Protocol和泛型的台前幕后

编译器是如何理解面向protocol编程的?

(https://www.boxueio.com/series/protocol-and-generic/ebook/193)

编译器是如何理解泛型编程的?

(https://www.boxueio.com/series/protocol-and-generic/ebook/195)

什么是value witness table?

Back to series (/series/protocol-and-generic)

字号
字号
默认主题
金色主题
暗色主题

欢迎回来，通过上一节提到的existential container我们知道，一个实现了 `protocol` 的对象，它的大小决定了existential container的创建方式。于是，为了在实际创建这样的对象时屏蔽掉这个差异，Swift引入了一个叫做Value Witness Table的结构（以下简称VWT）。

为了了解Swift是如何使用VWT的，我们先添加下面的代码：

```
func drawShape(_ shape: Drawable) {
    shape.draw()
}

drawShape(line)
```

这样，在调用 `drawShape` 时，就会在当前的上下文中创建一个参数的拷贝传递给 `drawShape` （因为existential container是通过 `struct` 实现的，它是一个值类型）。我们观察这个传参的过程以及 `drawShape` 的实现，就能理解VWT的用法了。

先在 `drawShape(line)` 设置一个断点，并让LLDB停在这里，在LLDB中执行 `di -s $rip -c 16`，观察生成的汇编指令：

```
(lldb) di -s $rip -c 16
WhatIsVWT`main:
->  0x100001d27 <+183>: mov    qword ptr [rbp - 0x30], rdx
    0x100001d2b <+187>: mov    qword ptr [rbp - 0x28], rcx
    0x100001d2f <+191>: call   0x100001de0
    0x100001d34 <+196>: lea    rdi, [rbp - 0x48]
    0x100001d38 <+200>: mov    qword ptr [rbp - 0x48], rax
    0x100001d3c <+204>: mov    rcx, qword ptr [rip + 0x3e0ddd]
    0x100001d43 <+211>: mov    rdx, qword ptr [rip + 0x3e0dde]
    0x100001d4a <+218>: mov    rsi, qword ptr [rip + 0x3e0ddf]
    0x100001d51 <+225>: mov    r8, qword ptr [rip + 0x3e0de0]
    0x100001d58 <+232>: mov    qword ptr [rax], rcx
    0x100001d5b <+235>: mov    qword ptr [rax + 0x8], rdx
    0x100001d5f <+239>: mov    qword ptr [rax + 0x10], rsi
    0x100001d63 <+243>: mov    qword ptr [rax + 0x18], r8
    0x100001d67 <+247>: call   0x100001d80               ; WhatIsVWT.drawS
hape (WhatIsVWT.Drawable) -> () at main.swift:3
    0x100001d6c <+252>: xor    eax, eax
    0x100001d6e <+254>: add    rsp, 0x60
```

从LLDB的提示可以看到，调用 `drawShape` 的代码在 `0x100001d67`，我们在这里设置个断点，并执行到这里：

```
(lldb) br s -a 0x100001d67
Breakpoint 2: where = WhatIsVWT`main + 247 at main.swift:8, address = 0x00
00000100001d67
(lldb) con
Process 38291 resuming
```

此时， `rdi` 应该是 `drawShape` 的第一个参数，也就是 `line` 的一个拷贝，因此，它应该存放着 `Line` 对象的existential container。我们先看下它的值：

```
(lldb) x -s8 -c5 -fx $rdi
0x7fff5fbff728: 0x0000000100b00000 0x000000010080d4c0
0x7fff5fbff738: 0x0000000000000000 0x000000010038d5a8
0x7fff5fbff748: 0x000000010038d3d8
```

可以看到，由于 `Line` 对象无法直接存放在value buffer里，这里的第一个格子存放的是一个内存地址。而这个地址引用的，应该是 `Line` 对象的坐标值。我们继续查看一下：

```
(lldb) x -s8 -c4 -fx 0x0000000100b00000
0x100b00000: 0x0000000000000002 0x0000000000000002
0x100b00010: 0x0000000000000006 0x0000000000000006
```

可以看到，这四个值，就是传递给 drawShape 的 line 对象的起点和终点。

> 还有一点要说明的是，在 drawShape 参数的existential container中，尽管我们没有用到第二和第三个格子，但此时，第二个格子的值也不为0。因此，对于这种存储大对象的情况，你只能认为第二和第三个格子的内容是未定义的，而不能认为它们一定是0。

