Protocol和泛型的台前幕后

如何通过泛型编程简化网络请求?
(https://www.boxueio.com/series/protocol-and-generic/ebook/192)

什么是value witness table?
(https://www.boxueio.com/series/protocol-and-generic/ebook/194)

编译器是如何理解面向protocol编程的?

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

字号
字号
默认主题
金色主题
暗色主题

从某种意义上说，protocol 也算是某种形式的“泛型”类型，只不过所有的类型都需要遵从一些共同的行为约束。通过 protocol，无关的两个值类型也可以实现类似多态的效果，来看下面这个例子。

首先，我们定义一个表示可绘制类型的 protocol：

```
protocol Drawable {
    func draw()
}
```

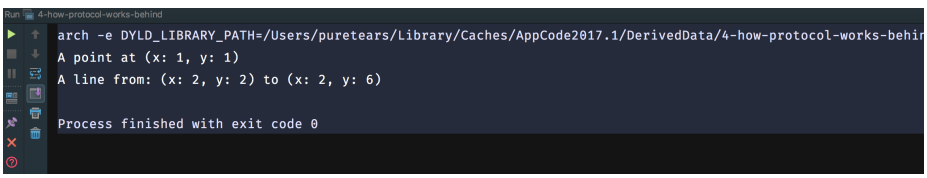
其次，定义两个实现了 Drawable 的 struct：

```
struct Point: Drawable {
    var x: Int
    var y: Int

    func draw() {
        print("A point at (x: \(x), y: \(y))")
    }
}

struct Line: Drawable {
    var x1: Int // From position
    var y1: Int
    var x2: Int // To position
    var y2: Int

    func draw() {
        print("A line from: (x: \(x1), y: \(y1)) " +
                "to (x: \(x1), y: \(y2))")
    }
}
```

这里，Point 和 Line 是完全无关的两个类型。然后，我们看下面的测试代码：

```
let point: Drawable = Point(x: 1, y: 1)
point.draw()

let line: Drawable = Line(x1: 2, y1: 2, x2: 6, y2: 6)
line.draw()
```

执行一下就会发现同样是 Drawable 类型，point 和 line 却调用了各自版本的 draw 方法。

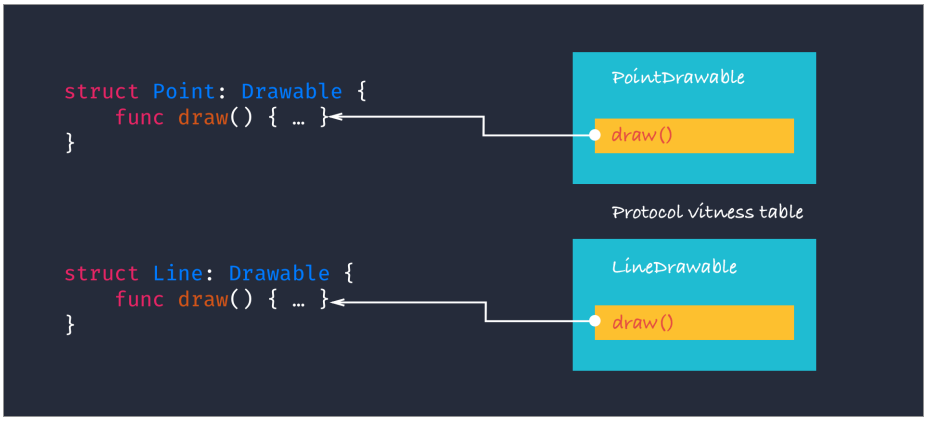


其实这就是我们经常说起的protocol oriented programming。它让value类型也有了实现多态的能力。

Protocol witness table

但仔细想想，Point 和 Line 并没有继承关系，我们无法通过之前讲过的witness table的机制实现方法的动态派发，编译器是如何为这两个类型的对象选择方法的呢?

