plan9 assembly 完全解析

众所周知，Go 使用了 Unix 老古董(误 们发明的 plan9 汇编。就算你对 x86 汇编有所了解，在 plan9 里还是有些许区别。说不定你在看代码的时候，偶然发现代码里的 SP 看起来是 SP，但它实际上不是 SP 的时候就抓狂了哈哈哈。

本文将对 plan9 汇编进行全面的介绍，同时解答你在接触 plan9 汇编时可能遇到的大部分问题。

本文所使用的平台是 linux amd64，因为不同的平台指令集和寄存器都不一样，所以没有办法共同探讨。这也是由汇编本身的性质决定的。

基本指令

栈调整

intel 或 AT&T 汇编提供了 push 和 pop 指令族，plan9 中没有 push 和 pop，栈的调整是通过对硬件 SP 寄存器进行运算来实现的，例如:

```
SUBQ $0x18, SP // 对 SP 做减法，为函数分配函数栈帧
...            // 省略无用代码
ADDQ $0x18, SP // 对 SP 做加法，清除函数栈帧
```

通用的指令和 X64 平台差不多，下面分节详述。

数据搬运

常数在 plan9 汇编用 $num 表示，可以为负数，默认情况下为十进制。可以用 $0x123 的形式来表示十六进制数。

```
MOVB $1, DI      // 1 byte
MOVW $0x10, BX   // 2 bytes
MOVD $1, DX      // 4 bytes
MOVQ $-10, AX     // 8 bytes
```

可以看到，搬运的长度是由 MOV 的后缀决定的，这一点与 intel 汇编稍有不同，看看类似的 X64 汇编:

```
mov rax, 0x1   // 8 bytes
mov eax, 0x100 // 4 bytes
mov ax, 0x22   // 2 bytes
mov ah, 0x33   // 1 byte
mov al, 0x44   // 1 byte
```

plan9 的汇编的操作数的方向是和 intel 汇编相反的，与 AT&T 类似。

```
MOVQ $0x10, AX ===== mov rax, 0x10
       |    |------------|      |
       |-------------------------|
```

不过凡事总有例外，如果想了解这种意外，可以参见参考资料中的 [1]。

常见计算指令

```
ADDQ  AX, BX   // BX += AX
SUBQ  AX, BX   // BX -= AX
IMULQ AX, BX   // BX *= AX
```

类似数据搬运指令，同样可以通过修改指令的后缀来对应不同长度的操作数。例如 ADDQ/ADDW/ADDL/ADDB。

条件跳转/无条件跳转

```
// 无条件跳转
JMP addr   // 跳转到地址，地址可为代码中的地址，不过实际上手写不会出现这种东西
JMP label  // 跳转到标签，可以跳转到同一函数内的标签位置
JMP 2(PC)  // 以当前指令为基础，向前/后跳转 x 行
JMP -2(PC) // 同上

// 有条件跳转
JNZ target // 如果 zero flag 被 set 过，则跳转
```

指令集

可以参考源代码的 arch 部分。

额外提一句，Go 1.10 添加了大量的 SIMD 指令支持，所以在该版本以上的话，不像之前写那样痛苦了，也就是不用人肉填 byte 了。

寄存器

通用寄存器

amd64 的通用寄存器:

```
(lldb) reg read
General Purpose Registers:
       rax = 0x0000000000000005
       rbx = 0x000000c420088000
       rcx = 0x0000000000000000
       rdx = 0x0000000000000000
       rdi = 0x000000c420088008
       rsi = 0x0000000000000000
```

```
    rbp = 0x000000c420047f78
    rsp = 0x000000c420047ed8
     r8 = 0x0000000000000004
     r9 = 0x0000000000000000
    r10 = 0x000000c420020001
    r11 = 0x0000000000000202
    r12 = 0x0000000000000000
    r13 = 0x00000000000000f1
    r14 = 0x0000000000000011
    r15 = 0x0000000000000001
    rip = 0x000000000108ef85  int`main.main + 213 at int.go:19
rflags = 0x0000000000000212
    cs = 0x000000000000002b
    fs = 0x0000000000000000
    gs = 0x0000000000000000
```