了解了 rdi 参数之后，我们还是回到 drawShape 的代码，执行下 di -l 确认下我们还停留在之前的 call 指令上：

```
(lldb) di -l
WhatIsVWT`main + 247 at main.swift:8
   7    let line = Line(x1: 2, y1: 2, x2: 6, y2: 6)
-> 8    drawShape(line)
WhatIsVWT`main:
->  0x100001d67 <+247>: call   0x100001d80               ; WhatIsVWT.drawS
hape (WhatIsVWT.Drawable) -> () at main.swift:3
    0x100001d6c <+252>: xor    eax, eax
    0x100001d6e <+254>: add    rsp, 0x60
    0x100001d72 <+258>: pop    rbp
    0x100001d73 <+259>: ret
```

此时，先别着急调用，我们先反汇编一下 0x100001d80 这个地址，大致了解下它会执行什么逻辑：

```
(lldb) di -s 0x100001d80 -c 30
WhatIsVWT`drawShape(Drawable) -> ():
    0x100001d80 <+0>:  push   rbp
    0x100001d81 <+1>:  mov    rbp, rsp
    0x100001d84 <+4>:  sub    rsp, 0x20
    0x100001d88 <+8>:  mov    qword ptr [rbp - 0x8], rdi
    0x100001d8c <+12>: mov    rax, qword ptr [rdi + 0x18]
    0x100001d90 <+16>: mov    rcx, qword ptr [rax - 0x8]
    0x100001d94 <+20>: mov    qword ptr [rbp - 0x10], rdi
    0x100001d98 <+24>: mov    rsi, rax
    0x100001d9b <+27>: mov    qword ptr [rbp - 0x18], rax
    0x100001d9f <+31>: call   qword ptr [rcx + 0x10]
    0x100001da2 <+34>: mov    rcx, qword ptr [rbp - 0x10]
    0x100001da6 <+38>: mov    rsi, qword ptr [rcx + 0x20]
    0x100001daa <+42>: mov    rdi, rax
    0x100001dad <+45>: mov    rax, qword ptr [rbp - 0x18]
    0x100001db1 <+49>: mov    qword ptr [rbp - 0x20], rsi
    0x100001db5 <+53>: mov    rsi, rax
    0x100001db8 <+56>: mov    rdx, qword ptr [rbp - 0x20]
    0x100001dbc <+60>: mov    r8, qword ptr [rbp - 0x20]
    0x100001dc0 <+64>: call   qword ptr [r8]
    0x100001dc3 <+67>: mov    rax, qword ptr [rbp - 0x10]
    0x100001dc7 <+71>: mov    rcx, qword ptr [rax + 0x18]
    0x100001dcb <+75>: mov    rdx, qword ptr [rcx - 0x8]
    0x100001dcf <+79>: mov    rdi, rax
    0x100001dd2 <+82>: mov    rsi, rcx
    0x100001dd5 <+85>: call   qword ptr [rdx]
    0x100001dd7 <+87>: add    rsp, 0x20
    0x100001ddb <+91>: pop    rbp
    0x100001ddc <+92>: ret
```

可以看到，一个源代码很简单的 drawShape 居然生成了这么多条汇编指令，如果你第一次看到这个结果，多少还是会有些意外吧。不过别着急，抓住这些汇编指令中的 call ，就可以很容易理解这段代码的功能了。在这段代码里，一共有三处 call 指令，我们就先从第一个开始。

在 0x100001d9f 打个断点并执行到这里：

```
(lldb) br s -a 0x100001d9f
Breakpoint 3: where = WhatIsVWT`WhatIsVWT.drawShape (WhatIsVWT.Drawable) -
> () + 31 at main.swift:4, address = 0x0000000100001d9f
(lldb) con
Process 38291 resuming

(lldb) di -l
WhatIsVWT`WhatIsVWT.drawShape (WhatIsVWT.Drawable) -> () + 12 at main.swif
t:4
   3    func drawShape(_ shape: Drawable) {
   4        shape.draw()
   5    }
WhatIsVWT`drawShape(Drawable) -> ():
    0x100001d8c <+12>: mov    rax, qword ptr [rdi + 0x18]
    0x100001d90 <+16>: mov    rcx, qword ptr [rax - 0x8]
    0x100001d94 <+20>: mov    qword ptr [rbp - 0x10], rdi
    0x100001d98 <+24>: mov    rsi, rax
    0x100001d9b <+27>: mov    qword ptr [rbp - 0x18], rax
->  0x100001d9f <+31>: call   qword ptr [rcx + 0x10]
```

