与调试器共舞 - LLDB 的华尔兹

nangege (https://github.com/nangege) 2014/12/19

你是否曾经苦恼于理解你的代码，而去尝试打印一个变量的值?

```
NSLog(@"%@", whatIsInsideThisThing);
```

或者跳过一个函数调用来简化程序的行为?

```
NSNumber *n = @7; // 实际应该调用这个函数: Foo();
```

或者短路一个逻辑检查?

```
if (1 || theBooleanAtStake) { ... }
```

或者伪造一个函数实现?

```
int calculateTheTrickyValue {
  return 9;

  /*
   先这么着
   ...
}
```

并且每次必须重新编译，从头开始?

构建软件是复杂的，并且 Bug 总会出现。一个常见的修复周期就是修改代码，编译，重新运行，并且祈祷出现最好的结果。

但是不一定要这么做。你可以使用调试器。而且即使你已经知道如何使用调试器检查变量，它可以做的还有很多。

这篇文章将试图挑战你对调试的认知，并详细地解释一些你可能还不了解的基本原理，然后展示一系列有趣的例子。现在就让我们开始与调试器共舞一曲华尔兹，看看最后能达到怎样的高度。

LLDB

LLDB (http://lldb.llvm.org/) 是一个有着 REPL 的特性和 C++ ,Python 插件的开源(http://lldb.llvm.org/source.html)调试器。LLDB 绑定在 Xcode 内部，存在于主窗口底部的控制台中。调试器允许你在程序运行的特定时暂停它，你可以查看变量的值，执行自定的指令，并且按照你所认为合适的步骤来操作程序的进展。(这里(http://eli.thegreenplace.net/2011/01/23/how-debuggers-work-part-1.html)有一个关于调试器如何工作的总体的解释。)

你以前有可能已经使用过调试器，即使只是在 Xcode 的界面上加一些断点。但是通过一些小的技巧，你就可以做一些非常酷的事情。GDB to LLDB (http://lldb.llvm.org/lldb-gdb.html) 参考是一个非常好的调试器可用命令的总览。你也可以安装 Chisel (https://github.com/facebook/chisel)，它是一个开源的 LLDB 插件合辑，这会使调试变得更加有趣。

与此同时，让我们以在调试器中打印变量来开始我们的旅程吧。

基础

这里有一个简单的小程序，它会打印一个字符串。注意断点已经被加在第 8 行。断点可以通过点击 Xcode 的源码窗口的侧边槽进行创建。

```
1  #import <Foundation/Foundation.h>
2
3  int main() {
4    @autoreleasepool {
5      NSUInteger count = 99;
6      NSString *objects = @"red balloons";
7
8      NSLog(@"%lu %@.", count, objects);
9    }
10
11   return 0;
12 }
```

程序会在这一行停止运行，并且控制台会被打开，允许我们和调试器交互。那我们应该打些什么呢?

help

最简单命令是 `help`，它会列举出所有的命令。如果你忘记了一个命令是做什么的，或者想知道更多的话，你可以通过 `help <command>` 来了解更多细节，例如 `help print` 或者 `help thread`。如果你甚至忘记了 `help` 命令是做什么的，你可以试试 `help help`。不过你如果知道这么做，那就说明你大概还没有忘光这个命令。😛

print

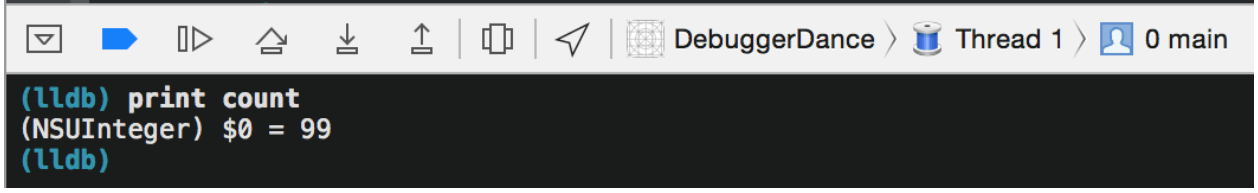
打印值很简单；只要试试 `print` 命令:

```
#import <Foundation/Foundation.h>

int main() {
  @autoreleasepool {
    NSUInteger count = 99;
    NSString *objects = @"red balloons";

    NSLog(@"%lu %@.", count, objects);
  }

```



LLDB 实际上会作前缀匹配。所以你也可以使用 `prin`， `pri`，或者 `p`。但你不能使用 `pr`，因为 LLDB 不能消除和 `process` 的歧义 (幸运的是 `p` 并没有歧义)。

你可能还注意到了，结果中有个 `$0`。实际上你可以使用它来指向这个结果。试试 `print $0 + 7`，你会看到 `106`。任何以美元符开头的东西都是存在于 LLDB 的命名空间的，它们是为了帮助你进行调试而存在的。

expression

如果想改变一个值怎么办？你或许会猜 *modify*。其实这时候我们要用到的是 `expression` 这个方便的命令。

```
(lldb) expression count = 42
(NSUInteger) $4 = 42
(lldb) print count
(NSUInteger) $5 = 42
(lldb)
```

这不仅会改变调试器中的值，实际上它改变了程序中的值。这时候继续执行程序，将会打印 `42 red balloons` 。神奇吧。

注意，从现在开始，我们将会偷懒分别以 `p` 和 `e` 来代替 `print` 和 `expression` 。

什么是 *print* 命令

考虑一个有意思的表达式： `p count = 18` 。如果我们运行这条命令，然后打印 `count` 的内容。我们将看到它的结果与 `expression count = 18` 一样。

和 `expression` 不同的是， `print` 命令不需要参数。比如 `e -h +17` 中，你很难区分到底是以 `-h` 为标识，仅仅执行 `+17` 呢，还是要计算 `17` 和 `h` 的差值。连字符号确实很让人困惑，你或许得不到自己想要的结果。