在 plan9 汇编里都是可以使用的，应用代码层面会用到的通用寄存器主要是: rax, rbx, rcx, rdx, rdi, rsi, r8~r15 这 14 个寄存器，虽然 rbp 和 rsp 也可以用，不过 bp 和 sp 会被用来管理栈顶和栈底，最好不要拿来进行运算。

plan9 中使用寄存器不需要带 r 或 e 的前缀，例如 rax，只要写 AX 即可:

```
MOVQ $101, AX = mov rax, 101
```

下面是通用通用寄存器的名字在 X64 和 plan9 中的对应关系:

X64	rax	rbx	rcx	rdx	rdi	rsi	rbp	rsp	r8	r9	r10	r11	r12	r13	r14
Plan9	AX	BX	CX	DX	DI	SI	BP	SP	R8	R9	R10	R11	R12	R13	R14

伪寄存器

Go 的汇编还引入了 4 个伪寄存器，援引官方文档的描述:

> - `FP` : Frame pointer: arguments and locals.
> - `PC` : Program counter: jumps and branches.
> - `SB` : Static base pointer: global symbols.
> - `SP` : Stack pointer: top of stack.

官方的描述稍微有一些问题，我们对这些说明进行一点扩充:

- FP: 使用形如 `symbol+offset(FP)` 的方式，引用函数的输入参数。例如 `arg0+0(FP)`，`arg1+8(FP)`，使用 FP 不加 symbol 时，无法通过编译，在汇编层面来讲，symbol 并没有什么用，加 symbol 主要是为了提升代码可读性。另外，官方文档虽然将伪寄存器 FP 称之为 frame pointer，实际上它根本不是 frame pointer，按照传统的 x86 的习惯来讲，frame pointer 是指向整个 stack frame 底部的 BP 寄存器。假如当前的 callee 函数是 add，在 add 的代码中引用 FP，该 FP 指向的位置不在 callee 的 stack frame 之内，而是在 caller 的 stack frame 上。具体可参见之后的 **栈结构** 一章。
- PC: 实际上就是在体系结构的知识中常见的 pc 寄存器，在 x86 平台下对应 ip 寄存器，amd64 上则是 rip。除了个别跳转之外，手写 plan9 代码与 PC 寄存器打交道的情况较少。
- SB: 全局静态基指针，一般用来声明函数或全局变量，在之后的函数知识和示例部分会看到具体用法。

- SP: plan9 的这个 SP 寄存器指向当前栈帧的局部变量的开始位置，使用形如 `symbol+offset(SP)` 的方式，引用函数的局部变量。offset 的合法取值是 [-framesize, 0)，注意是个左闭右开的区间。假如局部变量都是 8 字节，那么第一个局部变量就可以用 `localvar0-8(SP)` 来表示。这也是一个词不表意的寄存器。与硬件寄存器 SP 是两个不同的东西，在栈帧 size 为 0 的情况下，伪寄存器 SP 和硬件寄存器 SP 指向同一位置。手写汇编代码时，如果是 `symbol+offset(SP)` 形式，则表示伪寄存器 SP。如果是 `offset(SP)` 则表示硬件寄存器 SP。务必注意。对于编译输出(go tool compile -S / go tool objdump)的代码来讲，目前所有的 SP 都是硬件寄存器 SP，无论是否带 symbol。

我们这里对容易混淆的几点简单进行说明：

1. 伪 SP 和硬件 SP 不是一回事，在手写代码时，伪 SP 和硬件 SP 的区分方法是看该 SP 前是否有 symbol。如果有 symbol，那么即为伪寄存器，如果没有，那么说明是硬件 SP 寄存器。
2. SP 和 FP 的相对位置是会变的，所以不应该尝试用伪 SP 寄存器去找那些用 FP + offset 来引用的值，例如函数的入参和返回值。
3. 官方文档中说的伪 SP 指向 stack 的 top，是有问题的。其指向的局部变量位置实际上是整个栈的栈底(除 caller BP 之外)，所以说 bottom 更合适一些。
4. 在 go tool objdump/go tool compile -S 输出的代码中，是没有伪 SP 和 FP 寄存器的，我们上面说的区分伪 SP 和硬件 SP 寄存器的方法，对于上述两个命令的输出结果是没法使用的。在编译和反汇编的结果中，只有真实的 SP 寄存器。
5. FP 和 Go 的官方源代码里的 framepointer 不是一回事，源代码里的 framepointer 指的是 caller BP 寄存器的值，在这里和 caller 的伪 SP 是值是相等的。

