification.
      name    -- Commands to manage name tags for breakpoints
      read    -- Read and set the breakpoints previously saved to a file with
                 "breakpoint write".
      set     -- Sets a breakpoint or set of breakpoints in the executable.
      write   -- Write the breakpoints listed to a file that can be read in
                 with "breakpoint read".  If given no arguments, writes all
                 breakpoints.

For more help on any particular subcommand, type 'help <command> <subcommand>'.
```

其中可见， `breakpoint` 的子命令：

- `clear`
- `command`
- `delete`
- `disable`
- `enable`
- `modify`
- `name`
- `read`
- `set`
- `write`

不会和上面的缩写 `lis` 、 `li` 、 `l` ，有冲突和混淆

-》所以你会看到，很多人常把：

```
breakpoint list
```

写成：

```
br l
```

就是这个目的：

- 尽量用缩写
  - -》减少输入的字符数
    - -》提高调试效率

其他常见缩写

其他常用缩写：

- `expression` -> `e` 、 `exp`
- `disassemble` -> `dis`
- `register` -> `reg`
  - `register read` -> `reg r`
- `image` -> `im`
- `memory` -> `mem`

Xcode中lldb

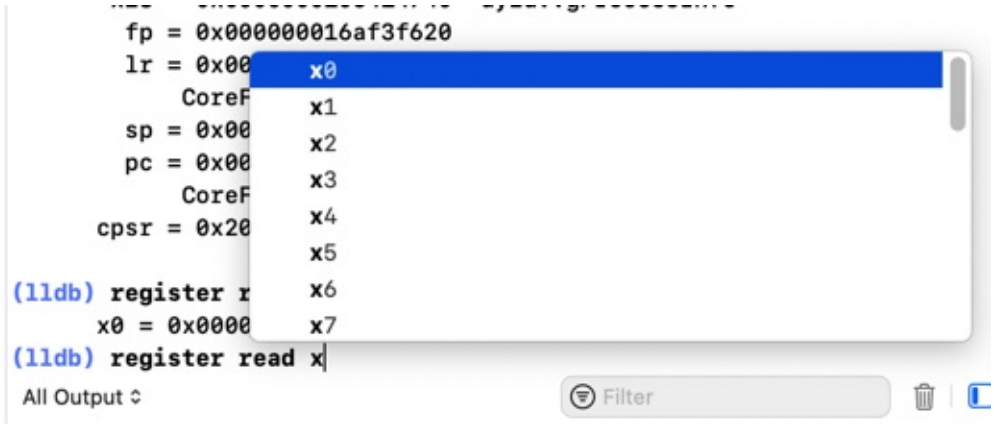