幸运的是，解决方案很简单。用 `--` 来表征标识的结束，以及输入的开始。如果想要 `-h` 作为标识，就用 `e -h -- +17` ，如果想计算它们的差值，就使用 `e -- -h +17` 。因为一般来说不使用标识的情况比较多，所以 `e --` 就有了一个简写的方式，那就是 `print` 。

输入 `help print` ，然后向下滚动，你会发现：

```
'print' is an abbreviation for 'expression --'.
(print是 `expression --` 的缩写)
```

打印对象

尝试输入

```
p objects
```

输出会有点啰嗦

```
(NSString *) $7 = 0x0000000104da4040 @"red balloons"
```

如果我们尝试打印结构更复杂的对象，结果甚至会更糟

```
(lldb) p @[ @"foo", @"bar" ]

(NSArray *) $8 = 0x00007fdb9b71b3e0 @"2 objects"
```

实际上，我们想看的是对象的 `description` 方法的结果。我么需要使用 `-O` (字母 O，而不是数字 0) 标志告诉 `expression` 命令以 `对象` (Object) 的方式来打印结果。

```
(lldb) e -O -- $8
<__NSArrayI 0x7fdb9b71b3e0>(
foo,
bar
)
```

幸运的是， `e -o --` 有也有个别名，那就是 `po` (**p**rint **o**bject 的缩写)，我们可以使用它来进行简化：

```
(lldb) po $8
<__NSArrayI 0x7fdb9b71b3e0>(
foo,
bar
)
(lldb) po @"lunar"
lunar
(lldb) p @"lunar"
(NSString *) $13 = 0x00007fdb9d0003b0 @"lunar"
```

打印变量

可以给 `print` 指定不同的打印格式。它们都是以 `print/<fmt>` 或者简化的 `p/<fmt>` 格式书写。下面是一些例子：

默认的格式

```
(lldb) p 16
16
```

十六进制:

```
(lldb) p/x 16
0x10
```

二进制 (`t` 代表 **t**wo):

```
(lldb) p/t 16
0b00000000000000000000000000010000
(lldb) p/t (char)16
0b00010000
```

你也可以使用 `p/c` 打印字符，或者 `p/s` 打印以空终止的字符串 (译者注：以 '\0' 结尾的字符串)。
这里 (https://sourceware.org/gdb/onlinedocs/gdb/Output-Formats.html)是格式的完整清单。

变量

现在你已经可以打印对象和简单类型，并且知道如何使用 `expression` 命令在调试器中修改它们了。现在让我们使用一些变量来减少输入量。就像你可以在 C 语言中用 `int a = 0` 来声明一个变量一样，你也可以在 LLDB 中做同样的事情。不过为了能使用声明的变量，变量**必须**以美元符开头。

```
(lldb) e int $a = 2
(lldb) p $a * 19
38
(lldb) e NSArray *$array = @[ @"Saturday", @"Sunday", @"Monday" ]
(lldb) p [$array count]
2
(lldb) po [[$array objectAtIndex:0] uppercaseString]
SATURDAY
(lldb) p [[$array objectAtIndex:$a] characterAtIndex:0]
error: no known method '-characterAtIndex:'; cast the message send to the method's
return type
error: 1 errors parsing expression
```

悲剧了，LLDB 无法确定涉及的类型 (译者注：返回的类型)。这种事情常常发生，给个说明就好了：

```
(lldb) p (char)[[$array objectAtIndex:$a] characterAtIndex:0]
'M'
(lldb) p/d (char)[[$array objectAtIndex:$a] characterAtIndex:0]
77
```

变量使调试器变的容易使用得多，想不到吧？😉

流程控制

当你通过 Xcode 的源码编辑器的侧边槽 (或者通过下面的方法) 插入一个断点，程序到达断点时会就会停止运行。

调试条上会出现四个你可以用来控制程序的执行流程的按钮。

从左到右，四个按钮分别是：continue，step over，step into，step out。

第一个，continue 按钮，会取消程序的暂停，允许程序正常执行 (要么一直执行下去，要么到达下一个断点)。在 LLDB 中，你可以使用 `process continue` 命令来达到同样的效果，它的别名为 `continue`，或者也可以缩写为 `c`。

第二个，step over 按钮，会以黑盒的方式执行一行代码。如果所在这行代码是一个函数调用，那么就**不会**跳进这个函数，而是会执行这个函数，然后继续。LLDB 则可以使用 `thread step-over`， `next`，或者 `n` 命令。

如果你确实想跳进一个函数调用来调试或者检查程序的执行情况，那就用第三个按钮，step in，或者在LLDB中使用 `thread step in`， `step`，或者 `s` 命令。注意，当前行不是函数调用时， `next` 和 `step` 效果是一样的。

大多数人知道 `c`， `n` 和 `s`，但是其实还有第四个按钮，step out。如果你曾经不小心跳进一个函数，但实际上你想跳过它，常见的反应是重复的运行 `n` 直到函数返回。其实这种情况，step out 按钮是你的救世主。它会继续执行到下一个返回语句 (直到一个堆栈帧结束) 然后再次停止。

例子

考虑下面一段程序：

```
1  #import <Foundation/Foundation.h>
2
3  static BOOL isEven(int i) {
4    if (i %2 == 0) {
5      NSLog(@"%d is even!", i);
6      return YES;
7    }
8
9    NSLog(@"%d is odd!", i);
10   return NO;
11 }
12
13 int main() {
14   @autoreleasepool {
15     int i = 99;
16     BOOL even0 = isEven(i + 2);
17     BOOL even1 = isEven(i + 11);
18
```