以上说明看不懂也没关系，在熟悉了函数的栈结构之后再反复回来查看应该就可以明白了。个人意见，这些是 Go 官方挖的坑。。

变量声明

在汇编里所谓的变量，一般是存储在 .rodata 或者 .data 段中的只读值。对应到应用层的话，就是已初始化过的全局的 const、var、static 变量/常量。

使用 DATA 结合 GLOBL 来定义一个变量。DATA 的用法为:

```
DATA    symbol+offset(SB)/width, value
```

大多数参数都是字面意思，不过这个 offset 需要稍微注意。其含义是该值相对于符号 symbol 的偏移，而不是相对于全局某个地址的偏移。

使用 GLOBL 指令将变量声明为 global，额外接收两个参数，一个是 flag，另一个是变量的总大小。

```
GLOBL divtab(SB), RODATA, $64
```

GLOBL 必须跟在 DATA 指令之后，下面是一个定义了多个 readonly 的全局变量的完整例子:

```
DATA age+0x00(SB)/4, $18  // forever 18
GLOBL age(SB), RODATA, $4

DATA pi+0(SB)/8, $3.1415926
GLOBL pi(SB), RODATA, $8
```

```
DATA birthYear+0(SB)/4, $1988
GLOBL birthYear(SB), RODATA, $4
```

正如之前所说，所有符号在声明时，其 offset 一般都是 0。

有时也可能会想在全局变量中定义数组，或字符串，这时候就需要用上非 0 的 offset 了，例如:

```
DATA bio<>+0(SB)/8, $"oh yes i"
DATA bio<>+8(SB)/8, $"am here "
GLOBL bio<>(SB), RODATA, $16
```

大部分都比较好理解，不过这里我们又引入了新的标记 `<>`，这个跟在符号名之后，表示该全局变量只在当前文件中生效，类似于 C 语言中的 static。如果在另外文件中引用该变量的话，会报 `relocation target not found` 的错误。

本小节中提到的 flag，还可以有其它的取值:

> - `NOPROF` = 1
>   (For `TEXT` items.) Don't profile the marked function. This flag is deprecated.
> - `DUPOK` = 2
>   It is legal to have multiple instances of this symbol in a single binary. The linker will choose one of the duplicates to use.
> - `NOSPLIT` = 4
>   (For `TEXT` items.) Don't insert the preamble to check if the stack must be split. The frame for the routine, plus anything it calls, must fit in the spare space at the top of the stack segment. Used to protect routines such as the stack splitting code itself.
> - `RODATA` = 8
>   (For `DATA` and `GLOBL` items.) Put this data in a read-only section.
> - `NOPTR` = 16
>   (For `DATA` and `GLOBL` items.) This data contains no pointers and therefore does not need to be scanned by the garbage collector.
> - `WRAPPER` = 32
>   (For `TEXT` items.) This is a wrapper function and should not count as disabling `recover`.
> - `NEEDCTXT` = 64
>   (For `TEXT` items.) This function is a closure so it uses its incoming context register.

当使用这些 flag 的字面量时，需要在汇编文件中 `#include "textflag.h"`。

.s 和 .go 文件的全局变量互通

在 `.s` 文件中是可以直接使用 `.go` 中定义的全局变量的，看看下面这个简单的例子:

refer.go:

```
package main

var a = 999
func get() int

func main() {
```

```
    println(get())
}
```

refer.s:

```
#include "textflag.h"

TEXT ·get(SB), NOSPLIT, $0-8
    MOVQ ·a(SB), AX
    MOVQ AX, ret+0(FP)
    RET
```

·a(SB)，表示该符号需要链接器来帮我们进行重定向(relocation)，如果找不到该符号，会输出 `relocation target not found` 的错误。