实际上，对于每一个实现了 protocol 的类型，编译器都会创建一个叫做protocol witness table的对象，其中存放了这个类型实现的每一个 protocol 方法的地址。因此，对于我们的 Point 和 Line 来说，这个表是这样的：



我们可以在LLDB中观察一下。在 point.draw() 上设置断点，运行起来之后，等LLDB停在断点上，我们执行 di -s $rip -c 8 ，可以看到类似这样的结果：

Swift调试上下文中不支持寄存器访问

```
(lldb) di -s $rip -c 8
GenericDemo`main:
->  0x100001d07 <+279>: mov    rax, qword ptr [rip + 0x3e0e2a]
    0x100001d0e <+286>: mov    rcx, qword ptr [rax - 0x8]
    0x100001d12 <+290>: mov    rsi, rax
    0x100001d15 <+293>: mov    qword ptr [rbp - 0x40], rax
    0x100001d19 <+297>: call   qword ptr [rcx + 0x10]
```

这里的 call 指令在执行什么呢？我们并没有很多线索。只能执行4次 si 命令，把 rip 移动到 call 指令这里。

先执行 di -l 确认下当前位置：

```
(lldb) di -l
4-how-protocol-works-behind`main + 279 at main.swift:6
   5
   6    point.draw()
   7    line.draw()
4-how-protocol-works-behind`main:
    0x100001d07 <+279>: mov    rax, qword ptr [rip + 0x3e0e2a]
    0x100001d0e <+286>: mov    rcx, qword ptr [rax - 0x8]
    0x100001d12 <+290>: mov    rsi, rax
    0x100001d15 <+293>: mov    qword ptr [rbp - 0x40], rax
->  0x100001d19 <+297>: call   qword ptr [rcx + 0x10]
    0x100001d1c <+300>: mov    rcx, qword ptr [rip + 0x3e0e1d]
```

从结果中可以看到，的确停在了 call 指令上，然后，我们先看下 rcx+0x10 这个地址保存的内容：

```
(lldb) x -s8 -c1 -fx $rcx+0x10
0x10038d440: 0x00000001000025c0
```

再反汇编下 0x00000001000025c0 这个地址：

```
(lldb) di -s 0x00000001000025c0 -c 4
GenericDemo`__swift_noop_self_return:
    0x1000025c0 <+0>: mov    rax, rdi
    0x1000025c3 <+3>: mov    qword ptr [rsp - 0x8], rsi
    0x1000025c8 <+8>: ret
```

从这个 __swift_noop_self_return 的名字就可以推测到，不是什么有用的东西。我们就只能顺着当前指令的执行继续找了：

```
(lldb) di -s $rip -c 8
GenericDemo`main:
->  0x100001d19 <+297>: call   qword ptr [rcx + 0x10]
    0x100001d1c <+300>: mov    rcx, qword ptr [rip + 0x3e0e1d]
    0x100001d23 <+307>: mov    rdi, rax
    0x100001d26 <+310>: mov    rsi, qword ptr [rbp - 0x40]
    0x100001d2a <+314>: mov    rdx, rcx
    0x100001d2d <+317>: call   qword ptr [rcx]
```

我们的下一个目标，自然就是处于 0x100001d2d 这里的 call 指令。执行 break set -a 0x100001d2d 在这里打个断点，并执行 continue 执行到这里。

等LLDB再次停下来的时候，执行 di -l 确认下我们处在正确的位置：

```
(lldb) di -l
GenericDemo`main + 307 at main.swift:6
   5
   6    point.draw()
   7    line.draw()
GenericDemo`main:
    0x100001d23 <+307>: mov    rdi, rax
    0x100001d26 <+310>: mov    rsi, qword ptr [rbp - 0x40]
    0x100001d2a <+314>: mov    rdx, rcx
->  0x100001d2d <+317>: call   qword ptr [rcx]
```

此时，先别着急反汇编 `[rcx]` 这里的内容，我们先来看下常用的CPU的通用寄存器，这有助于你了解要调用的方法。我们执行 `register read rdi rsi rcx rdx rax`：

> 为什么是它们呢？因为在AMD 64 ABI规范中，它们是常用的传递整数参数和返回值的寄存器。

```
(lldb) re r rdi rsi rdx rcx rax
     rdi = 0x00000001003e2b20  GenericDemo`__GenericDemo.point : __Generic
Demo.Drawable
     rsi = 0x0000000010038d4d8  GenericDemo`type metadata for __GenericDemo
.Point
     rdx = 0x0000000010038d3d0  GenericDemo`protocol witness table for __Ge
nericDemo.Point : __GenericDemo.Drawable in __GenericDemo
     rcx = 0x0000000010038d3d0  GenericDemo`protocol witness table for __Ge
nericDemo.Point : __GenericDemo.Drawable in __GenericDemo
     rax = 0x00000001003e2b20  GenericDemo`__GenericDemo.point : __Generic
Demo.Drawable
```