假如我们运行程序，让它停止在断点，然后执行下面一些列命令：

```
p i
n
s
p i
finish
p i
frame info
```

这里， `frame info` 会告诉你当前的行数和源码文件，以及其他一些信息；查看 `help frame`， `help thread` 和 `help process` 来获得更多信息。这一串命令的结果会是什么？看答案之前请先想一想。

```
(lldb) p i
(int) $0 = 99
(lldb) n
2014-11-22 10:49:26.445 DebuggerDance[60182:4832768] 101 is odd!
(lldb) s
(lldb) p i
(int) $2 = 110
(lldb) finish
2014-11-22 10:49:35.978 DebuggerDance[60182:4832768] 110 is even!
(lldb) p i
(int) $4 = 99
(lldb) frame info
frame #0: 0x000000010a53bcd4 DebuggerDance`main + 68 at main.m:17
```

它始终在 17 行的原因是 `finish` 命令一直运行到 `isEven()` 函数的 `return`，然后立刻停止。注意即使它还在 17 行，其实这行已经被执行过了。

Thread Return

调试时，还有一个很棒的函数可以用来控制程序流程： `thread return` 。它有一个可选参数，在执行时它会把可选参数加载进返回寄存器里，然后立刻执行返回命令，跳出当前栈帧。这意味这函数剩余的部分**不会被执行**。这会给 ARC 的引用计数造成一些问题，或者会使函数内的清理部分失效。但是在函数的开头执行这个命令，是个非常好的隔离这个函数，伪造返回值的方式 。

让我们稍微修改一下上面代码段并运行：

```
p i
s
thread return NO
n
p even0
frame info
```

看答案前思考一下。下面是答案：

```
(lldb) p i
(int) $0 = 99
(lldb) s
(lldb) thread return NO
(lldb) n
(lldb) p even0
(BOOL) $2 = NO
(lldb) frame info
frame #0: 0x00000001009a5cc4 DebuggerDance`main + 52 at main.m:17
```

断点

我们都把断点作为一个停止程序运行，检查当前状态，追踪 bug 的方式。但是如果我们改变和断点交互的方式，很多事情都变成可能。

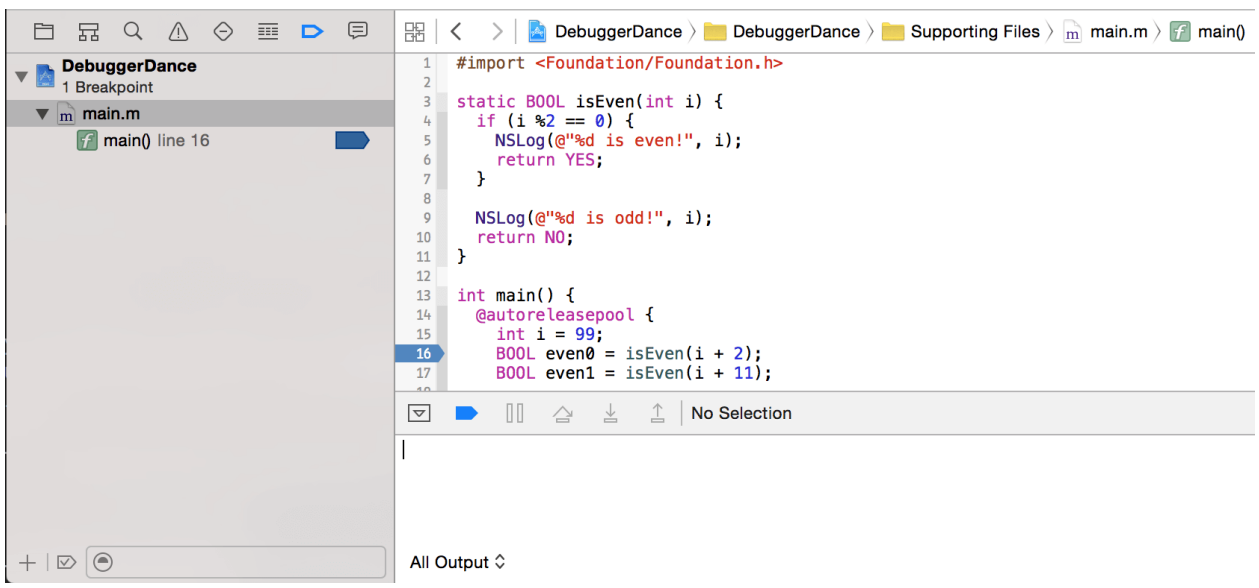
> 断点允许控制程序什么时候停止，然后允许命令的运行。

想象把断点放在函数的开头，然后用 `thread return` 命令重写函数的行为，然后继续。想象一下让这个过程自动化，听起来不错，不是吗?

管理断点

Xcode 提供了一系列工具来创建和管理断点。我们会一个个看过来并介绍 LLDB 中等价的命令 (是的，你可以在调试器**内部**添加断点)。

在 Xcode 的左侧面板，有一组按钮。其中一个看起来像断点。点击它打开断点导航，这是一个可以快速管理所有断点的面板。



在这里你可以看到所有的断点 - 在 LLDB 中通过 `breakpoint list` (或者 `br li`) 命令也做同样的事儿。你也可以点击单个断点来开启或关闭 - 在 LLDB 中使用 `breakpoint enable <breakpointID>` 和 `breakpoint disable <breakpointID>`：

```
(lldb) br li
Current breakpoints:
1: file = '/Users/arig/Desktop/DebuggerDance/DebuggerDance/main.m', line = 16, loc
ations = 1, resolved = 1, hit count = 1

  1.1: where = DebuggerDance`main + 27 at main.m:16, address = 0x000000010a3f6cab,
resolved, hit count = 1

(lldb) br dis 1
1 breakpoints disabled.
(lldb) br li
Current breakpoints:
1: file = '/Users/arig/Desktop/DebuggerDance/DebuggerDance/main.m', line = 16, loc
ations = 1 Options: disabled

  1.1: where = DebuggerDance`main + 27 at main.m:16, address = 0x000000010a3f6cab,
unresolved, hit count = 1

(lldb) br del 1
1 breakpoints deleted; 0 breakpoint locations disabled.
(lldb) br li
No breakpoints currently set.
```