例子比较简单，大家可以自行尝试。

函数声明

我们来看看一个典型的 plan9 的汇编函数的定义：

```
// func add(a, b int) int
//   => 该声明定义在同一个 package 下的任意 .go 文件中
//   => 只有函数头，没有实现
TEXT pkgname·add(SB), NOSPLIT, $0-8
    MOVQ a+0(FP), AX
    MOVQ a+8(FP), BX
    ADDQ AX, BX
    MOVQ BX, ret+16(FP)
    RET
```

为什么要叫 TEXT？如果对程序数据在文件中和内存中的分段稍有了解的同学应该知道，我们的代码在二进制文件中，是存储在 .text 段中的，这里也就是一种约定俗成的起名方式。实际上在 plan9 中 TEXT 是一个指令，用来定义一个函数。除了 TEXT 之外还有前面变量声明说到的 DATA/GLOBL。

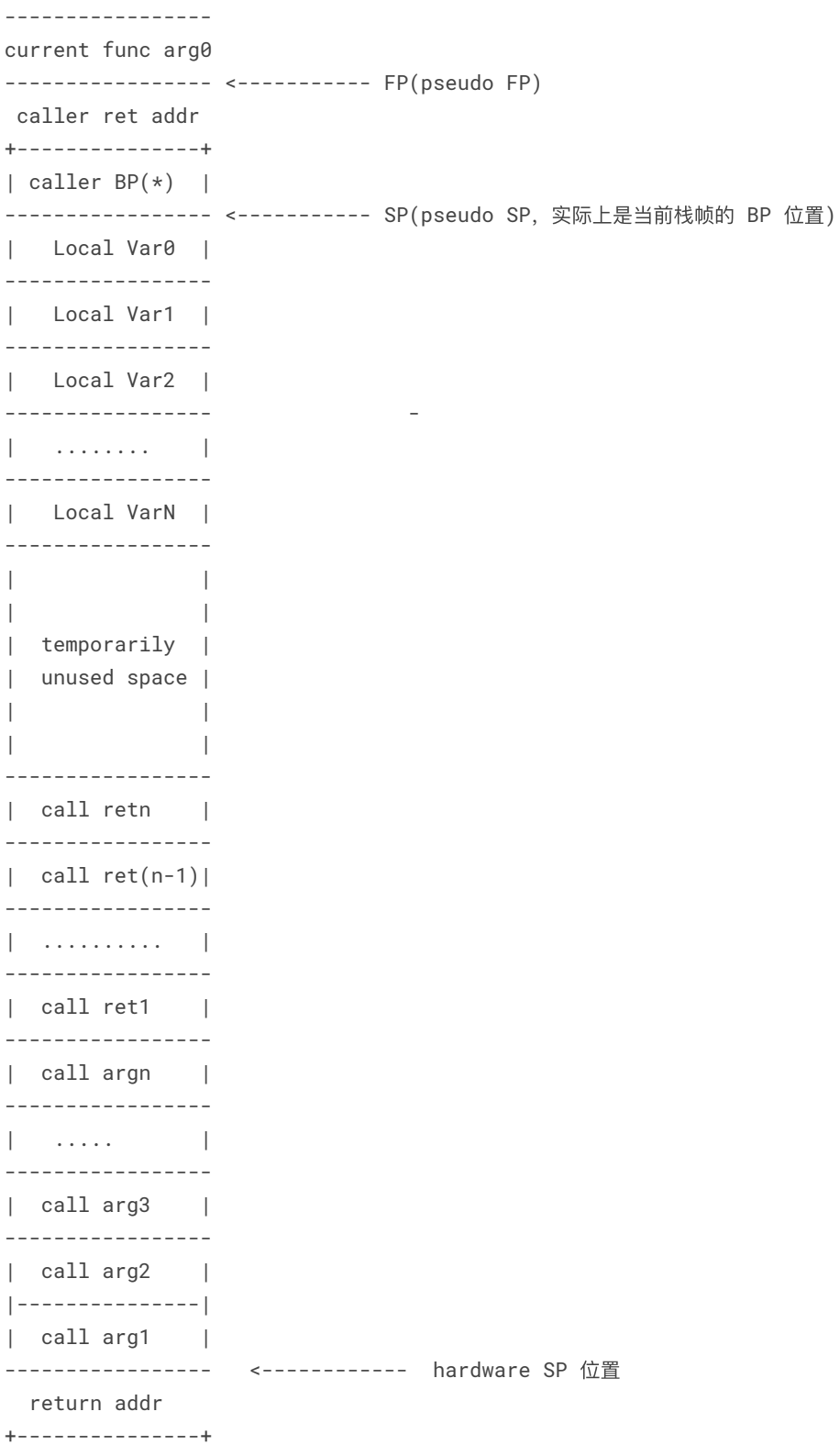
定义中的 pkgname 部分是可以省略的，非想写也可以写上。不过写上 pkgname 的话，在重命名 package 之后还需要改代码，所以推荐最好还是不要写。

