**用 debugger 学习 golang**

常见的工程语言可分为解释型和编译型两种，比如写 php 的，一般就不怎么在乎 debugger 之类的东西。为什么？如果真出了问题，我可以临时把出问题的服务机器从线上服务中摘除出来，甚至申请一个较高的权限去修改代码，然后到处去 die/echo。虽然有人说这么做不太好，或者一般公司也不给开权限。不过着急的时候，这个肯定是可行的。然而像 java/go 这种编译型的就比较麻烦了。线上一般只有程序的运行环境而没有编译环境。就算是在线下，每次去加一行 fmt.Println 或者 System.out.println 都去编译一遍代码也是会明显降低幸福感的事情(当然这里有人说现在 java 支持 hotswap 之类的功能，不过你总还是会遇到需要重新编译的场景。go 也是一样的，项目大了，编译时间还是可能会有个五六七八秒的。想要迅速地还原 bug 的现场，那还是能用 debugger 为上。

除了拿 debugger 来 debug。还可以用 debugger 来了解了解程序运行的机制，或者用 disass 来查看程序运行的汇编码。这一点也很重要。应用层的语言很多时候因为 runtime 事无巨细的封装，已经不是所见即所得的东西了，特别是像 go 这样，你写一个 var a = 1 却连最终这个变量会被分配到堆上还是栈上都不知道。而像应用层的空 interface 和非空的 interface 实际的数据结构完全不一样，这些如果你想知道的话一方面可以通过阅读源代码，但 go 的源代码到你的代码之间始终还是有一个转换过程。如果你可以通过汇编直接查看运行时的结构显然要更为直观。

这篇文章也不准备写得大而全，就简单地举一些可以靠 debugger 来帮我们更清楚地认识问题的场景吧。

**var a = new(T) 和 var a = &T{} 这两种语法有区别么？**

写两个差不多的程序，然后带上 gcflags="-N -l" 来 go build

-> 5 func main() {

di`main.main:

-> 0x104f400 <+0>: sub rsp, 0x28

0x104f404 <+4>: mov qword ptr [rsp + 0x20], rbp

0x104f409 <+9>: lea rbp, [rsp + 0x20]

\*\* 6 var a = &T{}

0x104f40e <+14>: mov qword ptr [rsp], 0x0

0x104f416 <+22>: lea rax, [rsp]

0x104f41a <+26>: mov qword ptr [rsp + 0x18], rax

0x104f41f <+31>: test al, byte ptr [rax]

0x104f421 <+33>: mov qword ptr [rsp], 0x0

0x104f429 <+41>: mov rax, qword ptr [rsp + 0x18]

0x104f42e <+46>: mov qword ptr [rsp + 0x10], rax

\*\* 7 a.age += 1

0x104f433 <+51>: test al, byte ptr [rax]

0x104f435 <+53>: mov rax, qword ptr [rax]

0x104f438 <+56>: mov qword ptr [rsp + 0x8], rax

0x104f43d <+61>: mov rcx, qword ptr [rsp + 0x10]

0x104f442 <+66>: test al, byte ptr [rcx]

0x104f444 <+68>: inc rax

0x104f447 <+71>: mov qword ptr [rcx], rax

-> 5 func main() {

di2`main.main:

-> 0x104f400 <+0>: sub rsp, 0x20

0x104f404 <+4>: mov qword ptr [rsp + 0x18], rbp

0x104f409 <+9>: lea rbp, [rsp + 0x18]

\*\* 6 var a = new(T)

0x104f40e <+14>: mov qword ptr [rsp], 0x0

0x104f416 <+22>: lea rax, [rsp]

0x104f41a <+26>: mov qword ptr [rsp + 0x10], rax

\*\* 7 a.age += 1

0x104f41f <+31>: test al, byte ptr [rax]

0x104f421 <+33>: mov rax, qword ptr [rsp]

0x104f425 <+37>: mov qword ptr [rsp + 0x8], rax

0x104f42a <+42>: mov rcx, qword ptr [rsp + 0x10]

0x104f42f <+47>: test al, byte ptr [rcx]

