[译]用 LLDB 调试 Swift 代码

> - 原文地址：Debugging Swift code with LLDB
> - 原文作者：Ahmed Sulaiman
> - 译文出自：掘金翻译计划
> - 本文永久链接：github.com/xitu/gold-m...
> - 译者：VernonVan
> - 校对者：ZhiyuanSun、Danny1451

用 LLDB 调试 Swift 代码

作为工程师，我们花了差不多 70% 的时间在调试上，剩下的 20% 用来思考架构以及和组员沟通，仅仅只有 10% 的时间是真的在写代码的。

> 调试就像是在犯罪电影中做侦探一样，同时你也是凶手。
>
> — Filipe Fortes 来自 Twitter

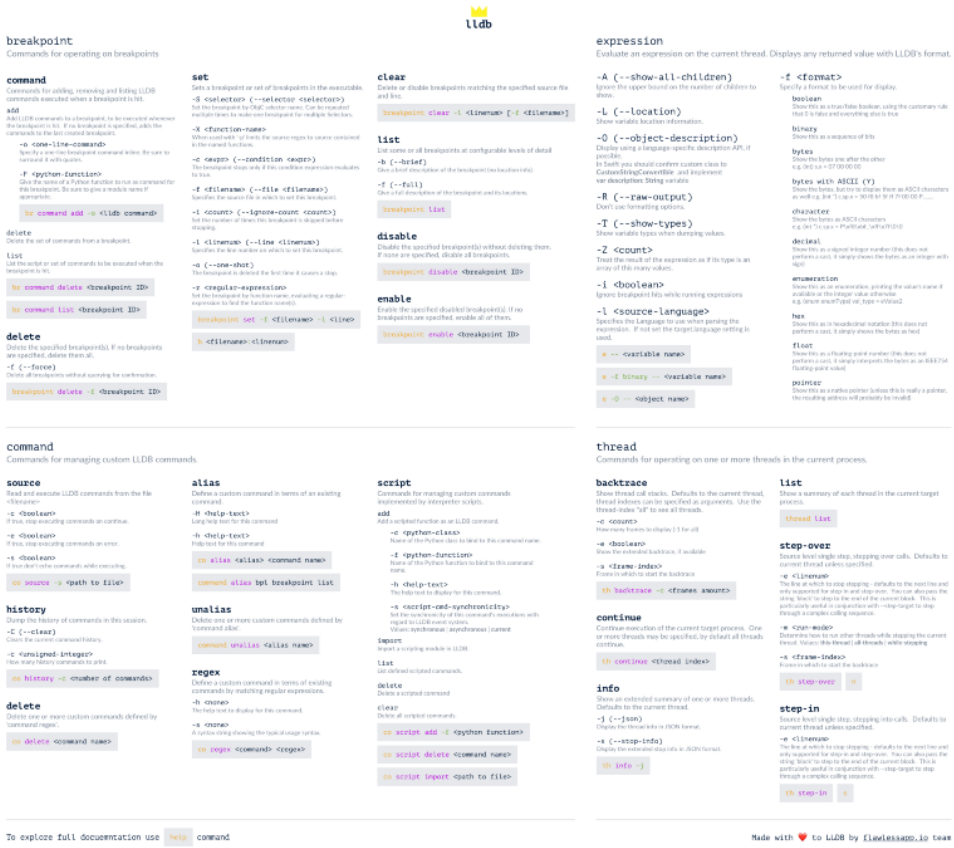
所以让我们在这70%的时间尽可能愉悦是相当重要的。LLDB 就是来打救我们的。奇妙的 Xcode Debugger UI 展示了所有你可用的信息，而不用敲入任何一个 LLDB 命令。然而，控制台在我们的工作中同样也是很重要的一部分。现在让我们来分析一些最有用的 LLDB 技巧。我自己每天都在用它们进行调试。

从哪里开始呢?

LLDB 是一个庞大的工具，内置了很多有用的命令。我不会全部讲解，而是带你浏览最有用的命令。这是我们的计划：

1. 获取变量值：`expression`, `e`, `print`, `po`, `p`
2. 获取整个应用程序的状态以及特定语言的命令：`bugreport`, `frame`, `language`
3. 控制应用的执行流程：`process`, `breakpoint`, `thread`, `watchpoint`
4. 荣誉奖：`command`, `platform`, `gui`