创建断点

在上面的例子中，我们通过在源码页面器的滚槽 16 上点击来创建断点。你可以通过把断点拖拽出滚槽，然后释放鼠标来删除断点 (消失时会有一个非常可爱的噗的一下的动画)。你也可以在断点导航页选择断点，然后按下删除键删除。

要在调试器中创建断点，可以使用 `breakpoint set` 命令。

```
(lldb) breakpoint set -f main.m -l 16
Breakpoint 1: where = DebuggerDance`main + 27 at main.m:16, address = 0x000000010a
3f6cab
```

也可以使用缩写形式 `br` 。虽然 `b` 是一个完全不同的命令 (`_regexp-break` 的缩写)，但恰好也可以实现和上面同样的效果。

```
(lldb) b main.m:17
Breakpoint 2: where = DebuggerDance`main + 52 at main.m:17, address = 0x000000010a
3f6cc4
```

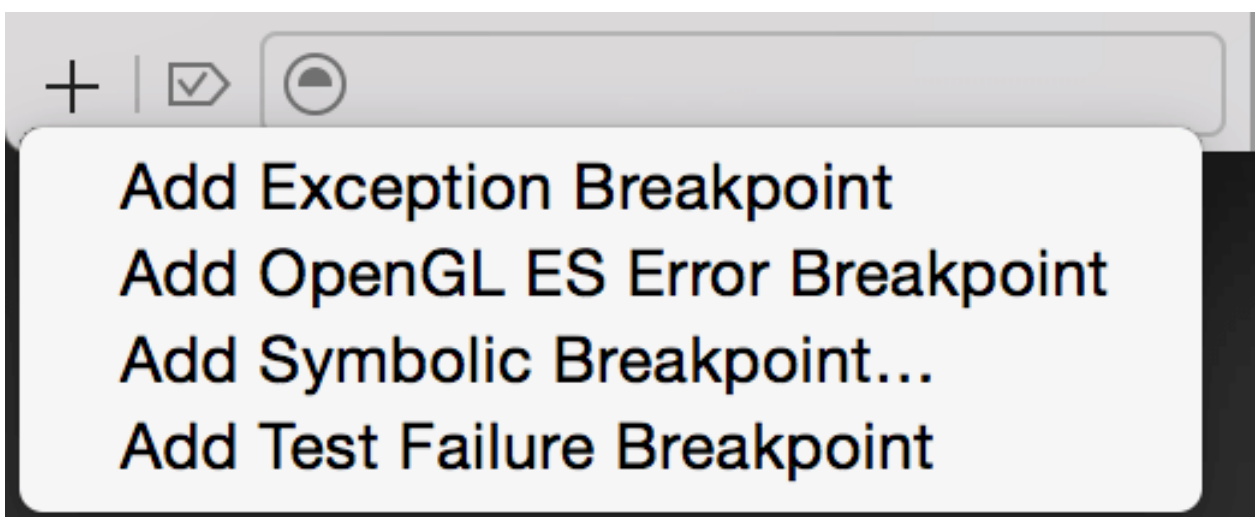
也可以在一个符号 (C 语言函数) 上创建断点，而完全不用指定哪一行

```
(lldb) b isEven
Breakpoint 3: where = DebuggerDance`isEven + 16 at main.m:4, address = 0x000000010
a3f6d00
(lldb) br s -F isEven
Breakpoint 4: where = DebuggerDance`isEven + 16 at main.m:4, address = 0x000000010
a3f6d00
```

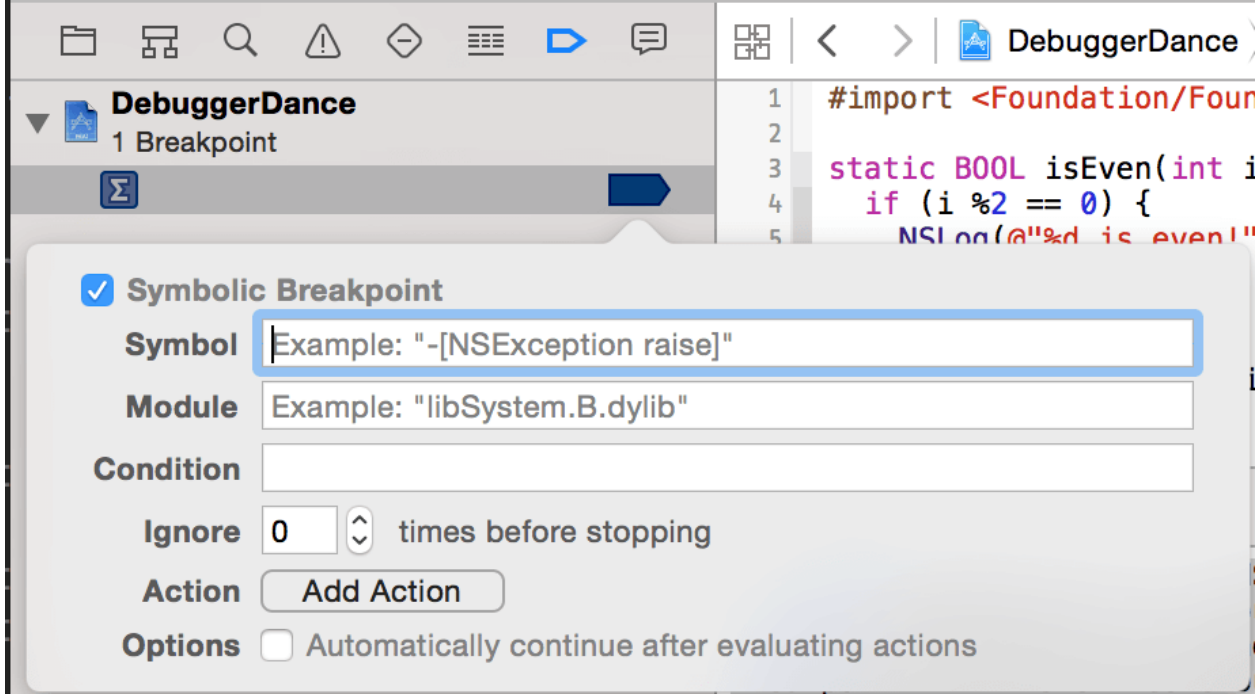
这些断点会准确的停止在函数的开始。Objective-C 的方法也完全可以：

```
(lldb) breakpoint set -F "-[NSArray objectAtIndex:]"
Breakpoint 5: where = CoreFoundation`-[NSArray objectAtIndex:], address = 0x000000
010ac7a950
(lldb) b -[NSArray objectAtIndex:]
Breakpoint 6: where = CoreFoundation`-[NSArray objectAtIndex:], address = 0x000000
010ac7a950
(lldb) breakpoint set -F "+[NSSet setWithObject:]"
Breakpoint 7: where = CoreFoundation`+[NSSet setWithObject:], address = 0x00000001
0abd3820
(lldb) b +[NSSet setWithObject:]
Breakpoint 8: where = CoreFoundation`+[NSSet setWithObject:], address = 0x00000001
0abd3820
```