0x104f431 <+49>: inc rax

0x104f434 <+52>: mov qword ptr [rcx], rax

两种代码反编译出来的汇编不一致，可以看到第一种比第二种多要了 8 个字节的栈空间。可以猜测实际上第一种写法是分两部走：

1. T{}；2.& 取地址

go build 不带 gcflags 参数时，两者出来的汇编代码就是完全一致的了。感兴趣的同学可以自行验证。

**查看 go 的 interface 的数据结构**

go 的 interface 一直是一个比较让人纠结的数据结构，官方和信徒们从 14 年就一直在花不少篇幅跟你讲，怎么判断 interface 和 nil，我们这个设计是这样的 blabla。不过我始终觉得 go 的 interface 设计是有点问题的，只不过这帮 unix 老古董们不想承认。。。

先来看一些例子吧：

package main

import (

"bytes"

"fmt"

"io"

)

var (

a \*bytes.Buffer = nil

b io.Writer

)

func set(v \*bytes.Buffer) {

if v == nil {

fmt.Println("v is nil")

}

b = v

}

func get() {

if b == nil {

fmt.Println("b is nil")

} else {

fmt.Println("b is not nil")

}

}

func main() {

set(nil)

get()

}

例子二(来自公司同事)：

package main

import (

"fmt"

"io"

"os"

"unsafe"

)

var (

v interface{}

r io.Reader

f \*os.File

fn os.File

)

func main() {

fmt.Println(v == nil)

fmt.Println(r == nil)

fmt.Println(f == nil)

v = r

fmt.Println(v == nil)

v = fn

fmt.Println(v == nil)

v = f

fmt.Println(v == nil)

r = f

fmt.Println(r == nil)

}

可以自己运行一下看看结果。有很多文章会讲，interface 包含有 type 和 data 两个元素，只有两者均为 nil 的时候才是真的 nil，然后再给你灌输了很多理由为什么要这么设计。甚至还援引了 Rob Pike 的某个 ppt。

对设计的吐槽先打住，我们看看 interface 在运行期到底是一个什么样的东西：

(lldb) p v

(interface {}) main.v = {

\_type = 0x0000000000000000

data = 0x0000000000000000

}

(lldb) p r

(io.Reader) main.r = {

tab = 0x0000000000000000

data = 0x0000000000000000

}

(lldb) p f

(\*os.File) main.f = 0x0000000000000000

这里可以看到，在 golang 中空 interface 和非空 interface 在数据结构上也是有差别的。空 interface 就只有 runtime.\_type 和 void *指针组成。而非空 interface 则是 runtime.itab 和 void* 指针组成。

把 \*os.File 分别赋值给空 interface 和 io.Reader 类型的接口变量之后。我们看看这个 runtime.\_type 和 runtime.itab 都变成什么样了：

(lldb) p v

(interface {}) main.v = {

\_type = 0x00000000010be0a0

data = 0x0000000000000000

}

(lldb) p \*r.tab

(runtime.itab) \*tab = {

inter = 0x00000000010ad520

\_type = 0x00000000010be0a0

link = 0x0000000000000000

hash = 871609668

bad = false

inhash = true

unused = ([0] = 0, [1] = 0)

fun = ([0] = 0x000000000106d610)

}

非空 interface 的 \_type 是存储在 tab 字段里了。除此之外，非空 interface 本身的类型(这里是 io.Reader)存储在 inter 字段中：

(runtime.interfacetype) \*inter = {

typ = {

size = 0x0000000000000010

ptrdata = 0x0000000000000010

hash = 3769182245

tflag = 7

align = 8

fieldalign = 8

kind = 20

alg = 0x000000000113cd80

gcdata = 0x00000000010d55f6

str = 12137

ptrToThis = 45152

}

pkgpath = {

bytes = 0x0000000001094538

}

mhdr = (len 1, cap 1) {

[0] = (name = 1236, ityp = 90528)

}

}

此外，非空 interface 还会在 itab 的 fun 数组里存储函数列表。

这里会有一个非常蛋疼的地方，如果你把一个非空 interface 类型的 nil 值的 interface 变量赋值给一个空 interface 类型的变量，那么就会得到一个非空类型的非空 interface 变量。

这绝对是 go 的设计缺陷。。。

现在为了避免判断时候的失误，也有人会用 reflect.ValueOf(v) 来判断一个 interface 是否为 nil。但也会比较别扭。

**学习 go 的 channel**

来一个简单的 demo：

package main

func main() {

var a = make(chan int, 4)

a <- 1

a <- 1

a <- 1

a <- 1

close(a)

println()

}

打上断点，查看 a 的结构：

\* thread #1, stop reason = step over

frame #0: 0x000000000104c354 normal\_example`main.main at normal\_example.go:5

2

3 func main() {

4 var a = make(chan int, 4)

-> 5 a <- 1

6 a <- 1

7 a <- 1

8 a <- 1

Target 0: (normal\_example) stopped.