我还准备好了有用的 LLDB 命令说明和实例的表格，有需要的可以把它贴在 Mac 上面记住这些命令 🤖



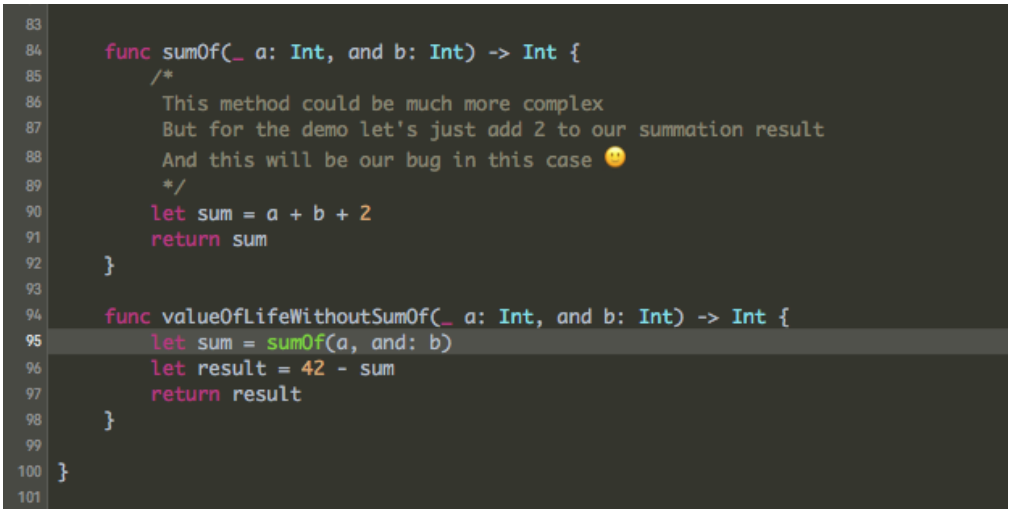
通过这条链接下载全尺寸的版本 — www.dropbox.com/s/9sv67e7f2…

1. 获取变量值和状态

命令：`expression`, `e`, `print`, `po`, `p`

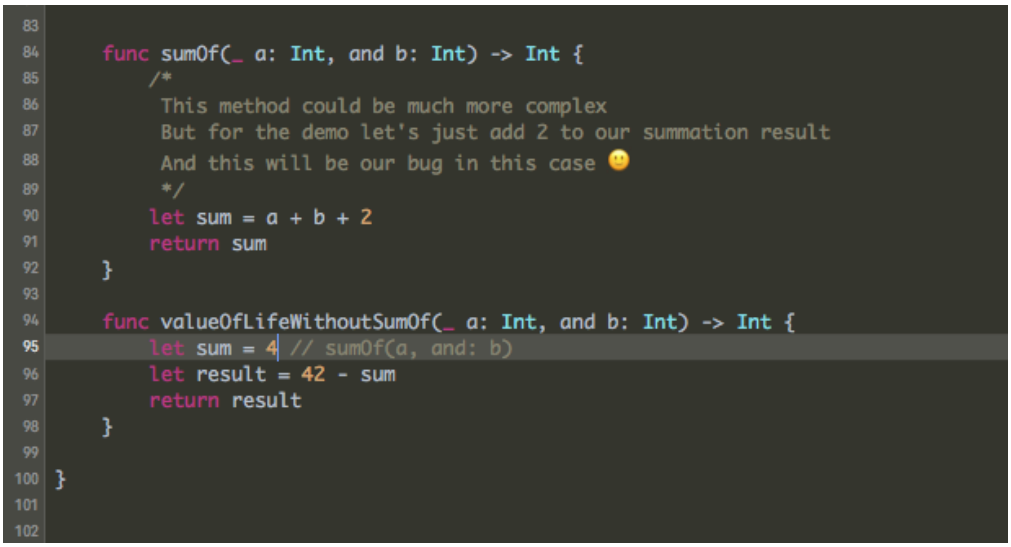
调试器的一个基础功能就是获取和修改变量的值。这就是 `expression` 或者 `e` 被创造的原因（当然他们还有更高级的功能）。您可以简单的在运行时执行任何表达式或命令。

假设你现在正在调试方法 `valueOfLifeWithoutSumOf()` ：对两个数求和，再用42去减得到结果。



```
83
84      func sumOf(_ a: Int, and b: Int) -> Int {
85          /*
86           This method could be much more complex
87           But for the demo let's just add 2 to our summation result
88           And this will be our bug in this case 🙂
89           */
90          let sum = a + b + 2
91          return sum
92      }
93
94      func valueOfLifeWithoutSumOf(_ a: Int, and b: Int) -> Int {
95          let sum = sumOf(a, and: b)
96          let result = 42 - sum
97          return result
98      }
99
100 }
101
```

继续假设你一直得到错误的结果并且你并不知道是什么原因。所以你可以做以下的事来找到问题：



```
83
84      func sumOf(_ a: Int, and b: Int) -> Int {
85          /*
86           This method could be much more complex
87           But for the demo let's just add 2 to our summation result
88           And this will be our bug in this case 🙂
89           */
90          let sum = a + b + 2
91          return sum
92      }
93
94      func valueOfLifeWithoutSumOf(_ a: Int, and b: Int) -> Int {
95          let sum = 4 // sumOf(a, and: b)
96          let result = 42 - sum
97          return result
98      }
99
100 }
101
102
```

