演示如何使用 GDB 动态分析 go 运行状态

样本: 将空指针转换为接口，看看内部到底发生了什么?

```
$ uname -a
Linux ubuntu 3.8.0-19-generic x86_64 GNU/Linux

$ go version
go version go1.2 linux/amd64
```

1. 编写代码，确保能跑起来。

```
package main

type Worker interface {
    A()
    B()
}

type Work struct {}

func (*Work) A() { println("A") }
func (*Work) B() { println("B") }
func (*Work) C() { println("C") }

func main() {
    var w Worker = (*Work)(nil)
    w.A()
    w.B()
}
```

注意禁用编译优化。运行正常。

```
$ go build -gcflags "-N -l"
$ ./test
A
B
```

2. 启动 gdb，某些时候需要使用 sudo。 注意载入 Go Runtime support。(我设置了 ~/.gdbinit 自动载入)

```
$ sudo gdb test
GNU gdb (GDB) 7.5.91.20130417-cvs-ubuntu

Reading symbols from /home/yuhen/go/src/test/test...done.
Loading Go Runtime support.
```

3. 在 main.main 里设置断点。

```
(gdb) l main.main
9
10      func (*Work) A() { println("A") }
11      func (*Work) B() { println("B") }
12      func (*Work) C() { println("C") }
13
14      func main() {
15          var w Worker = (*Work)(nil)
16          w.A()
17          w.B()
18      }

(gdb) b 16
Breakpoint 1 at 0x400d06: file /home/yuhen/go/src/test/main.go, line 16.
```

4. 启动，直到在断点停下。查看一下 locals 信息，尤其是 w。

```
(gdb) r
Starting program: /home/yuhen/go/src/test/test

Breakpoint 1, main.main () at /home/yuhen/go/src/test/main.go:16
16          w.A()

(gdb) info locals
w = {tab = 0x7ffff7fcde80, data = 0x0}
```

5. 打开 go/src/pkg/runtime 下 runtime.h 头文件。找到接口结构定义。可以看到 Iface 两个指针成员，其中 Itab 记录了接口

相关类型信息，而 data 就是实际的目标对象，也就是本例中的 (*Work)(nil)。

```
178 struct Iface
179 {
180     Itab*   tab;
181     void*   data;
182 };
```

继续看 Itab 的定义。

```
447 // layout of Itab known to compilers
448 // allocated in non-garbage-collected memory
449 struct  Itab
450 {
451     InterfaceType*  inter;
452     Type*   type;
453     Itab*   link;
454     int32   bad;
455     int32   unused;
456     void    (*fun[])(void);
457 };
```

Itab 结构中，我们关心的只有 inter 接口类型、type 原对象类型、fun 方法指针数组。

InterfaceType 和 Type 在 type.h 中定义。

Type 中需要注意的是 string，它保存了类型名称。按 8 字节对齐，数一下是第 5 个 8 字节，先记下来。

```
21 struct Type
22 {
23     uintptr size;
24     uint32 hash;
25     uint8 _unused;
26     uint8 align;
27     uint8 fieldAlign;
28     uint8 kind;
```

```
29      Alg *alg;
30      void *gc;
31      String *string;
32      UncommonType *x;
33      Type *ptrto;
34  };
```

InterfaceType 算是从 Type“继承”，其中 mhdr 记录了接口方法数量。连 Type 算在内，mhdr 位置是 8。

```
61  struct InterfaceType
62  {
63      Type;
64      Slice mhdr;
65      IMethod m[];
66  };
```

好了，回到 gdb。

6. 输出 w.itab 数据。

```
(gdb) x/7xg w.tab
0x7ffff7fcde80:     0x0000000000421700  0x0000000000422420
0x7ffff7fcde90:     0x0000000000000000  0x0000000000000000
0x7ffff7fcdea0:     0x0000000000400c00  0x0000000000400c40
0x7ffff7fcdeb0:     0x0000000000000000
```