(lldb) p a

(chan int) a = 0x000000c42007a000

(lldb) p \*a

(hchan<int>) \*a = {

qcount = 0

dataqsiz = 4

buf = 0x000000c42007a060

elemsize = 8

closed = 0

elemtype = 0x0000000001055ee0

sendx = 0

recvx = 0

recvq = {

first = 0x0000000000000000

last = 0x0000000000000000

}

sendq = {

first = 0x0000000000000000

last = 0x0000000000000000

}

lock = (key = 0x0000000000000000)

}

a.buf 是 void\* 类型，类似 c/c艹，这种类型需要用 x 指令来读取内容：

(lldb) n

Process 21186 stopped

\* thread #1, stop reason = step over

frame #0: 0x000000000104c369 normal\_example`main.main at normal\_example.go:6

3 func main() {

4 var a = make(chan int, 4)

5 a <- 1

-> 6 a <- 1

7 a <- 1

8 a <- 1

9 close(a)

Target 0: (normal\_example) stopped.

(lldb) p a.buf

(void \*) buf = 0x000000c42007a060

(lldb) x a.buf

0xc42007a060: 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................

0xc42007a070: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................

可以看到向 channel 中写入一个 1 之后，a.buf 中的内容发生了变化。同时，a 中的 sendx 和 qcount 也都发生了变化：

(lldb) p \*a

(hchan<int>) \*a = {

qcount = 1 // 这里这里

dataqsiz = 4

buf = 0x000000c42007a060

elemsize = 8

closed = 0

elemtype = 0x0000000001055ee0

sendx = 1 // 这里这里

recvx = 0

recvq = {

first = 0x0000000000000000

last = 0x0000000000000000

}

sendq = {

first = 0x0000000000000000

last = 0x0000000000000000

}

lock = (key = 0x0000000000000000)

}

这样就可以非常方便地结合代码，观察 channel 的发送和接收行为。其实从 debugger 里得到的信息都非常的直观，比看图表要直观得多。比如这里我们可以直接看到 lock 字段。这也说明 channel 本身为了并发安全是带锁的。

recvq 和 sendq 是用来维护发送接收时被阻塞需要休眠的 goroutine 列表。

elemtype 是 runtime.\_type 类型，可以看到 channel 中的元素类型信息。

close(a) 以后再看看结构：

(chan int) a = 0x000000c42007a000

(lldb) p \*a

(hchan<int>) \*a = {

qcount = 4

dataqsiz = 4

buf = 0x000000c42007a060

elemsize = 8

closed = 1 // 重点在这里

elemtype = 0x0000000001055ee0

sendx = 0

recvx = 0

recvq = {

first = 0x0000000000000000

last = 0x0000000000000000

}

sendq = {

first = 0x0000000000000000

last = 0x0000000000000000

}

lock = (key = 0x0000000000000000)

}

比画一堆图不知道高到哪里去了。

再尝试在 a 上阻塞几个 goroutine：

(lldb) p a.recvq

(waitq<int>) recvq = {

first = 0x000000c42007c000

last = 0x000000c42007c060

}

(lldb) p a.recvq.first

(\*sudog<int>) first = 0x000000c42007c000

(lldb) p \*a.recvq.first

(sudog<int>) \*first = {

g = 0x000000c420000f00

isSelect = false

next = 0x000000c42007c060

prev = 0x0000000000000000

elem = 0x0000000000000000

acquiretime = 0

releasetime = 0

ticket = 0

parent = 0x0000000000000000

waitlink = 0x0000000000000000

waittail = 0x0000000000000000

c = 0x000000c42007a000

}

可以看到，channel 的 recvq 和 sendq 就是个 sudog 的双向链表，没有什么难理解的~

**确认 panic 的现场**

程序里有时候会有这种代码：

someFunction(r.A, \*r.B, \*r.C, \*r.D, r.E, \*r.F)

然后在这里 panic 了。但是 go 只会告诉你 nil pointer deference，却不会告诉你是哪个 nil pointer deference。着实蛋疼。

这个就是用 debugger 最基本断点功能了。如果是用 delve，断点可以用很多种方法来设置，比如 function+行号，文件名+行号，如果有歧义，delve 也会告诉你具体要怎么来消除歧义。

(lldb) n

Process 22595 stopped

\* thread #1, stop reason = step over

frame #0: 0x000000000104c344 nilPointer`main.main at nilPointer.go:16

13 }

14

15 func main() {

-> 16 var t = T{A: 1}

17 test(t.A, \*t.B, \*t.C, \*t.D, t.E, \*t.F)

18 }

Target 0: (nilPointer) stopped.

