Protocol和泛型的台前幕后

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

返回视频 (/series/protocol-and-generic)

编译器是如何理解泛型编程的?

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

字号

字号

默认主题

金色主题

暗色主题

在这一节里，基于对 protocol 实现方式的理解，我们来看Swift是如何处理泛型参数的。还是上一节中的 protocol Drawable，下面两个实现方式有什么差别呢?

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

func genericDraw<T: Drawable>(_ shape: T) {
    shape.draw()
}
```

从使用的角度来看，它们确实是相同的。但从实现方式的角度看，它们又截然不同。对于 draw 来说，我们已经看到了，它通过VWT和existential container在运行时实现了多态。而泛型版本的 genericDraw 则实现了编译期多态，编译器会根据调用 genericDraw 时使用的参数，把泛型类型 T 和参数的类型绑定起来。我们来看下面的例子:

```
let line = Line(x1: 2, y1: 2, x2: 8, y2: 8)
genericDraw(line)
```

编译器会如何处理 genericDraw 的调用呢? 我们来看一下生成的汇编指令:

```
(lldb) di -s $rip -c 10
HowGenericWorks`main:
->  0x100001cdd <+173>: mov    rdi, qword ptr [rip + 0x3e0e3c]
    0x100001ce4 <+180>: mov    r8, qword ptr [rip + 0x3e0e3d]
    0x100001ceb <+187>: mov    r9, qword ptr [rip + 0x3e0e3e]
    0x100001cf2 <+194>: mov    r10, qword ptr [rip + 0x3e0e3f]
    0x100001cf9 <+201>: mov    qword ptr [rbp - 0x40], rdi
    0x100001cfd <+205>: mov    qword ptr [rbp - 0x38], r8
    0x100001d01 <+209>: mov    qword ptr [rbp - 0x30], r9
    0x100001d05 <+213>: mov    qword ptr [rbp - 0x28], r10
    0x100001d09 <+217>: mov    rdi, rcx
    0x100001d0c <+220>: call   0x100001d80               ; HowGenericWork
s.genericDraw <A where A: HowGenericWorks.Drawable> (A) -> () at main.swif
t:7
```

然后我们把 rip 移动到 0x100001d0c，这是我们很熟悉的套路了:

```
(lldb) br s -a 0x100001d0c
Breakpoint 2: where = HowGenericWorks`main + 220 at main.swift:12, address
 = 0x0000000100001d0c
(lldb) con
Process 39411 resuming

(lldb) di -l
HowGenericWorks`main + 217 at main.swift:12
   11   let line = Line(x1: 2, y1: 2, x2: 6, y2: 6)
   12   genericDraw(line)
   13   //Line(x1: 2, y1: 2, x2: 6, y2: 6).draw()
HowGenericWorks`main:
    0x100001d09 <+217>: mov    rdi, rcx
->  0x100001d0c <+220>: call   0x100001d80               ; HowGenericWork
s.genericDraw <A where A: HowGenericWorks.Drawable> (A) -> () at main.swif
t:7
    0x100001d11 <+225>: xor    eax, eax
```

然后，来看下调用前，寄存器的环境:

```
(lldb) re r rdi rsi rdx rsp
     rdi = 0x00007fff5fbff6f0
     rsi = 0x000000010038d5a8  HowGenericWorks`type metadata for HowGeneri
cWorks.Line
     rdx = 0x000000010038d3d8  HowGenericWorks`protocol witness table for
 HowGenericWorks.Line : HowGenericWorks.Drawable in HowGenericWorks
     rsp = 0x00007fff5fbff6e0
```

其实，这里的绝大部分内容，我们都已经非常熟悉了。可以看到，在调用 genericDraw 的时候：

- rdi 是一个在栈中的临时变量，也就是 genericDraw 的参数；
- rsi 是 Line 类型的metadata，通过这个地址我们可以读取 Line 的VWT；
- rdx 是 Line 类型的PWT，我们需要通过它找到最终要调用的 draw 方法；

看到这里，你可能会想，这不就跟 protocol 的版本一样一样么？所谓的编译器多态到底是怎么体现出来的呢？别着急，如果我们再确认下 rdi 的值，你就能找到点儿感觉了：