TODO：

- 【整理】Xcode的lldb调试心得：F7单步进入无名的汇编代码
- 【未解决】XCode和lldb如何根据函数地址加断点
- 【已解决】XCode的lldb中如何调试运行iOS的ObjC代码
- 【已解决】XCode和lldb调试常见用法和调试心得
- 【已解决】XCode的lldb中如何调试找到当前函数_dyld_get_image_name的返回值

此处整理 `Xcode` 中的 `lldb` 的一些心得：

支持自动补全

`Xcode` 中 `lldb` 中支持自动补全：



查看函数调用堆栈

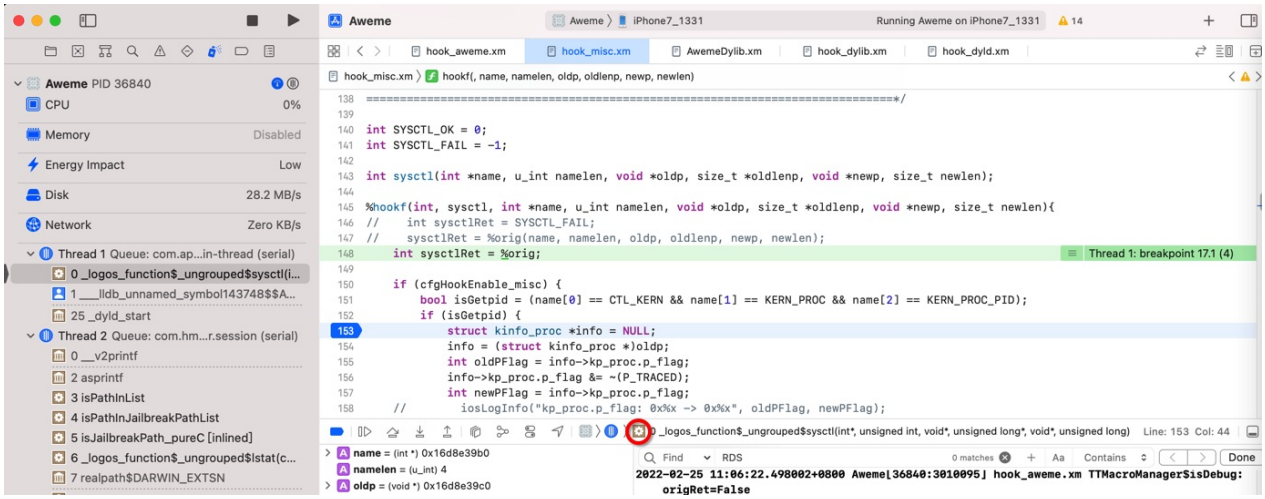
`lldb` 和 `XCode` 中查看 `函数调用堆栈` = `backtrace` ：

XCode调试期间，想要查看：函数调用堆栈

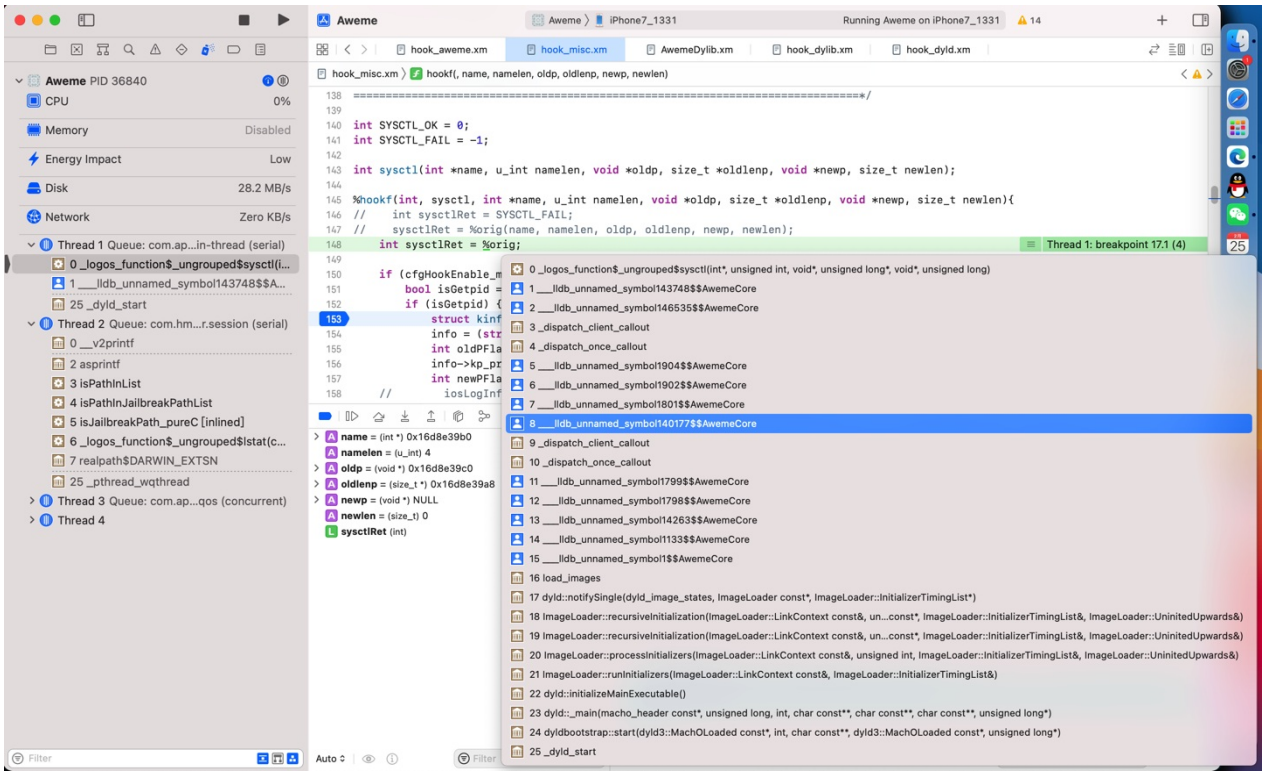
至少有2种方法：

XCode的UI界面中

`XCode` 中，`Command` + 鼠标单击：



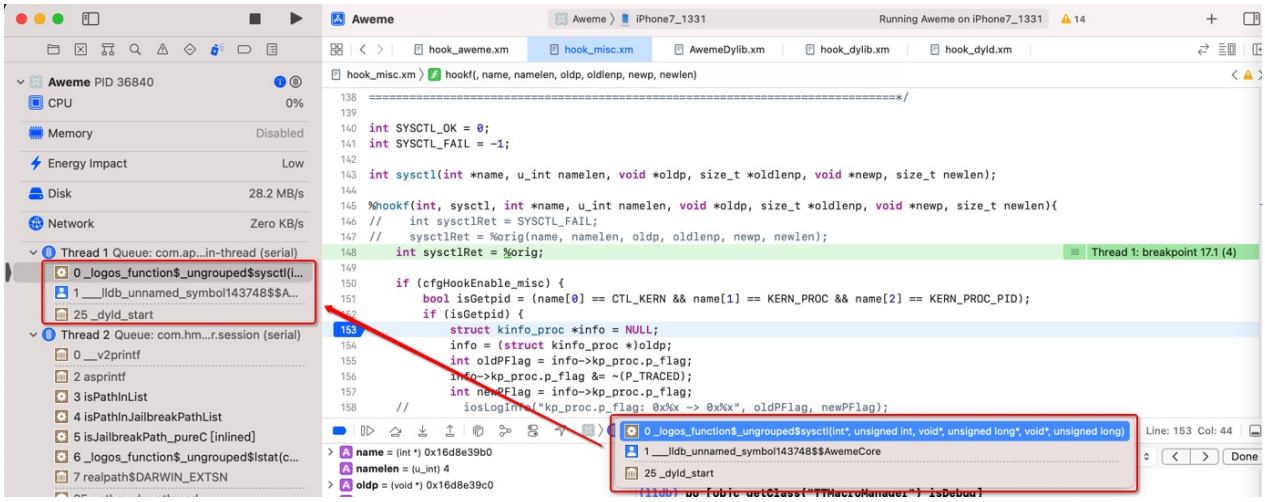
即可看到全部的函数调用堆栈：



注：

直接 `鼠标点击` （不加 `Command` 键），则只显示缩略后的信息：

且和 `Debug Navigator` 中的线程下面的 函数调用堆栈 简略信息 是一致的：



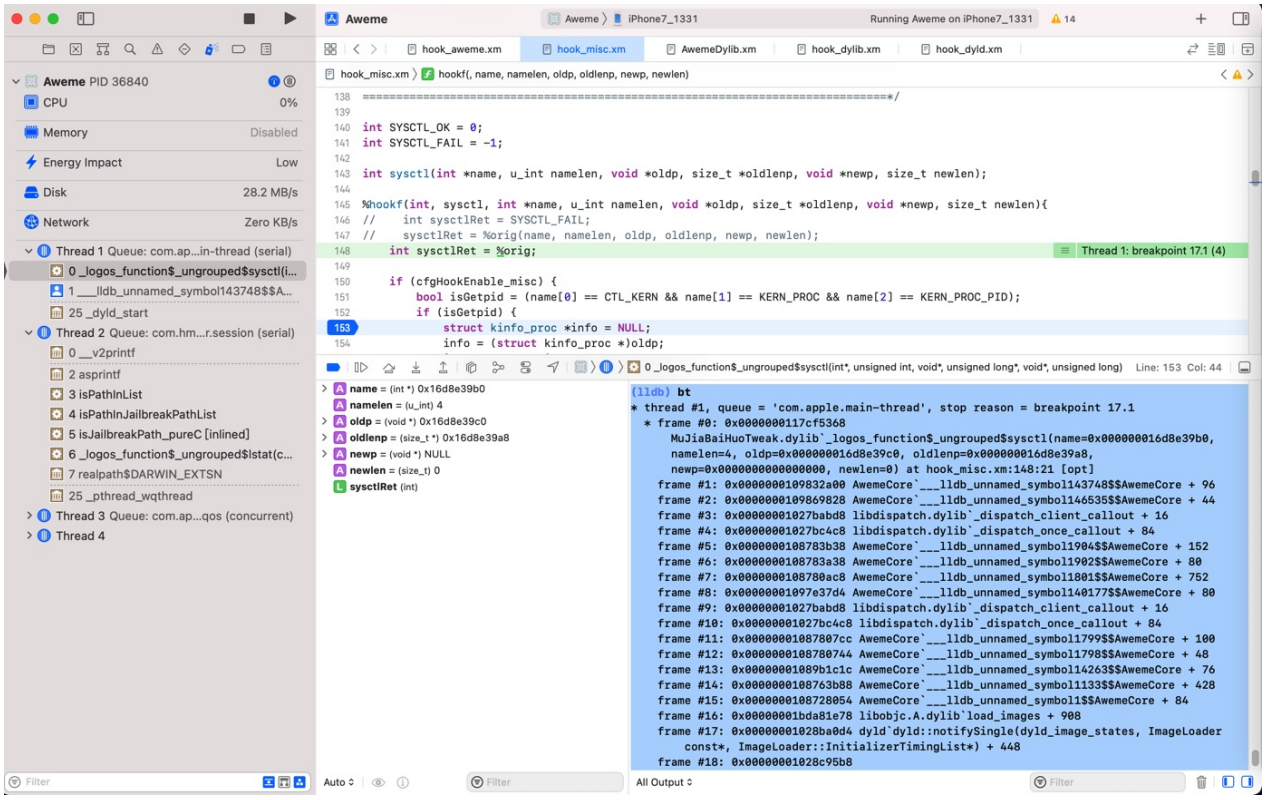
lldb命令bt

- `bt` = `thread backtrace`
  - = `th b`
  - = `th ba`

举例：

```
(lldb) bt
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 17.1
  * frame #0: 0x0000000117cf5368 MuJiaBaiHuoTweak.dylib`_logos_function$_ungrouped$sysctl(name=
0x000000016d8e39b0, namelen=4, oldp=0x000000016d8e39c0, oldlenp=0x000000016d8e39a8, newp=0x0000