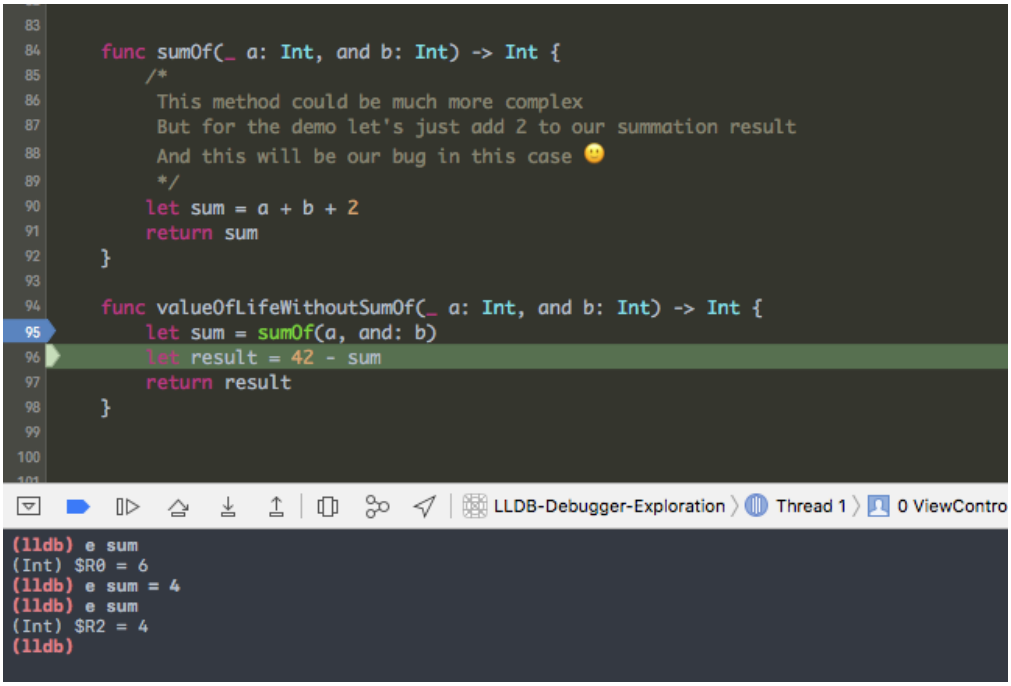
或者。。。使用 LLDB 表达式在运行时修改值才是更好的方法，同时可以找出问题是在哪里出现的。首先，在你感兴趣的地方设置一个断点，然后运行你的应用。

为了用 LLDB 格式打印指定的变量你应该调用：

```
(lldb) e <variable>
```

使用相同的命令来执行一些表达式：

```
(lldb) e <expression>
```




```
(lldb) e sum
(Int) $R0 = 6 // 下面你也可以用 $R0 来引用这个变量（在本次调试过程中）

(lldb) e sum = 4 // 修改变量 sum 的值

(lldb) e sum
(Int) $R2 = 4 // 直到本次调试结束变量 sum 都会是 "4"
```

`expression` 命令也有一些标志。在 `expression` 后面用双破折号 `--` 将标志和实际的表达式分隔开，就像这样：

```
(lldb) expression <some flags> -- <variable>
```

`expression` 命令差不多有30种不同的标志。我鼓励你多去探索它们。在终端中键入以下命令可以看到完整的文档：

```
> lldb
> (lldb) help # 获取所有变量的命令
> (lldb) help expression # 获取所有表达式的子命令
```

我会在下列 `expression` 的标志上多停留一会儿：

- `-D <count>` (`--depth <count>`) — 设置在转储聚合类型时的最大递归深度（默认为无穷大）。
- `-O` (`--object-description`) — 如果可能的话，使用指定语言的描述API来显示。
- `-T` (`--show-types`) — 在转储值的时候显示变量类型。
- `-f <format>` (`--format <format>`) — 指定一种用于显示的格式。
- `-i <boolean>` (`--ignore-breakpoints <boolean>`)— 在运行表达式时忽略断点。

假设我们有一个叫 `logger` 的对象，这个对象有一些字符串和结构体类型的属性。比如说，你可能只是想知道第一层的属性，那只需要用 `-D` 标志以及恰当的层级深度值，就像这样：

```
(lldb) e -D 1 -- logger

(LLDB_Debugger_Exploration.Logger) $R5 = 0x0000608000087e90 {
  currentClassName = "ViewController"
  debuggerStruct ={...}
}
```