```
(lldb) x -s8 -c4 -fx $rdi
0x7fff5fbff6f0: 0x0000000000000002 0x0000000000000002
0x7fff5fbff700: 0x0000000000000006 0x0000000000000006
```

看到了吧，实际上，传递给 genericDraw 的，并不是 Line 对象的existential container，而直接就是包含 Line 对象值的地址。也就是说 genericDraw 的参数是静态的，给它传递什么对象，它就会直接得到这个对象的值。而不像一个普通的 protocol 类型的参数，还要借用value witness table负责对象的创建和销毁。

理解了这点之后，我们执行 si 命令，进入函数内部，并执行 di -f 查看 genericDraw 的完整实现：

```
(lldb) si
(lldb) di -f
HowGenericWorks`genericDraw<A where ...> (A) -> ():
->  0x100001d80 <+0>:  push   rbp
    0x100001d81 <+1>:  mov    rbp, rsp
    0x100001d84 <+4>:  sub    rsp, 0x30
    0x100001d88 <+8>:  mov    qword ptr [rbp - 0x8], rsi
    0x100001d8c <+12>: mov    qword ptr [rbp - 0x10], rdi
    0x100001d90 <+16>: mov    qword ptr [rbp - 0x18], rdi
    0x100001d94 <+20>: mov    qword ptr [rbp - 0x20], rsi
    0x100001d98 <+24>: mov    qword ptr [rbp - 0x28], rdx
    0x100001d9c <+28>: mov    rax, qword ptr [rbp - 0x28]
    0x100001da0 <+32>: call   qword ptr [rax]
    0x100001da2 <+34>: mov    rax, qword ptr [rbp - 0x20]
    0x100001da6 <+38>: mov    rdx, qword ptr [rax - 0x8]
    0x100001daa <+42>: mov    rdi, qword ptr [rbp - 0x18]
    0x100001dae <+46>: mov    rsi, rax
    0x100001db1 <+49>: call   qword ptr [rdx + 0x20]
    0x100001db4 <+52>: add    rsp, 0x30
    0x100001db8 <+56>: pop    rbp
    0x100001db9 <+57>: ret
```

然后，我们推测， 0x100001da0 这个地址，就是在调用 Line.draw() 方法了。在这里打个断点，并执行到这里：

```
(lldb) br s -a 0x100001da0
Breakpoint 3: where = HowGenericWorks`HowGenericWorks.genericDraw <A where
 A: HowGenericWorks.Drawable> (A) -> () + 32 at main.swift:8, address = 0x
0000000100001da0
(lldb) con
Process 39411 resuming
(lldb) di -l
HowGenericWorks`HowGenericWorks.genericDraw <A where A: HowGenericWorks.Dr
awable> (A) -> () + 16 at main.swift:8
   7    func genericDraw<T: Drawable>(_ shape: T) {
   8        shape.draw()
   9    }
HowGenericWorks`genericDraw<A where ...> (A) -> ():
    0x100001d90 <+16>: mov    qword ptr [rbp - 0x18], rdi
    0x100001d94 <+20>: mov    qword ptr [rbp - 0x20], rsi
    0x100001d98 <+24>: mov    qword ptr [rbp - 0x28], rdx
    0x100001d9c <+28>: mov    rax, qword ptr [rbp - 0x28]
->  0x100001da0 <+32>: call   qword ptr [rax]
```

然后，我们先来看下调用的环境：

```
(lldb) re r rdi rsi rdx rcx rax
     rdi = 0x00007fff5fbff6f0
     rsi = 0x0000000010038d5a8  HowGenericWorks`type metadata for HowGeneri
cWorks.Line
     rdx = 0x0000000010038d3d8  HowGenericWorks`protocol witness table for
 HowGenericWorks.Line : HowGenericWorks.Drawable in HowGenericWorks
     rcx = 0x00007fff5fbff6f0
     rax = 0x0000000010038d3d8  HowGenericWorks`protocol witness table for
 HowGenericWorks.Line : HowGenericWorks.Drawable in HowGenericWorks
```

可以看到，此时 rax 是 Line 的PWT，而 Drawable 又只约束了一个方法，因此，[rax] 应该就是 Line.draw() 的地址了。

我们先来看下 rax 保存的值：

```
(lldb) x -s8 -c1 -fx $rax
0x10038d3d8: 0x0000000100002570
```

然后，再反汇编一下这个保存的地址：