000000000000, newlen=0) at hook_misc.xm:148:21 [opt]
    frame #1: 0x0000000109832a00 AwemeCore`___lldb_unnamed_symbol143748$$AwemeCore + 96
    frame #2: 0x0000000109869828 AwemeCore`___lldb_unnamed_symbol146535$$AwemeCore + 44
    frame #3: 0x00000001027babd8 libdispatch.dylib`_dispatch_client_callout + 16
    frame #4: 0x00000001027bc4c8 libdispatch.dylib`_dispatch_once_callout + 84
    frame #5: 0x0000000108783b38 AwemeCore`___lldb_unnamed_symbol1904$$AwemeCore + 152
    frame #6: 0x0000000108783a38 AwemeCore`___lldb_unnamed_symbol1902$$AwemeCore + 80
    frame #7: 0x00000001087780ac8 AwemeCore`___lldb_unnamed_symbol1801$$AwemeCore + 752
    frame #8: 0x00000001097e37d4 AwemeCore`___lldb_unnamed_symbol140177$$AwemeCore + 80
    frame #9: 0x00000001027babd8 libdispatch.dylib`_dispatch_client_callout + 16
    frame #10: 0x00000001027bc4c8 libdispatch.dylib`_dispatch_once_callout + 84
    frame #11: 0x00000001087807cc AwemeCore`___lldb_unnamed_symbol1799$$AwemeCore + 100