然后，查看下常用寄存器的值：

```
(lldb) re r rdi rsi rdx rcx rax rsp
     rdi = 0x00007fff5fbff728
     rsi = 0x0000000010038d5a8  WhatIsVWT`type metadata for WhatIsVWT.Line
     rdx = 0x0000000000000002
     rcx = 0x0000000010038d500  WhatIsVWT`value witness table for WhatIsVW
T.Line
     rax = 0x0000000010038d5a8  WhatIsVWT`type metadata for WhatIsVWT.Line
     rsp = 0x00007fff5fbff6e0
```

从LLDB的提示中我们可以看到， `rdi` 保存的是 `drawShpe` 的参数，也就是一个 `Line` 对象。 `rsi` 是 `Line` 的metadata。而 `rcx` ，则是我们这段视频的主角，它是 `Line` 的value witness table，这里保存着负责 `Line` 对象创建和销毁的方法。那么 `[rcx + 0x10]` 这里到底是什么呢？

我们先看下这个地址保存的内容：

```
(lldb) x -s8 -c1 -fx $rcx+0x10
0x10038d510: 0x0000000100002810
```

按照推测， `0x0000000100002810` 应该是一个方法的地址，我们反汇编一下：

```
(lldb) di -s 0x0000000100002810
WhatIsVWT`projectBuffer for Line:
    0x100002810 <+0>:  push   rbp
    0x100002811 <+1>:  mov    rbp, rsp
    0x100002814 <+4>:  mov    rdi, qword ptr [rdi]
    0x100002817 <+7>:  mov    rax, rdi
    0x10000281a <+10>: mov    qword ptr [rbp - 0x8], rsi
    0x10000281e <+14>: pop    rbp
    0x10000281f <+15>: ret
```

可以看到，这是一个叫 `projectBuffer` 的方法，而它最核心的功能就是下面这两行汇编指令：

```
0x100002814 <+4>:  mov    rdi, qword ptr [rdi]
0x100002817 <+7>:  mov    rax, rdi
```

我们知道， `rdi` 是 `drawShape` 参数的existential container，于是它的第一个格子存放的，就是保存 `Line` 对象值的地址。那么第一条指令的作用就是把这个地址读出来放到 `rdi` 寄存器，第二条指令的作用，就是把这个地址作为 `projectBuffer` 的返回值返回。因此， `drawShape` 的第一个 `call` 指令的目的我们就知道了，它就是返回保存参数 `Line` 对象值的地址。

然后，执行 `di -f` 确认下当前的位置还在 `drawShape` 方法里：

```
(lldb) di -f
WhatIsVWT`drawShape(Drawable) -> ():
    ...
    0x100001d94 <+20>: mov    qword ptr [rbp - 0x10], rdi
    0x100001d98 <+24>: mov    rsi, rax
    0x100001d9b <+27>: mov    qword ptr [rbp - 0x18], rax
->  0x100001d9f <+31>: call   qword ptr [rcx + 0x10]
    0x100001da2 <+34>: mov    rcx, qword ptr [rbp - 0x10]
    0x100001da6 <+38>: mov    rsi, qword ptr [rcx + 0x20]
    0x100001daa <+42>: mov    rdi, rax
    0x100001dad <+45>: mov    rax, qword ptr [rbp - 0x18]
    0x100001db1 <+49>: mov    qword ptr [rbp - 0x20], rsi
    0x100001db5 <+53>: mov    rsi, rax
    0x100001db8 <+56>: mov    rdx, qword ptr [rbp - 0x20]
    0x100001dbc <+60>: mov    r8, qword ptr [rbp - 0x20]
    0x100001dc0 <+64>: call   qword ptr [r8]
    0x100001dc3 <+67>: mov    rax, qword ptr [rbp - 0x10]
    0x100001dc7 <+71>: mov    rcx, qword ptr [rax + 0x18]
    ...
```

我们把目标移动到第二个 call 指令：

```
(lldb) br s -a 0x100001dc0
Breakpoint 4: where = WhatIsVWT`WhatIsVWT.drawShape (WhatIsVWT.Drawable) -
> () + 64 at main.swift:4, address = 0x0000000100001dc0
(lldb) con
Process 38291

(lldb) di -l
WhatIsVWT`WhatIsVWT.drawShape (WhatIsVWT.Drawable) -> () + 42 at main.swif
t:4
   3    func drawShape(_ shape: Drawable) {
   4        shape.draw()
   5    }
WhatIsVWT`drawShape(Drawable) -> ():
    0x100001daa <+42>: mov    rdi, rax
    0x100001dad <+45>: mov    rax, qword ptr [rbp - 0x18]
    0x100001db1 <+49>: mov    qword ptr [rbp - 0x20], rsi
    0x100001db5 <+53>: mov    rsi, rax
    0x100001db8 <+56>: mov    rdx, qword ptr [rbp - 0x20]
    0x100001dbc <+60>: mov    r8, qword ptr [rbp - 0x20]
->  0x100001dc0 <+64>: call   qword ptr [r8]
```