7. 第一个数据就应该是 itab.inter，我们输出它的名称看看。

```
(gdb) x/5xg 0x0000000000421700
0x421700:    0x0000000000000010  0x140808000bb06b0d
0x421710:    0x00000000004589e0  0x000000000041d8c0
0x421720:    0x0000000000426f20                          // 前面数过，是第 5 个。

(gdb) info symbol 0x0000000000426f20                     // 看看这个地址对应的符号信息。
string.* + 8048 in section .rodata of /home/yuhen/go/src/test/test

(gdb) x/2xg 0x0000000000426f20                           // 还记得 Go string 的结构吧。
0x426f20:    0x0000000000426f30  0x000000000000000b

(gdb) x/s 0x0000000000426f30                             // 输出字符串。
```

```
0x426f30:    "main.Worker"
```

另外一个数据是 mdhr，一个 go slice 结构。位置 8，长度 3。

```
(gdb) x/10xg 0x0000000000421700
0x421700:    0x0000000000000010  0x140808000bb06b0d
0x421710:    0x00000000004589e0  0x000000000041d8c0
0x421720:    0x0000000000426f20  0x0000000000421780
0x421730:    0x000000000041ea00  0x0000000000421750
0x421740:    0x0000000000000002  0x0000000000000002
```

也就是说该接口一共 2 个方法定义。

8. 再看看 itab.type 原对象类型信息。

```
(gdb) x/7xg w.tab
0x7ffff7fcde80:     0x0000000000421700  0x0000000000422420
0x7ffff7fcde90:     0x0000000000000000  0x0000000000000000
0x7ffff7fcdea0:     0x0000000000400c00  0x0000000000400c40
0x7ffff7fcdeb0:     0x0000000000000000

(gdb) x/5xg 0x0000000000422420                          // 一样找第 5 个位置的 name
0x422420:    0x0000000000000008  0x160808004d5e5cd1
0x422430:    0x00000000004588a0  0x000000000041d480
0x422440:    0x0000000000426540

(gdb) x/2xg 0x0000000000426540                          // string { pointer, len }
0x426540:    0x0000000000426550  0x000000000000000a

(gdb) x/s 0x0000000000426550
0x426550:    "*main.Work"
```

9. 现在可以确定 itab 引用了接口类型和原类型，那么方法调用呢？归根结底，就是要在接口对象中找到 *Work 的方法 A 和 B，但不能有 C。

在 itab 结构定义中，func 位置是 5，弹性成员。长度从 InterfaceType.mdhr 中已经获取是 2。

```
(gdb) x/8xg w.tab
0x7ffff7fcde80:     0x0000000000421700  0x0000000000422420
0x7ffff7fcde90:     0x0000000000000000  0x0000000000000000
0x7ffff7fcdea0:     0x0000000000400c00  0x0000000000400c40
0x7ffff7fcdeb0:     0x0000000000000000  0x0000000000000000

(gdb) info symbol 0x0000000000400c00
main.(*Work).A in section .text of /home/yuhen/go/src/test/test

(gdb) info symbol 0x0000000000400c40
main.(*Work).B in section .text of /home/yuhen/go/src/test/test
```

至此，我们已经知道接口对象里面到底都有什么。就算是用 (*Work)(nil) 转换，接口对象内部也会保存相应的 itab，其中有 Worker 类型、*Work类型，以及 *Work 对应的方法地址。

10. 这也就能解释下面这种让初学者莫名其妙的结果。

```
14 func main() {
15     var a Worker = (*Work)(nil)
16     var b Worker = nil
17
18     println(a == nil, b == nil)
19 }
```

输出: false, true

重新架起 gdb，看看 itab 就知道了。

```
(gdb) r
Starting program: /home/yuhen/go/src/test/test

Breakpoint 1, main.main () at /home/yuhen/go/src/test/main.go:18
18         println(a == nil, b == nil)

(gdb) info locals
```

```
a = {tab = 0x7ffff7fcde80, data = 0x0}
b = {tab = 0x0, data = 0x0}
```

作用呢？反射需要？还有其他用途...

有关接口的更多内容，可阅读 iface.c 。