    frame #12: 0x0000000108780744 AwemeCore`___lldb_unnamed_symbol1798$$AwemeCore + 48
    frame #13: 0x00000001089b1c1c AwemeCore`___lldb_unnamed_symbol14263$$AwemeCore + 76
    frame #14: 0x0000000108763b88 AwemeCore`___lldb_unnamed_symbol1133$$AwemeCore + 428
    frame #15: 0x0000000108728054 AwemeCore`___lldb_unnamed_symbol1$$AwemeCore + 84
    frame #16: 0x00000001bda81e78 libobjc.A.dylib`load_images + 908
    frame #17: 0x00000001028ba0d4 dyld`dyld::notifySingle(dyld_image_states, ImageLoader const*,
 ImageLoader::InitializerTimingList*) + 448
    frame #18: 0x00000001028c95b8 dyld`ImageLoader::recursiveInitialization(ImageLoader::LinkCo
ntext const&, unsigned int, char const*, ImageLoader::InitializerTimingList&, ImageLoader::Unin
itedUpwards&) + 524
    frame #19: 0x00000001028c953c dyld`ImageLoader::recursiveInitialization(ImageLoader::LinkCo
ntext const&, unsigned int, char const*, ImageLoader::InitializerTimingList&, ImageLoader::Unin
itedUpwards&) + 400
    frame #20: 0x00000001028c8334 dyld`ImageLoader::processInitializers(ImageLoader::LinkContex