(lldb) n

Process 22595 stopped

\* thread #1, stop reason = step over

frame #0: 0x000000000104c365 nilPointer`main.main at nilPointer.go:17

14

15 func main() {

16 var t = T{A: 1}

-> 17 test(t.A, \*t.B, \*t.C, \*t.D, t.E, \*t.F)

18 }

Target 0: (nilPointer) stopped.

(lldb) p t

(main.T) t = {

A = 1

B = 0x0000000000000000

C = 0x0000000000000000

D = 0x0000000000000000

E = 0

F = 0x0000000000000000

}

哪里是 nil 一目了然~

**string 和 byte 之间到底有没有进行相互转换**

例子：

package main

func main() {

var str = "abcde"

var b = []byte("defg")

println(str)

println(string(b))

}

还是看反编译的结果：

\*\* 6 var b = []byte("defg")

7

0x104cf17 <+71>: lea rax, [rsp + 0x30]

0x104cf1c <+76>: mov qword ptr [rsp], rax

0x104cf20 <+80>: lea rax, [rip + 0x1c95b] ; go.string.\* + 210

0x104cf27 <+87>: mov qword ptr [rsp + 0x8], rax

0x104cf2c <+92>: mov qword ptr [rsp + 0x10], 0x4

0x104cf35 <+101>: call 0x1038390 ; runtime.stringtoslicebyte at string.go:146

0x104cf3a <+106>: mov rax, qword ptr [rsp + 0x20]

0x104cf3f <+111>: mov rcx, qword ptr [rsp + 0x18]

0x104cf44 <+116>: mov rdx, qword ptr [rsp + 0x28]

0x104cf49 <+121>: mov qword ptr [rsp + 0xa0], rcx

0x104cf51 <+129>: mov qword ptr [rsp + 0xa8], rax

0x104cf59 <+137>: mov qword ptr [rsp + 0xb0], rdx

重点在这里的

0x104cf35 <+101>: call 0x1038390 ; runtime.stringtoslicebyte at string.go:146

runtime 里还有一个对应的：

0x104c624 <+196>: call 0x10378c0 ; runtime.slicebytetostring at string.go:72

有了这样的手段，如果别人和你说 go 会优化 string 和 []byte 之间的转换。你就可以随时掏出 debugger 来打他的脸了。

**我程序的 select 到底被翻译成什么样的执行过程了**

select 是 golang 提供的一种特权语法，实现的功能比较神奇。先不说行为怎么样。这种特权语法实际上最终一定会被翻译成某种汇编指令或者 runtime 的内置函数。

用反汇编来看一眼。

-> 6 select {

-> 0x104e3d5 <+117>: mov qword ptr [rsp + 0x38], 0x0

0x104e3de <+126>: lea rdi, [rsp + 0x40]

0x104e3e3 <+131>: xorps xmm0, xmm0

0x104e3e6 <+134>: lea rdi, [rdi - 0x10]

0x104e3ea <+138>: mov qword ptr [rsp - 0x10], rbp

0x104e3ef <+143>: lea rbp, [rsp - 0x10]

0x104e3f4 <+148>: call 0x1048d5a ; runtime.duffzero + 250 at duff\_amd64.s:87

0x104e3f9 <+153>: mov rbp, qword ptr [rbp]

0x104e3fd <+157>: lea rax, [rsp + 0x38]

0x104e402 <+162>: mov qword ptr [rsp], rax

0x104e406 <+166>: mov qword ptr [rsp + 0x8], 0xb8

0x104e40f <+175>: mov dword ptr [rsp + 0x10], 0x3

0x104e417 <+183>: call 0x10305d0 ; runtime.newselect at select.go:60

\*\* 6 select {

0x104e425 <+197>: mov rax, qword ptr [rsp + 0x30]