如果想在 Xcode 的UI上创建符号断点，你可以点击断点栏左侧的 + 按钮。

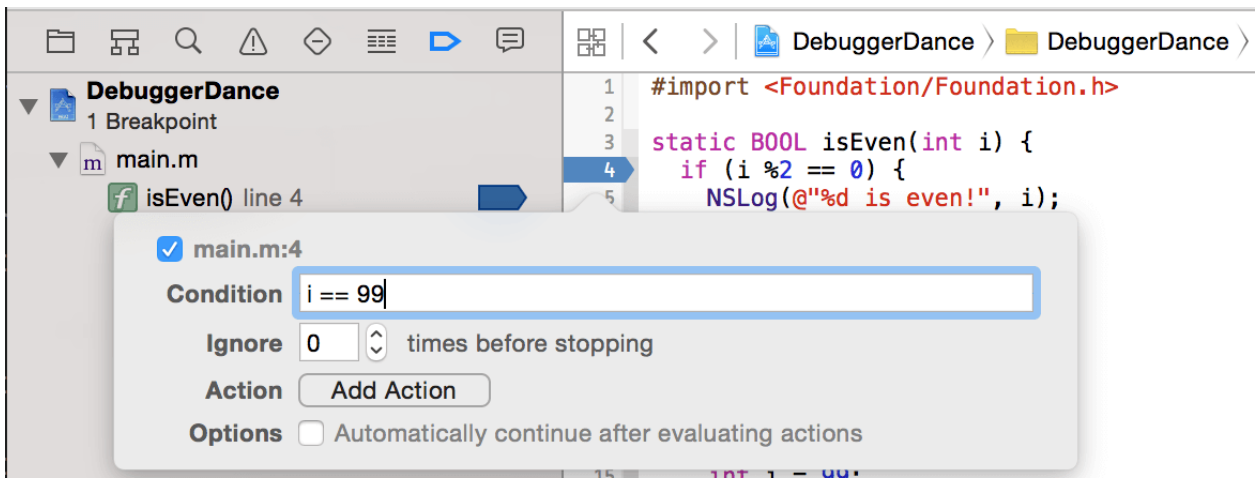


然后选择第三个选项：



这时会出现一个弹出框，你可以在里面添加例如 `-[NSArray objectAtIndex:]` 这样的符号断点。这样**每次**调用这个函数的时候，程序都会停止，不管是你调用还是苹果调用。

如果你 Xcode 的 UI 上右击**任意**断点，然后选择 "Edit Breakpoint" 的话，会有一些非常诱人的选择。

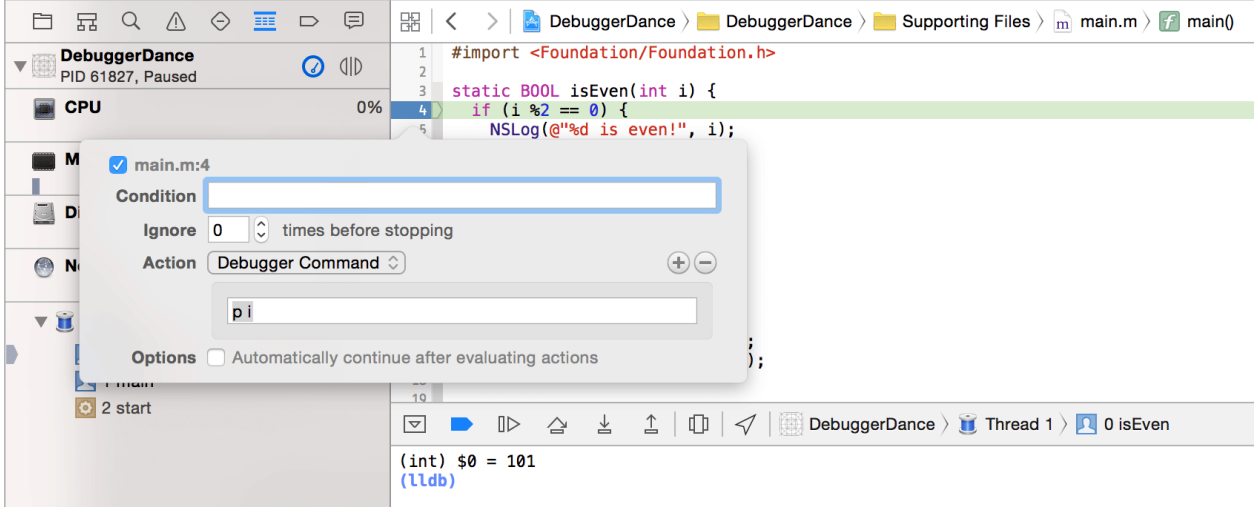


这里，断点已经被修改为**只有**当 `i` 是 `99` 的时候才会停止。你也可以使用 "ignore" 选项来告诉断点最初的 `n` 次调用 (并且条件为真的时候) 的时候不要停止。