t const&, unsigned int, ImageLoader::InitializerTimingList&, ImageLoader::UninitedUpwards&) + 1
```

```
84
    frame #21: 0x00000001028c83fc dyld`ImageLoader::runInitializers(ImageLoader::LinkContext co
nst&, ImageLoader::InitializerTimingList&) + 92
    frame #22: 0x00000001028ba420 dyld`dyld::initializeMainExecutable() + 216
    frame #23: 0x00000001028bedb4 dyld`dyld::_main(macho_header const*, unsigned long, int, char
 const**, char const**, char const**, unsigned long*) + 4616
    frame #24: 0x00000001028b9208 dyld`dyldbootstrap::start(dyld3::MachOLoaded const*, int, char
 const**, dyld3::MachOLoaded const*, unsigned long*) + 396
    frame #25: 0x00000001028b9038 dyld`_dyld_start + 56
```



iOS逆向

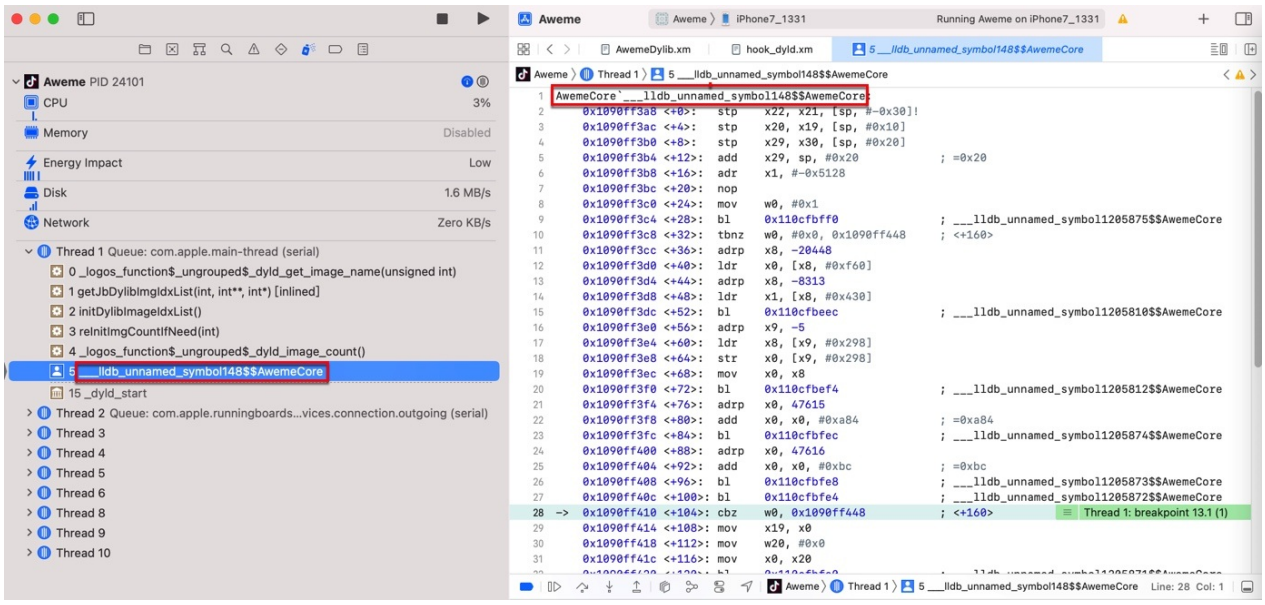
TODO：

- 整理常用的命令和举例
  - image
  - po
  - bt
  - reg
  - 等
- iOS逆向时用LLDB调试iOS中ObjC的对象和相关内容
- 【记录】iOS逆向Xcode调试心得：bl后cmn再b.eq很像是switch case或if else的代码逻辑跳转
- 【未解决】Xcode的lldb调试iOS的ObjC或Swift时如何打印出objc_msgSend第一个参数是什么类的实例