```
(lldb) di -s 0x0000000100002570
HowGenericWorks`protocol witness for Drawable.draw() -> () in conformance
 Line:
    0x100002570 <+0>:  push   rbp
    0x100002571 <+1>:  mov    rbp, rsp
    0x100002574 <+4>:  sub    rsp, 0x30
    0x100002578 <+8>:  lea    rax, [rbp - 0x20]
    0x10000257c <+12>: mov    rcx, qword ptr [rdi]
```

最终，我们就通过LLDB的提示验证了一开始的推测。但在这里，有一点要说明的是，尽管函数参数没有使用existential container的方式传递，但如果我们在 draw() 内部定义 T 类型的变量，那么编译器还是会通过对应类型的VWT，在函数的栈里创建一个value buffer，规则和上一节中的定义是一样的。大家可以试着把 genericDraw 的定义改成这样：

```
func genericDraw<T: Drawable>(_ shape: T) {
    let tmp = shape
    tmp.draw()
}
```

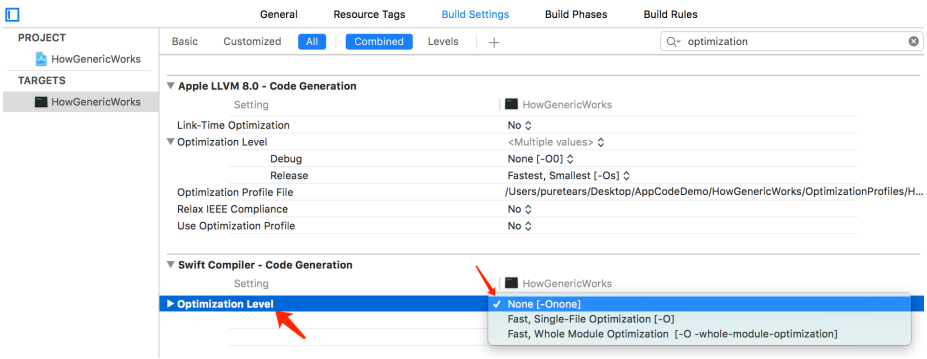
然后自己观察下 tmp 的创建过程，这里我们就不再重述了。

Protocol还是Generic?

了解了 protocol 和generic各自的工作原理，当我们再次回过头去看一开始的两个方法时，尽管它们完成相同的工作，我们应该选择哪种形式呢？一般来说，我们还是应该使用泛型的方式。因为编译器可以利用静态类型系统，尽可能对调用的泛型代码进行优化。接下来，我们就来具体感受下，Swift编译器对泛型代码的优化能力。

首先，在AppCode里，选择*File / Open Project in Xcode*，因为可能大家还是更习惯在Xcode中来进行设置。

其次，选中Target，选择Build Settings。这里，我们可以在筛选条件里输入optimization过滤一下各种选项，就可以在底部看到Swift优化级别的配置了：

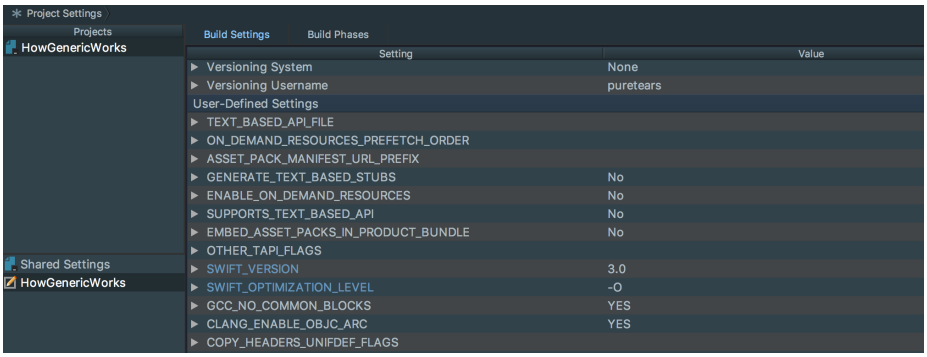


这里有三个值：

- 不优化（None）；
- 单文件优化（Single-file optimization）；
- 全模组优化（Whole module optimization）；

这是什么意思呢？我们先来看"Single-File Optimization"。选中这个配置之后，我们回到AppCode，之前我们在Xcode做的任何配置，都会直接在AppCode中生效，我们不必担心这个问题。

在AppCode里，按 Cmd + ; 打开项目设置，选中Target，同样可以在底部看到Swift优化级别的设定，可以看到，它已经被设置成了-O，这就是Sing-File Optimization的设置。



对我们定义的泛型函数 genericDraw 来说，如果在它定义的文件里可以满足下面两个条件：

- 在调用时可以推导出参数的类型；
- 可以看到推导出的类型的完整定义；

编译器就可以根据泛型函数的模板直接生成一个具象函数的版本。当然，我们的 genericDraw<T>() 满足上面两个要求，于是，只要我们打开了"Single-File Optimization"选项，Swift编译器就会为我们生成一个类似这样的函数：