默认情况下，LLDB 会无限地遍历该对象并且给你展示每个嵌套的对象的完整描述：

```
(lldb) e -- logger

(LLDB_Debugger_Exploration.Logger) $R6 = 0x0000608000087e90 {
  currentClassName = "ViewController"
  debuggerStruct = (methodName = "name", lineNumber = 2, commandCounter = 23)
}
```

你也可以用 `e -O --` 获取对象的描述或者更简单地用别名 `po`，就像下面的示例一样：

```
(lldb) po logger

<Logger: 0x608000087e90>
```

并不是很有描述性，不是吗？为了获取更加可阅读的描述，你自定义的类必须遵循 `CustomStringConvertible` 协议，同时实现 `var description: String { return ...}` 属性。接下来只需要用 `po` 就能返回可读的描述。

在本节的开始，我也提到了 `print` 命令。基本上 `print <expression/variable>` 就等同于 `expression -- <expression/variable>`。但是 `print` 命令不能带任何标志或者额外的参数。

2. 获取整个 APP 的状态和指定语言的命令

`bugreport`, `frame`, `language`

你是否经常复制粘贴崩溃日志到任务管理器中方便稍后能考虑这个问题吗？LLDB 提供了一个很好用的命令叫 `bugreport`，这个命令能生成当前应用状态的完整报告。在你偶然触发某些问题但是想在稍后再解决它时这个命令就会很有帮助了。为了能恢复应用的状态，你可以使用 `bugreport` 生成报告。

```
(lldb) bugreport unwind --outfile <path to output file>
```

最终的报告看起来就像下面截图中的例子一样：

```
(lldb)  thread backtrace
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 2.1
  * frame #0: 0x000000010bbe4609 LLDB-Debugger-Exploration`ViewController.viewDidLoad(self=0x00007fa0c1406900) -> () at ViewController.swift:60
    frame #1: 0x000000010bbe49e2 LLDB-Debugger-Exploration`@objc ViewController.viewDidLoad() -> () at ViewController.swift:0
    frame #2: 0x000000010c83801a UIKit`-[UIViewController loadViewIfRequired] + 1235
    frame #3: 0x000000010c83845a UIKit`-[UIViewController view] + 27
    frame #4: 0x000000010c70098a UIKit`-[UIWindow addRootViewControllerViewIfPossible] + 65
    frame #5: 0x000000010c701070 UIKit`-[UIWindow _setHidden:forced:] + 294
    frame #6: 0x000000010c713ebe UIKit`-[UIWindow makeKeyAndVisible] + 42
    frame #7: 0x000000010c68d37f UIKit`-[UIApplication _callInitializationDelegatesForMainScene:transitionContext:] + 4346
    frame #8: 0x000000010c6935e4 UIKit`-[UIApplication _runWithMainScene:transitionContext:completion:] + 1709
    frame #9: 0x000000010c6907f3 UIKit`-[UIApplication workspaceDidEndTransaction:] + 182
    frame #10: 0x000000011062d5f6 FrontBoardServices`__FBSSERIALQUEUE_IS_CALLING_OUT_TO_A_BLOCK__ + 24
    frame #11: 0x000000011062d46d FrontBoardServices`-[FBSSerialQueue _performNext] + 186
    frame #12: 0x000000011062d7f6 FrontBoardServices`-[FBSSerialQueue _performNextFromRunLoopSource] + 45
    frame #13: 0x000000010eec3c01 CoreFoundation`__CFRUNLOOP_IS_CALLING_OUT_TO_A_SOURCE0_PERFORM_FUNCTION__ + 17
    frame #14: 0x000000010eea90cf CoreFoundation`__CFRunLoopDoSources0 + 527
    frame #15: 0x000000010eea85ff CoreFoundation`__CFRunLoopRun + 911
    frame #16: 0x000000010eea8016 CoreFoundation`CFRunLoopRunSpecific + 406
```

`bugreport` 命令输出的示例。

frame

Commands for selecting and examing the current thread's stack frames.

`(lldb) frame ...` = `(lldb) fr ...`

假设你想要获取当前线程的当前栈帧的概述，`frame` 命令可以帮你完成：

```
93
94      func valueOfLifeWithoutSumOf(_ a: Int, and b: Int) -> Int {
95          let sum = sumOf(a, and: b)
96          let result = 42 - sum
97          return result
98      }
99
100
```

LLDB-Debugger-Exploration 〉 Thread 1 〉 0 ViewController.

使用下面的代码片段来快速获取当前地址以及当前的环境条件：

```
(lldb) frame info

frame #0: 0x000000010bbe4b4d LLDB-Debugger-
Exploration`ViewController.valueOfLifeWithoutSumOf(a=2, b=2,
self=0x00007fa0c1406900) -> Int at ViewController.swift:96
```

这些信息在本文后面将要说到的断点管理中非常有用。

language

Commands specific to a source language.