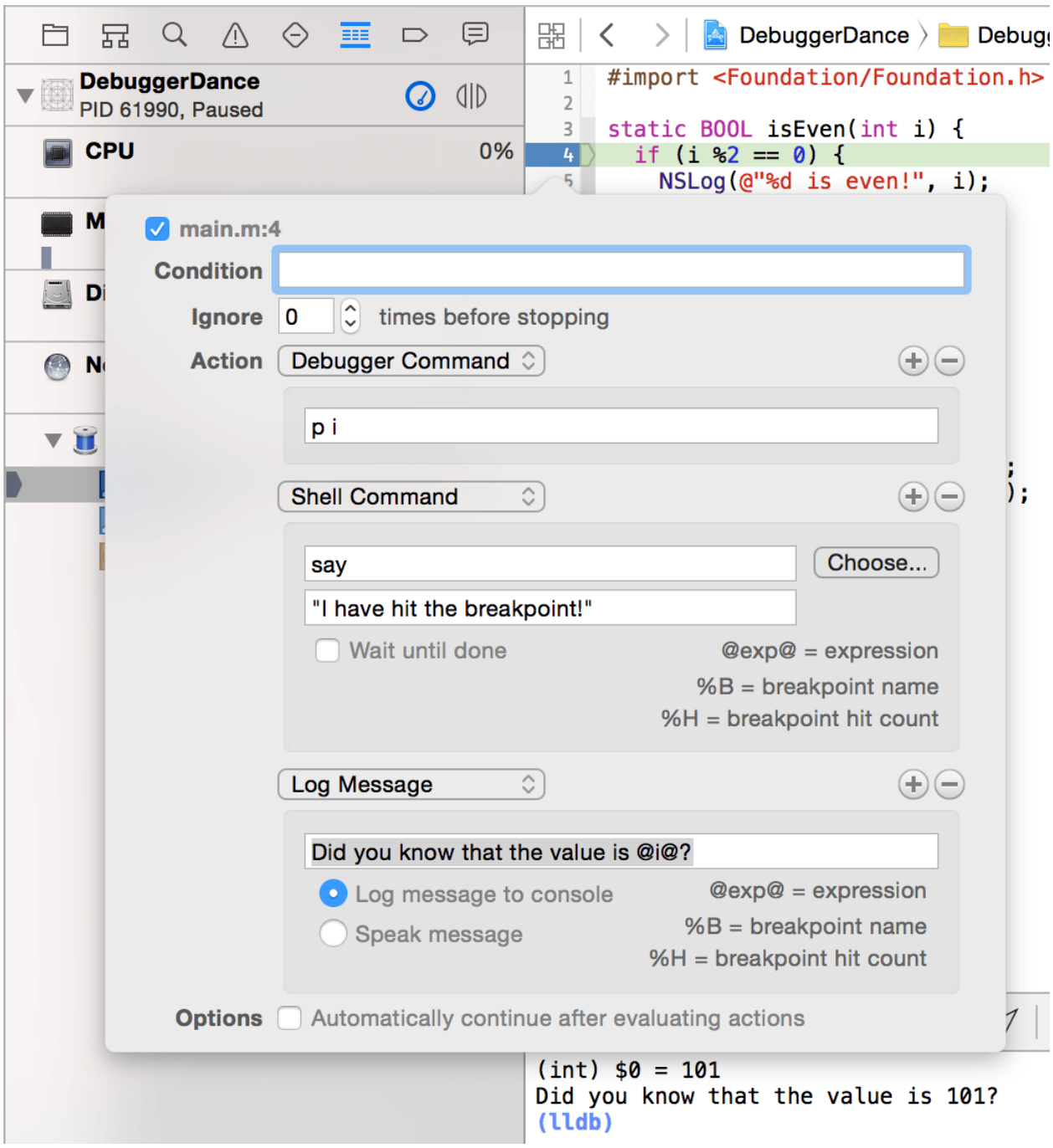
接下来介绍 'Add Action' 按钮...

断点行为 (Action)

上面的例子中，你或许想知道每一次到达断点的时候 `i` 的值。我们可以使用 `p i` 作为断点行为。这样每次到达断点的时候，都会自动运行这个命令。



你也可以添加多个行为，可以是调试器命令，shell 命令，也可以是更直接的打印：



可以看到它打印 `i`，然后大声念出那个句子，接着打印了自定义的表达式。

下面是在 LLDB 而不是 Xcode 的 UI 中做这些的时候，看起来的样子。

```
(lldb) breakpoint set -F isEven
Breakpoint 1: where = DebuggerDance`isEven + 16 at main.m:4, address = 0x000000010
83b5d00
(lldb) breakpoint modify -c 'i == 99' 1
(lldb) breakpoint command add 1
Enter your debugger command(s).  Type 'DONE' to end.
> p i
> DONE
(lldb) br li 1
1: name = 'isEven', locations = 1, resolved = 1, hit count = 0
    Breakpoint commands:
      p i

Condition: i == 99

  1.1: where = DebuggerDance`isEven + 16 at main.m:4, address = 0x00000001083b5d00
, resolved, hit count = 0
```

接下来说说自动化。

赋值后继续运行

看编辑断点弹出窗口的底部，你还会看到一个选项： "*Automatically continue after evaluation actions.*" 。它仅仅是一个选择框，但是却很强大。选中它，调试器会运行你所有的命令，然后继续运行。看起来就像没有执行任何断点一样 (除非断点太多，运行需要一段时间，拖慢了你的程序)。