无名函数

iOS逆向期间，往往可以看到这种函数名：

```
AwemeCore`___lldb_unnamed_symbol148$$AwemeCore
```



其实就是个：`无名函数`

完整的解释是：

- `AwemeCore` ___lldb_unnamed_symbol148$$AwemeCore`
  - `AwemeCore`：函数所属于的（哪个）二进制
    - 注：理论上，同一个函数，可能会出现在多个二进制中
  - `___lldb_unnamed_symbol148$$AwemeCore`
    - 无名函数
      - `___lldb_unnamed_symbol148`：函数名的部分

        - `AwemeCore`：二进制的名字

-》由此可以总结出：

- lldb中的无名函数的命名规则
  - ___lldb_unnamed_symbolNNN$$BinaryName
    - `___lldb_unnamed_symbol148$$AwemeCore`
      - `NNN = 148`
        - 从1开始编号
      - `BinaryName = AwemeCore`
        - 对应着当前lldb正在调试的二进制是 `AwemeCore`

-》

- 知道这个能干什么？
  - 后续去给某个无名函数去加断点时，要注意把函数名写完整了，不要漏写成：
    - `___lldb_unnamed_symbol148`
      - 否则是无法触发断点的
  - 要写成完整的函数名：
    - `___lldb_unnamed_symbol148$$AwemeCore`
      - 才能正常触发断点

LLVM

此处也顺带去整理 `LLDB` 所属的开源项目 `LLVM` 的相关内容：

- `LLVM`
  - = `Low Level Virtual Machine`
  - 是什么=一句话描述
    - 一套用于构建出高度优化的编译器、优化器、运行环境的工具集合的开源项目
      - a toolkit for the construction of highly optimized compilers, optimizers, and runtime environments.