`(lldb) language ...` = `(lldb) la ...`

LLDB 有几个指定语言的命令，包括C++，Objective-C，Swift 和 RenderScript。在这篇文章中，我们重点关注 Swift。这是两个命令：`demangle` 和 `refcount`。

`demangle` 正如其名字而言，就是用来重组 Swift 类型名的（因为 Swift 在编译的时候会生成类型名来避免命名空间的问题）。如果你想了解多一点的话，我建议你看 WWDC14 的这个分享会 — [“Advanced Swift Debugging in LLDB”]。

`refcount` 同样也是一个相当直观的命令，能获得指定对象的引用数量。一起来看一下对象输出的示例，我们用了上一节讲到的对象 — `logger`：

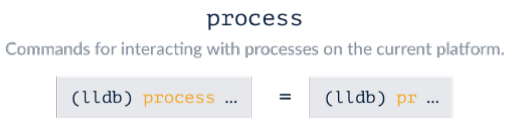
```
(lldb) language swift refcount logger

refcount data: (strong = 4, weak = 0)
```

当然了，在你调试某些内存泄露问题时，这个命令就会很有帮助。

3. 控制应用的执行流程

`process`, `breakpoint`, `thread`

这节是我最喜欢的一节，因为在 LLDB 使用这几个命令（尤其是 `breakpoint` 命令），你可以在调试的时候使很多常规任务变得自动化，这样就能大大加快你的调试工作。



通过 `process` 基本上你就可以控制调试的过程了，还能链接到特定的 target 或者停止调试器。 但是因为 Xcode 已经自动地帮我们做好了这个工作了（Xcode 在任何时候运行一个 target 时都会连接 LLDB）。我不会在这儿讲太多，你可以在这篇 Apple 的指南中阅读一下如何用终端连接到一个 target — [“Using LLDB as a Standalone Debugger”]。

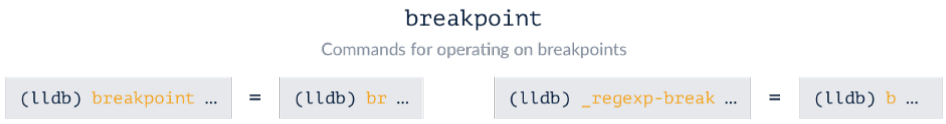
使用 `process status` 的话，你可以知道当前调试器停住的地址：

```
(lldb) process status

Process 27408 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = step over
frame #0: 0x000000010bbe4889 LLDB-Debugger-
Exploration`ViewController.viewDidLoad(self=0x00007fa0c1406900) -> () at
ViewController.swift:69
66
67          let a = 2, b = 2
68          let result = valueOfLifeWithoutSumOf(a, and: b)
-> 69          print(result)
70
71
72
```

想要继续 target 的执行过程直到遇到下次断点的话，运行这个命令：

```
(lldb) process continue

(lldb) c // 或者只键入 "c", 这跟上一条命令是一样的
```

这个命令等同于 Xcode 调试器工具栏上的”continue“按钮：



`breakpoint` 命令允许你用任何可能的方式操作断点。我们跳过最显而易见的命令：`breakpoint enable`，`breakpoint disable` 和 `breakpoint delete`。

首先，查看你所有断点的话可以用如下示例中的 `list` 子命令：

```
(lldb) breakpoint list

Current breakpoints:
1: file = '/Users/Ahmed/Desktop/Recent/LLDB-Debugger-Exploration/LLDB-Debugger-

'/Users/Ahmed/Desktop/Recent/LLDB-Debugger-Exploration/LLDB-Debugger-Exploration/
ViewController.swift', line = 95, exact_match = 0, locations = 1, resolved = 1, hit count = 1

1.1: where = LLDB-Debugger-Exploration`LLDB_Debugger_Exploration.ViewController.valueOfLifeWithou

2: file = '/Users/Ahmed/Desktop/Recent/LLDB-Debugger-Exploration/LLDB-Debugger-

'/Users/Ahmed/Desktop/Recent/LLDB-Debugger-Exploration/LLDB-Debugger-Exploration/
ViewController.swift', line = 60, exact_match = 0, locations = 1, resolved = 1, hit count = 1

2.1: where = LLDB-Debugger-Exploration`LLDB_Debugger_Exploration.ViewController.viewDidLoad () ->
```

列表中的第一个数字是是断点的 ID，你可以通过这个 ID 引用到指定的断点。现在让我们在控制台中设置一些新的断点：

```
(lldb) breakpoint set -f ViewController.swift -l 96

Breakpoint 3: where = LLDB-Debugger-Exploration`LLDB_Debugger_Exploration.ViewController.valueOfL
```

这个例子中的 `-f` 是你想要放置断点处的文件名，`-l` 是新断点的行数。还有一种更简洁的方式设置同样的断点，就是用快捷方式 `b`：

```
(lldb) b ViewController.swift:96
```