这次，终于看到点眉目了。首先， `rcx` 是 `struct Point` 的PWT，按照我们之前的说明，这里应该存放的就是 `Point.draw` 方法的地址；其次， `rdi` 中存放的是 `Point` 对象的地址，它应该是调用 `draw` 方法的第一个参数，也就是我们经常提到的 `self` 。

> `rdi` 是AMD64 ABI规范中用于传递第一个整数参数的寄存器。

为了验证我们的推断，先来看下 `rdi` 指向的内容：

```
(lldb) x -s8 -c2 -fx $rdi
0x1003e2b20: 0x0000000000000001 0x0000000000000001
```

看到了吧，就是我们开始定义的xy都为1的 `Point` 对象。然后，我们再查看下 `rcx` 这个地址的内容：

```
(lldb) x -s8 -c1 -fx $rcx
0x10038d3d0: 0x0000000100001ff0
```

再反汇编下 `0x0000000100001ff0` 这个地址：

```
(lldb) di -s 0x0000000100001ff0 -c 4
GenericDemo`protocol witness for Drawable.draw() -> () in conformance Poin
t:
    0x100001ff0 <+0>: push   rbp
    0x100001ff1 <+1>: mov    rbp, rsp
    0x100001ff4 <+4>: sub    rsp, 0x20
    0x100001ff8 <+8>: mov    rax, qword ptr [rdi]
```

从LLDB的提示中，我们已经可以最终确认，这就是 `Point.draw()` 方法了。

> 理解了这个过程之后，你可以试着自己分析下 `Line` 的PWT。

Existential Container

“较小对象”的处理方式

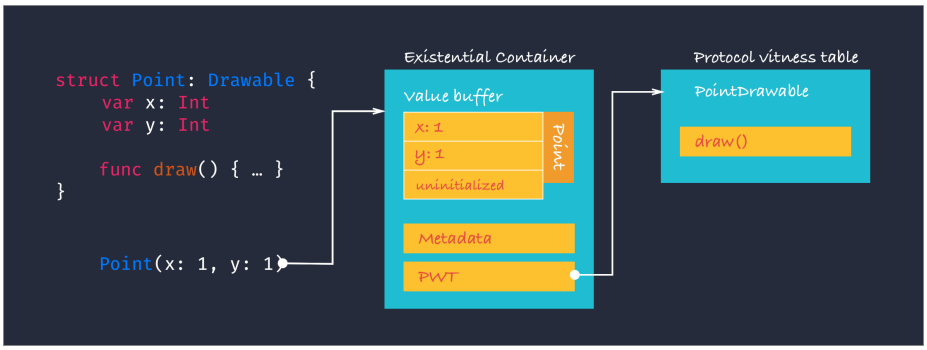
在解决了识别不同类型实现的 `protocol` 方法之后，为了通过 `protocol` 类型实现多态的效果，我们还需要解决另外一个问题。当我们定义一个 `[Drawable]` 时：

```
let shapes: [Drawable] = [point, line]
```

我们应该为 `shapes` 的存储分配多少内存空间呢？按照我们之前对 `struct` 的理解，在64位平台上， `Point` 是两个 `Int` ，应该是16字节； `Line` 是4个 `Int` ，应该是32字节。但 `Array` 中的每个元素应该是大小相等的，这样我们才能用固定的偏移值访问 `Array` 中不同的元素。由于 `point` 和 `line` 都是值类型，我们又无法像类对象一样，在数组中存放这两个对象的引用。该怎么办呢？

为此，我们只能人为创建一个中间层，让这个中间层把每一个实现了 protocol 的类型的对象封装起来，并且，让封装之后的对象都有相同的大小。这样，把封装后的对象放到 Array 里，一切就都顺理成章了。而这个所谓的中间层，就是existential container。

那么，这个container里究竟有什么呢？我们可以把它想象成5个格子，在64位平台上，每个格子都是8字节。前3个格子叫做value buffer，用来存放被封装的对象的值，第4个“格子”保存指向类型metadata的引用，稍后我们会看到它的用法，第5个“格子”用来存放该类型的PWT。于是， point 对象的existential container看上去是这样的：