```
func drawLine(_ shape: Line) {
    shape.draw()
}
```

我们实际上调用的，也将是这个具象的版本。并且，只要我们每传递一个不同类型的参数，编译器就会为我们优化出一个不同的具象函数。这也就意味着，我们不再需要传递额外的PWT/metadata，无须再使用VWT拷贝和销毁对象，这听起来简直棒极了。

但你也可能会想，等等，如果我们使用了多种不同类型的对象调用了 genericDraw<T> ，岂不是就要多出来很多不同版本的具象函数。如果 genericDraw<T> 是个复杂函数呢？我们就要为此付出一笔不小的空间成本。这样做真的值得么？

Swift的开发者当然也想到了这个问题，为此，除了把泛型函数具象化之外，编译器还可以采取进一步的优化措施。对于这样的代码：

```
let line = Line(x1: 2, y1: 2, x2: 8, y2: 8)
genericDraw(line)
```

首先，编译器可以把泛型调用替换成具象函数调用：

```
let line = Line(x1: 2, y1: 2, x2: 8, y2: 8)
drawLine(line)
```

其次，由于 drawLine 的实现很简单，编译器可以选择直接把它执行的代码inline进来：

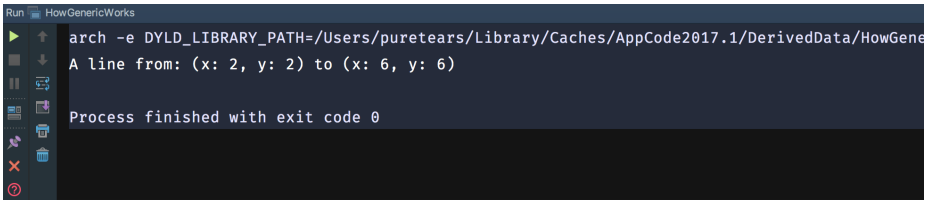
```
// Swift pesudo code
let line = Line(x1: 2, y1: 2, x2: 8, y2: 8)
line.draw()
```

最后，通过进一步代码分析，我们知道，变量 line 定义实际上是没用的。因此，上面的两行代码还可以优化成这样：

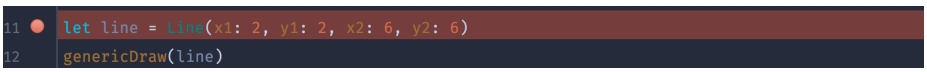
```
// Swift pesudo code
Line(x1: 2, y1: 2, x2: 8, y2: 8).draw()
```

所以，对于泛型代码的优化，理论上的确有可能增大代码的体积，但这并不一定会发生。绝大多数时候，编译器都可以为我们生成更好的代码。并且，如果上面这些优化真的发生，我们之前定义在 genericDraw(line) 上的断点就会失效了。

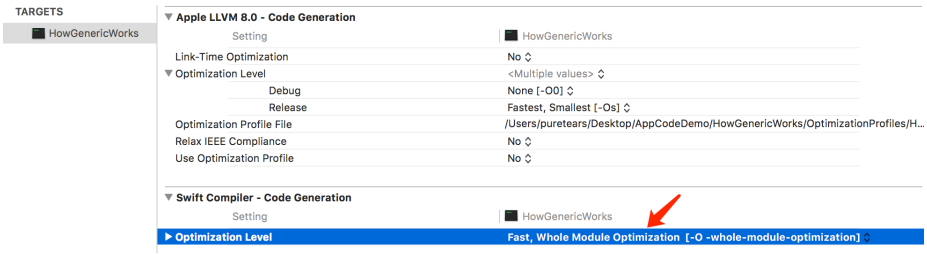
执行一下：



就可以看到，我们设置的断点并没有生效。为了能确切了解编译器的行为，我们可以把断点放在 `line` 的定义上：



并且，为了让Swift编译器在优化的时候火力全开，我们在Xcode里把优化级别设置为Whole Module Optimization：



设置完成后，回到AppCode，等配置同步完成后，我们启动LLDB，这次，应该就可以断下来了。我们执行 `di -f` 来看下生成的代码（这里，我们只截取了其中有用的部分）：