  - 主要包含3个部分
    - `LLVM套件` = `LLVM Suite`
      - 包含各种
        - 工具
          - `汇编器` = `assembler`
          - `反汇编器` = `disassembler`
          - `位码分析器` = `bitcode analyzer`
          - `位码优化器` = `bitcode optimizer`
          - 简单的回归测试
            - 用于测试LLVM工具和Clang前端
        - 库
        - 头文件
    - `Clang` = `Clang前端` = `Clang front end`
      - 是什么：LLVM的内置的原生的 `C` / `C++` / `Objective-C` 编译器
      - 可以把 `C` ，`C++` ，`Objective-C` 和 `Objective-C++` 的代码，编译成 `LLVM bitcode`
        - 然后就可以用LLVM套件去操作此（编译后的）程序了
    - `测试套件` = `Test Suite`
      - 一堆工具的集合
        - 测试LLVM的功能和性能
  - 子项目
    - `LLVM Core` libraries
      - a modern source- and target-independent optimizer, along with code generation support for many popular CPUs
    - `Clang`
      - an `LLVM native` C/C++/Objective-C compiler
    - `LLDB`
      - a great native debugger
        - 基于 `LLVM` 和 `Clang`
    - `libc++` 和 `libc++ ABI`
      - a standard conformant and high-performance implementation of the C++ Standard Library
        - including full support for C++11 and C++14
    - `compiler-rt`
      - provides highly tuned implementations of the low-level code generator

        - `MLIR`
            - a novel approach to building reusable and extensible compiler infrastructure
        - `OpenMP`
            - an OpenMP runtime for use with the OpenMP implementation in Clang
        - `polly`
            - a suite of cache-locality optimizations as well as auto-parallelism and vectorization using a polyhedral model
        - `libclc`
            - implement the OpenCL standard library
        - `klee`
            - implements a "symbolic virtual machine" which uses a theorem prover to try to evaluate all dynamic paths through a program in an effort to find bugs and to prove properties of functions
        - `LLD`
            - a new linker
                - a drop-in replacement for system linkers and runs much faster
    - 资料
        - 官网
            - The LLVM Compiler Infrastructure Project
                - https://llvm.org
        - 快速上手
            - Getting Started with the LLVM System — LLVM 12 documentation
                - https://llvm.org/docs/GettingStarted.html
    - 相关
        - 概念
            - `IR` = `Intermediate Representation` = `中间表示层`

附录

下面列出相关参考资料。

文档

- 官网
  - LLDB Homepage — The LLDB Debugger
    - http://lldb.llvm.org
- 教程
  - Tutorial — The LLDB Debugger
    - https://lldb.llvm.org/use/tutorial.html
- LLDB和GDB命令对比
  - GDB to LLDB command map — The LLDB Debugger
    - https://lldb.llvm.org/use/map.html
      - 背景：由于GDB使用更广泛，所以LLDB为了让从GDB转过来的人，更快上手，而整理了GDB命令到LLDB命令的映射的文档，介绍的很详细，值得参考

参考资料

- lldb调试器知多少 - 掘金 (juejin.cn)
- LLDB调试器使用简介 | 南峰子的技术博客 (southpeak.github.io)
- ObjC 中国 - 与调试器共舞 - LLDB 的华尔兹 (objccn.io)
- GDB to LLDB command map — The LLDB Debugger
- LLDB 调试命令使用指南 - 链滴 (ld246.com)
- LLDB Homepage — The LLDB Debugger (llvm.org)
- Tutorial — The LLDB Debugger (llvm.org)
- LLDB (debugger) - Wikipedia)
- Dancing in the Debugger — A Waltz with LLDB · objc.io
- lldb cheat sheet
-