\*\* 6 select {

0x104e445 <+229>: mov rax, qword ptr [rsp + 0x28]

\*\* 6 select {

0x104e46a <+266>: lea rax, [rsp + 0x38]

0x104e46f <+271>: mov qword ptr [rsp], rax

0x104e473 <+275>: call 0x1030b10 ; runtime.selectgo at select.go:202

0x104e478 <+280>: mov rax, qword ptr [rsp + 0x8]

0x104e47d <+285>: mov qword ptr [rsp + 0x20], rax

看起来 select 被翻译成了多段汇编代码。说明这个函数稍微复杂一些，不过反汇编过程已经帮我们定位到了 select 被翻译成的函数的位置。

实际上 select 的执行过程为： newselect->selectsend/selectrecv->selectgo 这几个过程。如果你的程序是下面这样的：

for {

select {

case <-ch:

case ch2<-1:

default:

}

}

在每次进入 for 循环的时候，runtime 里的 hselect 结构都会重新创建。也就是说写一个有 default case 的无限循环，不仅仅是你知道的 cpu 占用爆炸，实际上还在不断地在堆上分配、释放、分配、释放空间。感觉这里官方应该是可以做一些优化的，不知道为什么逻辑这么原始。(当然，在 go 语言学习笔记里看到雨痕老师也吐槽他们的代码写得渣哈哈哈。

**正在运行的 goroutine 到底是阻塞在什么地方了**

golang 中常见的内存泄露套路是这样的：

func main() {

var ch chan int

go func() {

select {

case <-ch:

}

}()

}

监听了一个永远阻塞的 channel，或者向一个没有接收方的 channel 发数据，如果这些事情没有发生在主 goroutine 里的话，在 runtime 的 checkdead 函数中不会认为这是个 deadlock。而这样的 goroutine 创建过程往往在 for 循环里。

公司内的某个程序就曾经在线下 debug 的时候发现每次来一个请求，就会导致 goroutine 总数 +1。这显然是不正常的。在 goroutine 达到一定数量之后，可以适用 delve attach 到你的进程，然后运行：

goroutines

一下就看到你泄露的 goroutine 都是卡在什么地方了。

当然，如果你的程序开了 pprof，那通过网页来看倒是更为方便。

之前公司内的某个库在找不到 disf 的 ip 的时候就会阻塞在 lib 的 channel 上。用这个办法可以非常快的找到问题根结。不用像某些程序员一样到处加 fmt.Println 了。

**程序的 cpu 占用非常高，似乎在哪里有死循环**

这个问题有两个工具可以用，一个是 perf，一个是 debugger。

sudo perf top

可以找到死循环所处的位置，这个在之前写的文章中有过涉及了。这里就不再赘述。

还有一种死循环，但是程序本身没死掉的，那就可以直接用 dlv attach 进去了，基本上切换至可疑的 goroutine，跟个十几步就可以找到问题所在，当然，结合 perf 来看更高效。这个可以参考之前定位 jsoniter 时候的步骤：<https://github.com/gin-gonic/gin/issues/1086>。

**怎么一直观察某一个变量的变化过程**

也很简单，在希望观察的地方打上断点，如果断点 id 是 13，那么用 delve 的 on 命令：

on 13 print xxx

即可

(dlv) n

> main.main() ./for.go:6 (hits goroutine(1):11 total:11) (PC: 0x44d694)

count: 45

1: package main

2:

3: func main() {

4: count:=0

5: for i:=0;i<10000;i++ {

=> 6: count+=i

7: }

8: println(count)

9: }

(dlv) n

**我的程序只有运行到 for 循环的第 1000 次叠代的时候才会出 bug，我怎么在第 1000 次循环的时候才设置这个断点**

用 delve 很简单：

ubuntu@ubuntu-xenial:~$ dlv exec ./for

Type 'help' for list of commands.

(dlv) b for.go:6

Breakpoint 1 set at 0x44d694 for main.main() ./for.go:6

(dlv) cond 1 i==1000 ////// => 重点在这里

(dlv) r

Process restarted with PID 29024

(dlv) c

> main.main() ./for.go:6 (hits goroutine(1):1 total:1) (PC: 0x44d694)

1: package main

2:

3: func main() {

4: count:=0

5: for i:=0;i<10000;i++ {

=> 6: count+=i

7: }

8: println(count)

9: }

(dlv) p i

1000

(dlv) p count

499500