同样，我们可以在LLDB中查看这个container对象。首先，我们添加下面两条语句：

```
print(MemoryLayout.size(ofValue: point))
print(MemoryLayout.size(ofValue: line))
```

执行一下就会发现，两个值都是40：

这也正好印证了我们提到的，一个container对象包含5个8字节“格子”的事实。其次，我们还是在 point.draw() 这里设置一个断点，等LLDB断下来之后，执行 di -l 确认下位置：

```
(lldb) di -l
4-how-protocol-works-behind`main + 288 at main.swift:6
   5
-> 6    point.draw()
   7    line.draw()
4-how-protocol-works-behind`main:
->  0x100001ae0 <+288>: mov    rax, qword ptr [rip + 0x3e1051]
    0x100001ae7 <+295>: mov    rcx, qword ptr [rax - 0x8]
    0x100001aeb <+299>: mov    rsi, rax
    0x100001aee <+302>: mov    qword ptr [rbp - 0x90], rax
    0x100001af5 <+309>: call   qword ptr [rcx + 0x10]
    0x100001af8 <+312>: mov    rcx, qword ptr [rip + 0x3e1041]
```

执行 br s -a 0x100001af5 让程序断在下一条 call 指令上，并执行 con 执行到这里。然后，执行 re r rdi 查看 rdi 寄存器的值：

```
(lldb) re r rdi
     rdi = 0x00000001003e2b20  4-how-protocol-works-behind`__how_protocol_
works_behind.point : __how_protocol_works_behind.Drawable
```

可以看到，这就是 point 对象的地址，我们执行 x -s8 -c5 -fx $rdi 查看一下这个地址的值，这就是existential container了：

```
(lldb) x -s8 -c5 -fx $rdi
0x1003e2b20: 0x0000000000000001 0x0000000000000001
0x1003e2b30: 0x0000000000000000 0x000000010038d4d8
0x1003e2b40: 0x000000010038d3d0
```

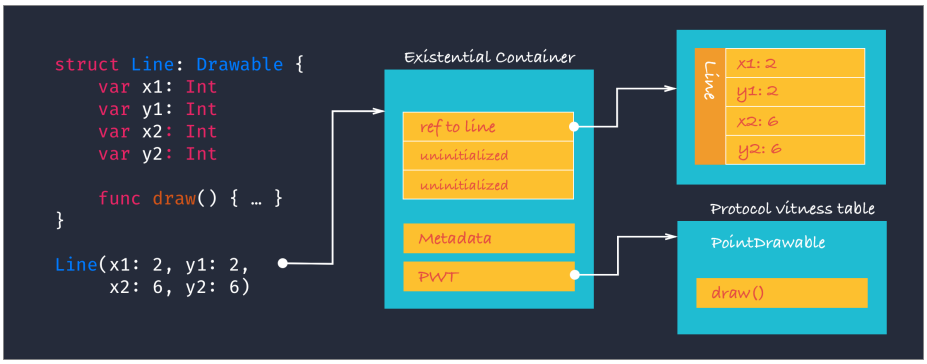
按照我们之前的理解，它的前两个“格子”是 point.x 和 point.y 的值，第三个“格子”我们没有使用，第四个“格子”是类型metadata的地址，第五个“格子”是 Point 的PWT。我们可以通过下面的方式来确认下PWT的存在：

```
(lldb) x -s8 -c1 -fx 0x000000010038d3d0
0x10038d3d0: 0x0000000100002000
(lldb) di -s 0x0000000100002000 -c 5
4-how-protocol-works-behind`protocol witness for Drawable.draw() -> () in
 conformance Point:
    0x100002000 <+0>:  push   rbp
    0x100002001 <+1>:  mov    rbp, rsp
    0x100002004 <+4>:  sub    rsp, 0x20
    0x100002008 <+8>:  mov    rax, qword ptr [rdi]
    0x10000200b <+11>: mov    rdi, qword ptr [rdi + 0x8]
```

可以看到， `0x100002000` 这个地址保存的的确是 `Point.draw` 这个方法的地址，因此，container中的第五个“格子”里存放的，的确是 `Point` 的PWT。