通过观察之前的汇编指令我们知道， rdi 就是 Line 对象的坐标值。因此，不出意外，这里，就应该是调用 draw() 方法了。先查看下 r8 寄存器的值：

```
(lldb) re r r8
      r8 = 0x000000010038d3d8  WhatIsVWT`protocol witness table for WhatIs
VWT.Line : WhatIsVWT.Drawable in WhatIsVWT
```

可以看到，这是 Line 的protocol witness table，由于 Drawable 只约定了一个方法，因此它的第一个地址，应该就是 Line.draw() 的实现了。为了确认我们的推测，首先读一下 r8 这个地址保存的值：

```
(lldb) x -s8 -c1 -fx 0x000000010038d3d8
0x10038d3d8: 0x0000000100002590
```

然后，反汇编下这个地址：

```
(lldb) di -s 0x0000000100002590 -c 6
WhatIsVWT`protocol witness for Drawable.draw() -> () in conformance Line:
    0x100002590 <+0>:  push   rbp
    0x100002591 <+1>:  mov    rbp, rsp
    0x100002594 <+4>:  sub    rsp, 0x30
    0x100002598 <+8>:  lea    rax, [rbp - 0x20]
    0x10000259c <+12>: mov    rcx, qword ptr [rdi]
    0x10000259f <+15>: mov    r8, qword ptr [rdi + 0x8]
```

从LLDB的提示中可以看到，这的确就是 Line.draw() 的实现。因此， drawShape 的第二个 call 指令的目的我们也了解了，还剩下最后一个。我们执行下 di -f 找到最后一条 call 指令的地址：

```
(lldb) di -f
WhatIsVWT`drawShape(Drawable) -> ():
    ...
    0x100001dbc <+60>: mov    r8, qword ptr [rbp - 0x20]
->  0x100001dc0 <+64>: call   qword ptr [r8]
    0x100001dc3 <+67>: mov    rax, qword ptr [rbp - 0x10]
    0x100001dc7 <+71>: mov    rcx, qword ptr [rax + 0x18]
    0x100001dcb <+75>: mov    rdx, qword ptr [rcx - 0x8]
    0x100001dcf <+79>: mov    rdi, rax
    0x100001dd2 <+82>: mov    rsi, rcx
    0x100001dd5 <+85>: call   qword ptr [rdx]
    0x100001dd7 <+87>: add    rsp, 0x20
    0x100001ddb <+91>: pop    rbp
    0x100001ddc <+92>: ret
```

在 0x100001dd5 打个断点并执行到这里：

```
(lldb) br s -a 0x100001dd5
Breakpoint 3: where = WhatIsVWT`WhatIsVWT.drawShape (WhatIsVWT.Drawable) -
> () + 85 at main.swift:5, address = 0x0000000100001dd5
(lldb) con
Process 38958 resuming

(lldb) di -l
WhatIsVWT`WhatIsVWT.drawShape (WhatIsVWT.Drawable) -> () + 67 at main.swif
t:5
   4        shape.draw()
   5    }
   6
WhatIsVWT`drawShape(Drawable) -> ():
    0x100001dc3 <+67>: mov    rax, qword ptr [rbp - 0x10]
    0x100001dc7 <+71>: mov    rcx, qword ptr [rax + 0x18]
    0x100001dcb <+75>: mov    rdx, qword ptr [rcx - 0x8]
    0x100001dcf <+79>: mov    rdi, rax
    0x100001dd2 <+82>: mov    rsi, rcx
->  0x100001dd5 <+85>: call   qword ptr [rdx]
    0x100001dd7 <+87>: add    rsp, 0x20
    0x100001ddb <+91>: pop    rbp
    0x100001ddc <+92>: ret
```

现在，这条 call 指令用来做什么呢？在没什么思绪的时候，我们就先看下常用寄存器，它们通常能给我们一些提示：

```
(lldb) re r rdi rsi rdx rcx
     rdi = 0x00007fff5fbff728
     rsi = 0x000000010038d5a8  WhatIsVWT`type metadata for WhatIsVWT.Line
     rdx = 0x000000010038d500  WhatIsVWT`value witness table for WhatIsVWT
.Line
     rcx = 0x000000010038d5a8  WhatIsVWT`type metadata for WhatIsVWT.Line
```

可以看到， rdi 是 drawShape 参数的existential container； rsi 是 Line 的metadata；而 rdx 则又是 Line 的value witness table，这次调用的这个函数又是做什么的呢？