这个选项框的效果和让最后断点的最后一个行为是 `continue` 一样。选框只是让这个操作变得更简单。调试器的输出是：

```
(lldb) breakpoint set -F isEven
Breakpoint 1: where = DebuggerDance`isEven + 16 at main.m:4, address = 0x000000010
83b5d00
(lldb) breakpoint command add 1
Enter your debugger command(s).  Type 'DONE' to end.
> continue
> DONE
(lldb) br li 1
1: name = 'isEven', locations = 1, resolved = 1, hit count = 0
    Breakpoint commands:
      continue

  1.1: where = DebuggerDance`isEven + 16 at main.m:4, address = 0x00000001083b5d00
, resolved, hit count = 0
```

执行断点后自动继续运行，允许你完全通过断点来修改程序！你可以在某一行停止，运行一个 `expression` 命令来改变变量，然后继续运行。

例子

想想所谓的"打印调试"技术吧，不要这么做：

```
NSLog(@"%@", whatIsInsideThisThing);
```

而是用个打印变量的断点替换 log 语句，然后继续运行。

也不要：

```
int calculateTheTrickyValue {
  return 9;

  /*
   Figure this out later.
   ...
}
```

而是加一个使用 `thread return 9` 命令的断点，然后让它继续运行。

符号断点加上 action 真的很强大。你也可以在你朋友的 Xcode 工程上添加一些断点，并且加上大声朗读某些东西的 action。看看他们要花多久才能弄明白发生了什么。😄

完全在调试器内运行

在开始舞蹈之前，还有一件事要看一看。实际上你可以在调试器中执行任何 C/Objective-C/C++/Swift 的命令。唯一的缺点就是不能创建新函数... 这意味着不能创建新的类，block，函数，有虚拟函数的 C++ 类等等。除此之外，它都可以做。

我们可以申请分配一些字节：

```
(lldb) e char *$str = (char *)malloc(8)
(lldb) e (void)strcpy($str, "munkeys")
(lldb) e $str[1] = 'o'
(char) $0 = 'o'
(lldb) p $str
(char *) $str = 0x00007fd04a900040 "monkeys"
```

我们可以查看内存 (使用 `x` 命令)，来看看新数组中的四个字节：

```
(lldb) x/4c $str
0x7fd04a900040: monk
```

我们也可以去掉 3 个字节 (`x` 命令需要斜引号，因为它只有一个内存地址的参数，而不是表达式；使用 `help x` 来获得更多信息)：

```
(lldb) x/1w `$str + 3`
0x7fd04a900043: keys
```

做完了之后，一定不要忘了释放内存，这样才不会内存泄露。(哈，虽然这是调试器用到的内存)：

```
(lldb) e (void)free($str)
```

让我们起舞

现在我们已经知道基本的步调了，是时候开始跳舞并玩一些疯狂的事情了。我曾经写过一篇 `NSArray` 深度探究 (http://arigrant.com/blog/2014/1/19/adventures-in-the-land-of-nsarray)的博客。这篇博客用了很多 `NSLog` 语句，但实际上我的所有探索都是在调试器中完成的。看看你能不能弄明白怎么做的，这会是一个有意思的练习。

不用断点调试

程序运行时，Xcode 的调试条上会出现暂停按钮，而不是继续按钮：

点击按钮会暂停 app (这会运行 `process interrupt` 命令，因为 LLDB 总是在背后运行)。这会让你可以访问调试器，但看起来可以做的事情不多，因为在当前作用域没有变量，也没有特定的代码让你看。

这就是有意思的地方。如果你正在运行 iOS app，你可以试试这个： (因为全局变量是可访问的)

```
    (lldb) po [[[UIApplication sharedApplication] keyWindow] recursiveDescription]
<UIWindow: 0x7f82b1fa8140; frame = (0 0; 320 568); gestureRecognizers = <NSArray:
0x7f82b1fa92d0>; layer = <UIWindowLayer: 0x7f82b1fa8400>>
   | <UIView: 0x7f82b1d01fd0; frame = (0 0; 320 568); autoresize = W+H; layer = <C
ALayer: 0x7f82b1e2e0a0>>
```

你可以看到整个层次。Chisel (https://github.com/facebook/chisel) 中 `pviews` 就是这么实现的。

更新UI

有了上面的输出，我们可以获取这个 view：

```
(lldb) e id $myView = (id)0x7f82b1d01fd0
```

然后在调试器中改变它的背景色：

```
(lldb) e (void)[$myView setBackgroundColor:[UIColor blueColor]]
```

但是只有程序继续运行之后才会看到界面的变化。因为改变的内容必须被发送到渲染服务中，然后显示才会被更新。

渲染服务实际上是一个另外的进程 (被称作 `backboardd`)。这就是说即使我们正在调试的内容所在的进程被打断了， `backboardd` 也还是继续运行着的。

这意味着你可以运行下面的命令，而不用继续运行程序：

```
(lldb) e (void)[CATransaction flush]
```

即使你仍然在调试器中，UI 也会在模拟器或者真机上实时更新。Chisel (https://github.com/facebook/chisel) 为此提供了一个别名叫做 `caflush`，这个命令被用来实现其他的快捷命令，例如 `hide <view>`， `show <view>` 以及其他很多命令。所有

Chisel (https://github.com/facebook/chisel) 的命令都有文档，所以安装后随意运行 `help` `show` 来看更多信息。

Push 一个 View Controller

想象一个以 `UINavigationController` 为 root ViewController 的应用。你可以通过下面的命令，轻松地获取它：

```
(lldb) e id $nvc = [[[UIApplication sharedApplication] keyWindow] rootViewControll
er]
```

然后 push 一个 child view controller:

```
(lldb) e id $vc = [UIViewController new]
(lldb) e (void)[[$vc view] setBackgroundColor:[UIColor yellowColor]]
(lldb) e (void)[$vc setTitle:@"Yay!"]
(lldb) e (void)[$nvc pushViewContoller:$vc animated:YES]
```