同样地，你也可以用指定的正则（比如函数名）来设置断点，使用下面的命令：

```
(lldb) breakpoint set --func-regex valueOfLifeWithoutSumOf

(lldb) b -r valueOfLifeWithoutSumOf // 上一条命令的简化版本
```

有些时候设置断点只命中一次也是有用的，然后指示这个断点立即删除自己，当然啦，有一个命令来处理这件事：

```
(lldb) breakpoint set --one-shot -f ViewController.swift -l 90

(lldb) br s -o -f ViewController.swift -l 91 // 上一条命令的简化版本
```

现在我们来到了最有趣的部分 — 自动化断点。你知道你可以设置一个特定的动作使它在断点停住的时候执行吗？是的，你可以！你是否会在代码中用 `print()` 来在调试的时候得到你感兴趣的值？请不要再这样做了，这里有一种更好的方法。

通过 `breakpoint` 命令，你可以设置好命令，使其在断点命中时可以正确执行。你甚至可以设置”不可见“的断点，这种断点并不会打断运行过程。从技术上讲，这些“不可见的”断点其实是会中断执行的，但如果在命令链的末尾添上“continue”命令的话，你就不会注意到它。

```
(lldb) b ViewController.swift:96 // Let's add a breakpoint first Breakpoint 2:
where = LLDB-Debugger-
Exploration`LLDB_Debugger_Exploration.ViewController.valueOfLifeWithoutSumOf
(Swift.Int, and : Swift.Int) -> Swift.Int + 45 at ViewController.swift:96, address
= 0x000000010c555b4d (lldb) breakpoint command add 2 // 准备某些命令 Enter your
debugger command(s). Type 'DONE' to end. > p sum // 打印变量 "sum" 的值 > p a + b //
运行 a + b > DONE
```

为了确保你添加的命令是正确的，可以使用 `breakpoint command list <breakpoint id>` 子命令：

```
(lldb) breakpoint command list 2

Breakpoint 2:
Breakpoint commands:
p sum
p a + b
```

当下次断点命中时我们就会在控制台看到下面的输出：

```
Process 36612 resuming
p sum
(Int) $R0 = 6

p a + b
(Int) $R1 = 4
```

太棒了！这正是我们想要的。你可以通过在命令链的末尾添加 `continue` 命令让执行过程更加顺畅，这样你就不会停在这个断点。

```
(lldb) breakpoint command add 2 // 准备某些命令

Enter your debugger command(s).  Type 'DONE' to end.
> p sum // 打印变量 "sum" 的值
> p a + b // 运行 a + b
> continue // 第一次命中断点后直接恢复
> DONE
```

结果会是这样：

```
p sum
(Int) $R0 = 6
```

```
p a + b
(Int) $R1 = 4

continue
Process 36863 resuming
Command #3 'continue' continued the target.
```

thread

Commands for operating on one or more threads in the current process.

`(lldb) thread ...` = `(lldb) th ...`

通过 `thread` 命令和它的子命令，你可以完全操控执行流程：`step-over`，`step-in`，`step-out` 和 `continue`。这些命令等同于 Xcode 调试器工具栏上的流程控制按钮。

LLDB 同样也对这些特殊的命令预先定义好了快捷方式：

```
(lldb) thread step-over
(lldb) next // 和 "thread step-over" 命令效果一样
(lldb) n // 和 "next" 命令效果一样

(lldb) thread step-in
(lldb) step // 和 "thread step-in" 命令效果一样
(lldb) s // 和 "step" 命令效果一样
```

为了获取当前线程的更多信息，我们只需要调用 `info` 子命令：

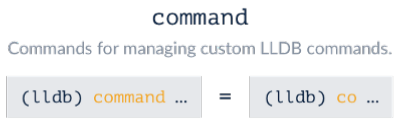
```
(lldb) thread info

thread #1: tid = 0x17de17, 0x0000000109429a90 LLDB-Debugger-
Exploration`ViewController.sumOf(a=2, b=2, self=0x00007fe775507390) -> Int at
ViewController.swift:90, queue = 'com.apple.main-thread', stop reason = step in
```

想要看到当前所有的活动线程的话使用 `list` 子命令：

```
(lldb) thread list

Process 50693 stopped

* thread #1: tid = 0x17de17, 0x0000000109429a90 LLDB-Debugger-
Exploration`ViewController.sumOf(a=2, b=2, self=0x00007fe775507390) -> Int at
ViewController.swift:90, queue = 'com.apple.main-thread', stop reason = step in

  thread #2: tid = 0x17df4a, 0x000000010daa4dc6 libsystem_kernel.dylib`kevent_qos
+ 10, queue = 'com.apple.libdispatch-manager'

  thread #3: tid = 0x17df4b, 0x000000010daa444e
libsystem_kernel.dylib`__workq_kernreturn + 10

  thread #5: tid = 0x17df4e, 0x000000010da9c34a
libsystem_kernel.dylib`mach_msg_trap + 10, name = 'com.apple.uikit.eventfetch-
thread'
```