我们先查看下 Line VWT中第一个位置保存的地址：

```
(lldb) x -s8 -c1 -fx $rdx
0x10038d500: 0x0000000100002760
```

然后反汇编一下 0x0000000100002760：

```
(lldb) di -s 0x0000000100002760 -c 15
WhatIsVWT`destroyBuffer for Line:
    0x100002760 <+0>:  push   rbp
    0x100002761 <+1>:  mov    rbp, rsp
    0x100002764 <+4>:  sub    rsp, 0x10
    0x100002768 <+8>:  mov    eax, 0x20
    0x10000276d <+13>: mov    ecx, eax
    0x10000276f <+15>: mov    eax, 0x7
    0x100002774 <+20>: mov    edx, eax
    0x100002776 <+22>: mov    rdi, qword ptr [rdi]
    0x100002779 <+25>: mov    qword ptr [rbp - 0x8], rsi
    0x10000277d <+29>: mov    rsi, rcx
    0x100002780 <+32>: call   0x100002790               ; rt_swift_slowDea
lloc
    0x100002785 <+37>: add    rsp, 0x10
    0x100002789 <+41>: pop    rbp
    0x10000278a <+42>: ret
```

可以看到，是一个叫做 `destroyBuffer` 的函数，其中关键的指令是下面4行：

```
0x100002776 <+22>: mov    rdi, qword ptr [rdi]
0x100002779 <+25>: mov    qword ptr [rbp - 0x8], rsi
0x10000277d <+29>: mov    rsi, rcx
0x100002780 <+32>: call   0x100002790               ; rt_swift_slowDealloc
```

也就是说，它先读取了保存 `Line` 值的buffer的地址，把它作为 `rt_swift_slowDealloc` 的第一个参数；然后，把 `Line` 的metadata传给了 `rsi` ，作为了 `rt_swift_slowDealloc` 的第二个参数。由此，我们不难推测，`rt_swift_slowDealloc` 的作用，就是根据 `Line` metadata中的信息，回收存储 `Line` 对象值的buffer，而这也就是 `destroyBuffer` 的作用。

至此， `drawShape` 方法中的三个 `call` 指令就都分析完了，程序的执行也就结束了。

以上，就是VWT在创建 `protocol` 类型对象的过程中扮演的角色。面对不同大小的对象，VWT中包含了不同的用于初始化以及销毁buffer的方法，它们其中有的是内存的拷贝、有的会调用必要的 `init` / `deinit` 方法，当然也有些其实什么都不用做。理解了VWT的工作方式，你也可以创建不同大小和类型的对象自己观察一下。

What's next?

以上，就是Swift处理 `protocol` 类型的工作方式。正是PWT，existential container以及VWT这三者紧密合作，才为Swift提供了面向 `protocol` 编程的能力。

接下来，如果我们往前多想一步就会发现，其实一个 `protocol` 类型和一个泛型类型，真的就只有一步之遥，只是 `protocol` 对类型多了一些约束而已。在目前的Swift版本中，泛型编程和 `protocol` 几乎总是同时出现的，如果我们不通过 `protocol` 对类型做出约束，很难对泛型参数进行处理。在下一节，我们就基于已经掌握的知识，了解下Swift是如何处理泛型参数的。

◀ 编译器是如何理解面向protocol编程的?
(https://www.boxueio.com/series/protocol-and-generic/ebook/193)

编译器是如何理解泛型编程的? ▶
(https://www.boxueio.com/series/protocol-and-generic/ebook/195)

Jan 9, 2016

猎云网采访报道泊学 (http://www.lieyunwang.com/archives/144329)
Dec 31, 2015

What most schools do not teach (https://www.boxueio.com/what-most-schools-do-not-teach)
Dec 21, 2015

一个工作十年PM终创业的故事（一） (https://www.boxueio.com/founder-story)
May 8, 2015

泊学相关

关于泊学 ›

加入泊学 ›

泊学用户隐私以及服务条款 (HTTPS://WWW.BOXUEIO.COM/TERMS-OF-SERVICE)

版权声明 (HTTPS://WWW.BOXUEIO.COM/COPYRIGHT-STATEMENT)

联系泊学

Email: 10[AT]boxue.io (mailto:10@boxue.io)

QQ: 2085489246

友情链接 SwiftV (http://www.swiftv.cn) | Seay信息安全博客 (http://www.cnseay.com) | Swift.gg (http://swift.gg/) | Laravist (http://laravist.com/) | SegmentFault (https://segmentfault.com) | 靛青K的博客 (http://blog.dianqk.org/)