“大型”对象的处理方式

看到这里，你可能会想了， `Point` 中有两个属性，还可以将就放在value buffer里。而 `Line` 中有4个属性，明显塞不进value buffer啊，这种情况该怎么办呢？这种value buffer装不下的对象，就是我们标题中描述的“大型”对象。对于这类对象，Swift会把对象创建在系统堆中，然后，只用value buffer的第一个“格子”存放这块内存的地址的引用。像这样：



为了观察这种对象的存储，我们可以在 `line.draw` 设置一个断点，让LLDB停在这里之后，执行 `di -l` 找到下一条 `call` 的地址：

```
(lldb) di -l
4-how-protocol-works-behind`main + 341 at main.swift:7
   6    point.draw()
-> 7    line.draw()
   8
4-how-protocol-works-behind`main:
->  0x100001b15 <+341>: mov    rax, qword ptr [rip + 0x3e1044] ; __how_pro
tocol_works_behind.line : __how_protocol_works_behind.Drawable + 24
    0x100001b1c <+348>: mov    rcx, qword ptr [rax - 0x8]
    0x100001b20 <+352>: mov    rsi, rax
    0x100001b23 <+355>: mov    qword ptr [rbp - 0x98], rax
    0x100001b2a <+362>: call   qword ptr [rcx + 0x10]
    0x100001b2d <+365>: mov    rcx, qword ptr [rip + 0x3e1034]
```

根据之前的经验我们知道， `0x100001b2a` 就应该是 `line.draw()` 的调用了。因此，在 `0x100001b2a` 设置个断点之后，执行到这里：

```
(lldb) br s -a 0x100001b2a
Breakpoint 2: where = 4-how-protocol-works-behind`main + 362 at main.swif
t:7, address = 0x0000000100001b2a
(lldb) con
Process 37163 resuming
(lldb) di -l
4-how-protocol-works-behind`main + 341 at main.swift:7
   6    point.draw()
   7    line.draw()
   8
4-how-protocol-works-behind`main:
    0x100001b15 <+341>: mov    rax, qword ptr [rip + 0x3e1044]
    0x100001b1c <+348>: mov    rcx, qword ptr [rax - 0x8]
    0x100001b20 <+352>: mov    rsi, rax
    0x100001b23 <+355>: mov    qword ptr [rbp - 0x98], rax
->  0x100001b2a <+362>: call   qword ptr [rcx + 0x10]
    0x100001b2d <+365>: mov    rcx, qword ptr [rip + 0x3e1034]
```

此时， `rdi` 中存放的，应该是 `line` 对象existential container的地址：

```
(lldb) re r rdi
     rdi = 0x00000001003e2b48  4-how-protocol-works-behind`__how_protocol_
works_behind.line : __how_protocol_works_behind.Drawable
```

我们执行 `x -s8 -c5 -fx $rdi` 来看一下：

```
(lldb) x -s8 -c5 -fx $rdi
0x1003e2b48: 0x0000000100903970 0x0000000000000000
0x1003e2b58: 0x0000000000000000 0x000000010038d5a8
0x1003e2b68: 0x000000010038d3d8
```

按照刚才我们的设计，value buffer中的第一个“格子”里存放的，应该是分配在系统堆里 `line` 对象的值。我们继续跟着这个地址去看看：

```
(lldb) x -s8 -c4 -fx 0x0000000100903970
0x100903970: 0x0000000000000002 0x0000000000000002
0x100903980: 0x0000000000000006 0x0000000000000006
```

看到了吧，这里保存的正是 `line` 的四个属性，记录了线段的起始和结束坐标。

What's next?

现在，我们已经了解了PWT以及existential container，知道了针对不同大小的对象，existential container的存储方式不同。接下来，我们需要一个统一的接口来处理在value buffer中创建和销毁对象的行为，在Swift里，这个接口就是value witness table（以下简称VWT）。别急，我知道这个话题里的新生事物已经够多的了，稍微休息一会儿，下一节，我们再来探索VWT的实现细节。

◀ 如何通过泛型编程简化网络请求?
(https://www.boxueio.com/series/protocol-and-generic/ebook/192)

什么是value witness table? ▶
(https://www.boxueio.com/series/protocol-and-generic/ebook/194)

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/)