荣誉奖

`command`, `platform`, `gui`



在 LLDB 中你可以找到一个命令管理其他的命令，听起来很奇怪，但实际上它是非常有用的小工具。首先，它允许你从文件中执行一些 LLDB 命令，这样你就可以创建一个储存着一些实用命令的文件，然后就能立刻允许这些命令，就像是单个命令那样。这是所说的文件的简单例子：

```
thread info // 显示当前线程的信息
br list // 显示所有的断点
```

下面是实际命令的样子：

```
(lldb) command source /Users/Ahmed/Desktop/lldb-test-script

Executing commands in '/Users/Ahmed/Desktop/lldb-test-script'.

thread info
thread #1: tid = 0x17de17, 0x0000000109429a90 LLDB-Debugger-
Exploration`ViewController.sumOf(a=2, b=2, self=0x00007fe775507390) -> Int at
ViewController.swift:90, queue = 'com.apple.main-thread', stop reason = step in

br list
Current breakpoints:
1: file = '/Users/Ahmed/Desktop/Recent/LLDB-Debugger-Exploration/LLDB-Debugger-

'/Users/Ahmed/Desktop/Recent/LLDB-Debugger-Exploration/LLDB-Debugger-Exploration/
ViewController.swift', line = 60, exact_match = 0, locations = 1, resolved = 1, hit count = 0
1.1: where = LLDB-Debugger-Exploration`LLDB_Debugger_Exploration.ViewController.viewDidLoad () ->
```

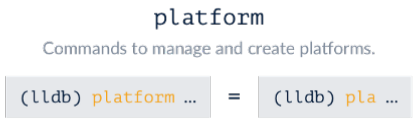
遗憾的是还有一个缺点，你不能传递任何参数给这个源文件（除非你在脚本文件本身中创建一个有效的变量）。

如果你需要更高级的功能，你也可以使用 `script` 子命令，这个命令允许你用自定义的 Python 脚本 管理(`add`, `delete`, `import` 和 `list`)，通过 `script` 命令能实现真正的自动化。请阅读这个优秀的教程 Python scripting for LLDB。为了演示的目的，让我们创建一个脚本文件 script.py，然后写一个简单的命令 **print_hello()**，这个命令会在控制台中打印出“Hello Debugger!“：

```
import lldb

def print_hello(debugger, command, result, internal_dict):
        print "Hello Debugger!"

def __lldb_init_module(debugger, internal_dict):
        debugger.HandleCommand('command script add -f script.print_hello
print_hello') // 控制脚本的初始化同时从这个模块中添加命令
        print 'The "print_hello" python command has been installed and is ready
for use.' // 打印确认一切正常
```

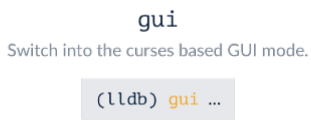
接下来我们需要导入一个 Python 模块，就能开始正常地使用我们的脚本命令了：

```
(lldb) command import ~/Desktop/script.py

The "print_hello" python command has been installed and is ready for use.

(lldb) print_hello

Hello Debugger!
```