```
(lldb)  di -f
HowGenericWorks`main:
    ...
    0x100001d0c <+60>:  mov    qword ptr [rax], rbx
->  0x100001d0f <+63>:  movaps xmm0, xmmword ptr [rip + 0x31103a]
    0x100001d16 <+70>:  movaps xmmword ptr [rip + 0x3b66e3], xmm0 ; HowGen
ericWorks.line : HowGenericWorks.Line

    0x100001d1d <+77>:  movaps xmm1, xmmword ptr [rip + 0x31103c]
    0x100001d24 <+84>:  movaps xmmword ptr [rip + 0x3b66e5], xmm1 ; HowGen
ericWorks.line : HowGenericWorks.Line + 16

    0x100001d2b <+91>:  movaps xmmword ptr [rbp - 0x30], xmm0
    0x100001d2f <+95>:  movaps xmmword ptr [rbp - 0x20], xmm1

    0x100001d33 <+99>:  lea    rdi, [rbp - 0x30]
    0x100001d37 <+103>: call   0x100001eb0               ; HowGenericWorks
.Line.draw () -> () at Shape.swift
    0x100001d3c <+108>: xor    eax, eax
```

其中，从下面4条指令行可以看到：

```
0x100001d0f <+63>:  movaps xmm0, xmmword ptr [rip + 0x31103a]
0x100001d16 <+70>:  movaps xmmword ptr [rip + 0x3b66e3], xmm0

0x100001d1d <+77>:  movaps xmm1, xmmword ptr [rip + 0x31103c]
0x100001d24 <+84>:  movaps xmmword ptr [rip + 0x3b66e5], xmm1
```

编译器使用了 `xmm0` 和 `xmm1` 这两个128位寄存器参与了 `Line` 对象的创建。它们执行过后， `xmm0` 和 `xmm1` 寄存器里，就应该是一个完整的 `Line` 对象的值了。

接下来，编译器把 `xmm0` 和 `xmm1` 的结果保存在了 `[rbp-0x30]` 这个位置：

```
0x100001d2b <+91>:  movaps xmmword ptr [rbp - 0x30], xmm0
0x100001d2f <+95>:  movaps xmmword ptr [rbp - 0x20], xmm1
```

这是 `main` 函数的一个局部变量，我们可以把它理解为就是一个 `Line` 的临时对象；。

最后，编译器把这个临时对象的地址保存在了 `rdi` 寄存器，它作为 `0x100001eb0` 这个函数的第一个参数：

```
0x100001d33 <+99>:  lea    rdi, [rbp - 0x30]
0x100001d37 <+103>: call   0x100001eb0               ; HowGenericWorks.Lin
e.draw () -> () at Shape.swift
```

从LLDB的提示中就可以看到， `0x100001eb0` 是 `Line.draw()` 方法的地址。也就是说，编译器直接优化掉了 `genericDraw` 的调用，而把它要执行的代码用direct dispatch的方式inline在了这里。

最后，我们执行三次 `si` 命令，来看下 `xmm` 寄存器的值，来确认下我们之前的所有推测：

```
(lldb) si
(lldb) si
(lldb) si
(lldb) re r xmm0 xmm1
    xmm0 = {0x02 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x02 0x00 0x00 0x00 0x
00 0x00 0x00 0x00}
    xmm1 = {0x06 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x06 0x00 0x00 0x00 0x
00 0x00 0x00 0x00}
```

可以看到，此时 `xmm0` 和 `xmm1` 寄存器的值，的确就是我们传递的 `line` 对象的两个值。

Conclusion

看到这里，你应该对Swift编译器如何处理泛型代码有一个比较具体的认识了，并且，对于所谓的编译期多态，也应该有了更深刻的理解。通过最终优化过的代码我们可以看到，泛型代码最后没有任何运行时动态处理的迹象，编译器生成的代码几乎和面向过程编程是同等高效的。

至此，无论是面向对象、面向 `protocol` ，还是泛型编程，在这些编程范式背后隐藏的各种成本和利益交换已经清晰呈现在我们眼前了。在下一章，我们将开始讨论另外一大类应用场景，如何在Swift中正确的处理各种错误。

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

返回视频 ⏭
(/series/protocol-and-generic)