最后运行下面的命令：

```
(lldb) caflush // e (void)[CATransaction flush]
```

navigation Controller 就会立刻就被 push 到你眼前。

查找按钮的 target

想象你在调试器中有一个 `$myButton` 的变量，可以是创建出来的，也可以是从 UI 上抓取出来的，或者是你停止在断点时的一个局部变量。你想知道，按钮按下的时候谁会接收到按钮发出的 action。非常简单：

```
(lldb) po [$myButton allTargets]
{(
    <MagicEventListener: 0x7fb58bd2e240>
)}
(lldb) po [$myButton actionsForTarget:(id)0x7fb58bd2e240 forControlEvent:0]
<__NSArrayM 0x7fb58bd2aa40>(
_handleTap:
)
```

现在你或许想在它发生的时候加一个断点。在 `-[MagicEventListener _handleTap:]` 设置一个符号断点就可以了，在 Xcode 和 LLDB 中都可以，然后你就可以点击按钮并停在你所希望的地方了。

观察实例变量的变化

假设你有一个 `UIView`，不知道为什么它的 `_layer` 实例变量被重写了 (糟糕)。因为有可能并不涉及到方法，我们不能使用符号断点。相反的，我们想**监视**什么时候这个地址被写入。

首先，我们需要找到 `_layer` 这个变量在对象上的相对位置：

```
(lldb) p (ptrdiff_t)ivar_getOffset((struct Ivar *)class_getInstanceVariable([MyVie
w class], "_layer"))
(ptrdiff_t) $0 = 8
```

现在我们知道 (`$myView + 8`) 是被写入的内存地址：

```
(lldb) watchpoint set expression -- (int *)$myView + 8
Watchpoint created: Watchpoint 3: addr = 0x7fa554231340 size = 8 state = enabled t
ype = w
    new value: 0x0000000000000000
```

这被以 `wivar $myView _layer` 加入到 Chisel (https://github.com/facebook/chisel) 中。

非重写方法的符号断点

假设你想知道 `-[MyViewController viewDidAppear:]` 什么时候被调用。如果这个方法并没有在 `MyViewController` 中实现，而是在其父类中实现的，该怎么办呢？试着设置一个断点，会出现以下结果：

```
(lldb) b -[MyViewController viewDidAppear:]
Breakpoint 1: no locations (pending).
WARNING:  Unable to resolve breakpoint to any actual locations.
```

因为 LLDB 会查找一个**符号**，但是实际在这个类上却找不到，所以断点也永远不会触发。你需要做的是为断点设置一个条件 `[self isKindOfClass:[MyViewController class]]`，然后把断点放在 `UIViewController` 上。正常情况下这样设置一个条件可以正常工作。但是这里不会，因为我们没有父类的实现。

`viewDidAppear:` 是苹果实现的方法，因此没有它的符号；在方法内没有 `self` 。如果想在符号断点上使用 `self`，你必须知道它在哪里 (它可能在寄存器上，也可能在栈上；在 x86 上，你可以在 `$esp+4` 找到它)。但是这是很痛苦的，因为现在你必须至少知道四种体系结构 (x86，x86-64，armv7，armv64)。想象你需要花多少时间去学习命令集以及它们每一个的调用约定 (http://en.m.wikipedia.org/wiki/Calling_convention)，然后正确的写一个在你的超类上设置断点并且条件正确的命令。幸运的是，这个在 Chisel (https://github.com/facebook/chisel) 被解决了。这被成为 `bmessage`：

```
(lldb) bmessage -[MyViewController viewDidAppear:]
Setting a breakpoint at -[UIViewController viewDidAppear:] with condition (void*)o
bject_getClass((id)$rdi) == 0x000000010e2f4d28
Breakpoint 1: where = UIKit`-[UIViewController viewDidAppear:], address = 0x000000
010e11533c
```

LLDB 和 Python

LLDB 有内建的，完整的 Python (http://lldb.llvm.org/python-reference.html) 支持。在LLDB中输入 `script`，会打开一个 Python REPL。你也可以输入一行 python 语句作为 `script 命令` 的参数，这可以运行 python 语句而不进入REPL：

```
(lldb) script import os
(lldb) script os.system("open http://www.objc.io/")
```

这样就允许你创造各种酷的命令。把下面的语句放到文件 `~/myCommands.py` 中：

```
def caflushCommand(debugger, command, result, internal_dict):
  debugger.HandleCommand("e (void)[CATransaction flush]")
```

然后再 LLDB 中运行：

```
command script import ~/myCommands.py
```

或者把这行命令放在 `/.lldbinit` 里，这样每次进入 LLDB 时都会自动运行。Chisel (https://github.com/facebook/chisel) 其实就是一个 Python 脚本的集合，这些脚本拼接 (命令) 字符串 ，然后让 LLDB 执行。很简单，不是吗?

紧握调试器这一武器

LLDB 可以做的事情很多。大多数人习惯于使用 `p`， `po`， `n`， `s` 和 `c`，但实际上除此之外，LLDB 可以做的还有很多。掌握所有的命令 (实际上并不是很多)，会让你在揭示代码运行时的运行状态，寻找 bug，强制执行特定的运行路径时获得更大的能力。你甚至可以构建简单的交互原型 - 比如要是现在以 modal 方式弹出一个 View Controller 会怎么样? 使用调试器，一试便知。

这篇文章是为了想你展示 LLDB 的强大之处，并且鼓励你多去探索在控制台输入命令。

打开 LLDB，输入 `help`，看一看列举的命令。你尝试过多少? 用了多少?

但愿 `NSLog` 看起来不再那么吸引你去用，每次编辑再运行并不有趣而且耗时。

调试愉快!

原文 Dancing in the Debugger — A Waltz with LLDB (http://www.objc.io/issue-19/lldb-debugging.html)

译者简介

nangege (https://github.com/nangege)
现在是程序员，希望以后成为会写代码的设计师。喜欢简单的东西。