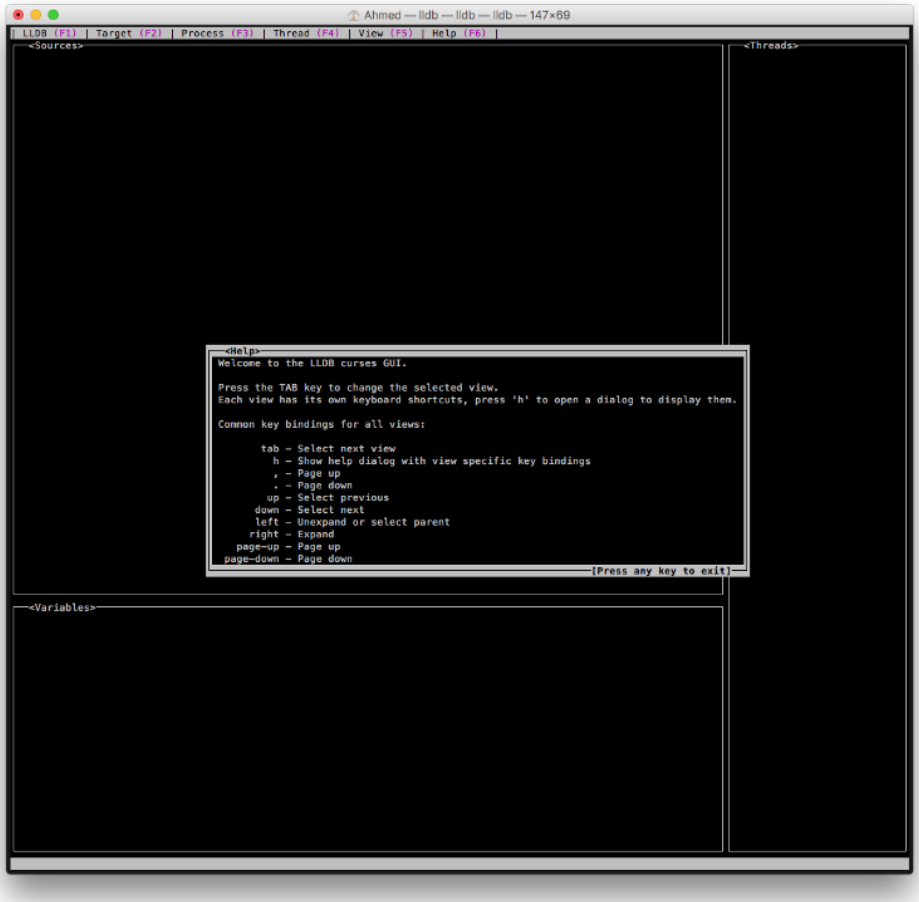
你可以使用 `status` 子命令来快速检查当前的环境信息，`status` 会告诉你：SDK 路径、处理器的架构、操作系统版本甚至是该 SDK 可支持的设备的列表。

```
(lldb) platform status

Platform: ios-simulator
Triple: x86_64-apple-macosx
OS Version: 10.12.5 (16F73)
Kernel: Darwin Kernel Version 16.6.0: Fri Apr 14 16:21:16 PDT 2017; root:xnu-3789.60.24~6/RELEASE
Hostname: 127.0.0.1
WorkingDir: /
SDK Path: "/Applications/Xcode.app/Contents/Developer/Platforms/
iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator.sdk"

Available devices:
614F8701-3D93-4B43-AE86-46A42FEB905A: iPhone 4s
CD516CF7-2AE7-4127-92DF-F536FE56BA22: iPhone 5
0D76F30F-2332-4E0C-9F00-B86F009D59A3: iPhone 5s
3084003F-7626-462A-825B-193E6E5B9AA7: iPhone 6
...
```



你不能在 Xcode 中使用 LLDB GUI 模式，但你总是可以从终端使用（LLDB GUI 模式）。

```
(lldb) gui

// 如果你试着在 Xcode 中执行这个 gui 命令的话，你将会看到这个错误：the gui command requires an interactive
```



这就是 LLDB GUI 模式看起来的样子。

结论：

在这篇文章中，我只是浅析了 LLDB 的皮毛知识而已，即使 LLDB 已经有好些年头了，但是仍然有许多人并没有完全发挥出它的潜能。我只是对基本的方法做了一个概述，以及谈了 LLDB 如何自动化调试步骤。我希望这会是有帮助的。

还有很多 LLDB 的方法并没有写到，然后还有一些视图调试技术我没有提及。如果你对这些话题感兴趣的话，请在下面留下你的评论，我会更加乐于写这些话题。

我强烈建议你打开终端，启动 LLDB，只需要敲入 `help`，就会向你展示完整的文档。你可以花费数小时去阅读，但是我保证这将是一个合理的时间投资。因为了解你的工具是工程师真正产出的唯一途径。

- LLDB 官方网站 — 你会在这里找到所有与 LLDB 相关的材料。文档、指南、教程、源文件以及更多。
- LLDB Quick Start Guide by Apple — 同样地，Apple 提供了很好的文档。这篇指南能帮你快速上手 LLDB，当然，他们也叙述了怎样不通过 Xcode 地用 LLDB 调试。
- How debuggers work: Part 1—Basics — 我非常喜欢这个系列的文章，这是对调试器实际工作方式很好的概述。文章介绍了用 C 语言手工编写的调试器代码要遵循的所有基本原理。我强烈建议你去阅读这个优秀系列的所有部分（第2部分, 第3部分）。
- WWDC14 Advanced Swift Debugging in LLDB — 关于在 LLDB 中用 Swift 调试的一篇不错的概述，也讲了 LLDB 如何通过内建的方法和特性实现完整的调试操作，来帮你变得更加高效。
- Introduction To LLDB Python Scripting — 这篇介绍 LLDB Python 脚本的指南能让你快速上手。

- Dancing in the Debugger. A Waltz with LLDB — 对 LLDB 一些基础知识的介绍，有些知识有点过时了（比如说 `(lldb) thread return` 命令）。遗憾的是，它不能直接用于 Swift，因为它会对引用计数带了一些潜在的隐患。但是，这仍然是你开始 LLDB 之旅不错的文章。