中点 · 比较特殊，是一个 unicode 的中点，该点在 mac 下的输入方法是 `option+shift+9` 。在程序被链接之后，所有的中点 · 都会被替换为句号 . ，比如你的方法是 `runtime·main` ，在编译之后的程序里的符号则是 `runtime.main` 。嗯，看起来很变态。简单总结一下：

```
                                  参数及返回值大小
                                        |
TEXT pkgname·add(SB),NOSPLIT,$32-32
       |         |                |
      包名       函数名          栈帧大小(局部变量+可能需要的额外调用函数的参数空间的总大小，但不包括调用其它
```

栈结构

下面是一个典型的函数的栈结构图:



从原理上来讲，如果当前函数调用了其它函数，那么 return addr 也是在 caller 的栈上的，不过往栈上插 return addr 的过程是由 CALL 指令完成的，在 RET 时，SP 又会恢复到图上位置。我们在计算 SP 和参数相对位置时，可以认为硬件 SP 指向的就是图上的位置。

图上的 caller BP，指的是 caller 的 BP 寄存器值，有些人把 caller BP 叫作 caller 的 frame pointer，实际上这个习惯是从 x86 架构沿袭来的。Go 的 asm 文档中把伪寄存器 FP 也称为 frame pointer，但是这两个 frame pointer 根本不是一回事。

此外需要注意的是，caller BP 是在编译期由编译器插入的，用户手写代码时，计算 frame size 时是不包括这个 caller BP 部分的。是否插入 caller BP 的主要判断依据是:

1. 函数的栈帧大小大于 0
2. 下述函数返回 true

```
func Framepointer_enabled(goos, goarch string) bool {
    return framepointer_enabled != 0 && goarch == "amd64" && goos != "nacl"
}
```

如果编译器在最终的汇编结果中没有插入 caller BP(源代码中所称的 frame pointer)的情况下，伪 SP 和伪 FP 之间只有 8 个字节的 caller 的 return address，而插入了 BP 的话，就会多出额外的 8 字节。也就说伪 SP 和伪 FP 的相对位置是不固定的，有可能是间隔 8 个字节，也有可能间隔 16 个字节。并且判断依据会根据平台和 Go 的版本有所不同。

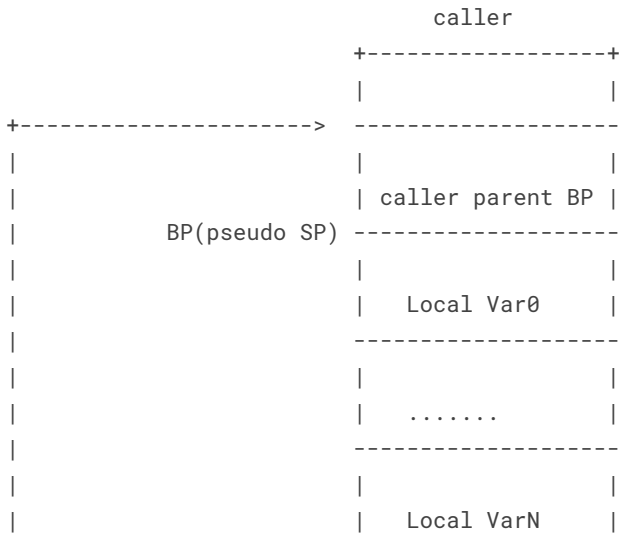
图上可以看到，FP 伪寄存器指向函数的传入参数的开始位置，因为栈是朝低地址方向增长，为了通过寄存器引用参数时方便，所以参数的摆放方向和栈的增长方向是相反的，即:

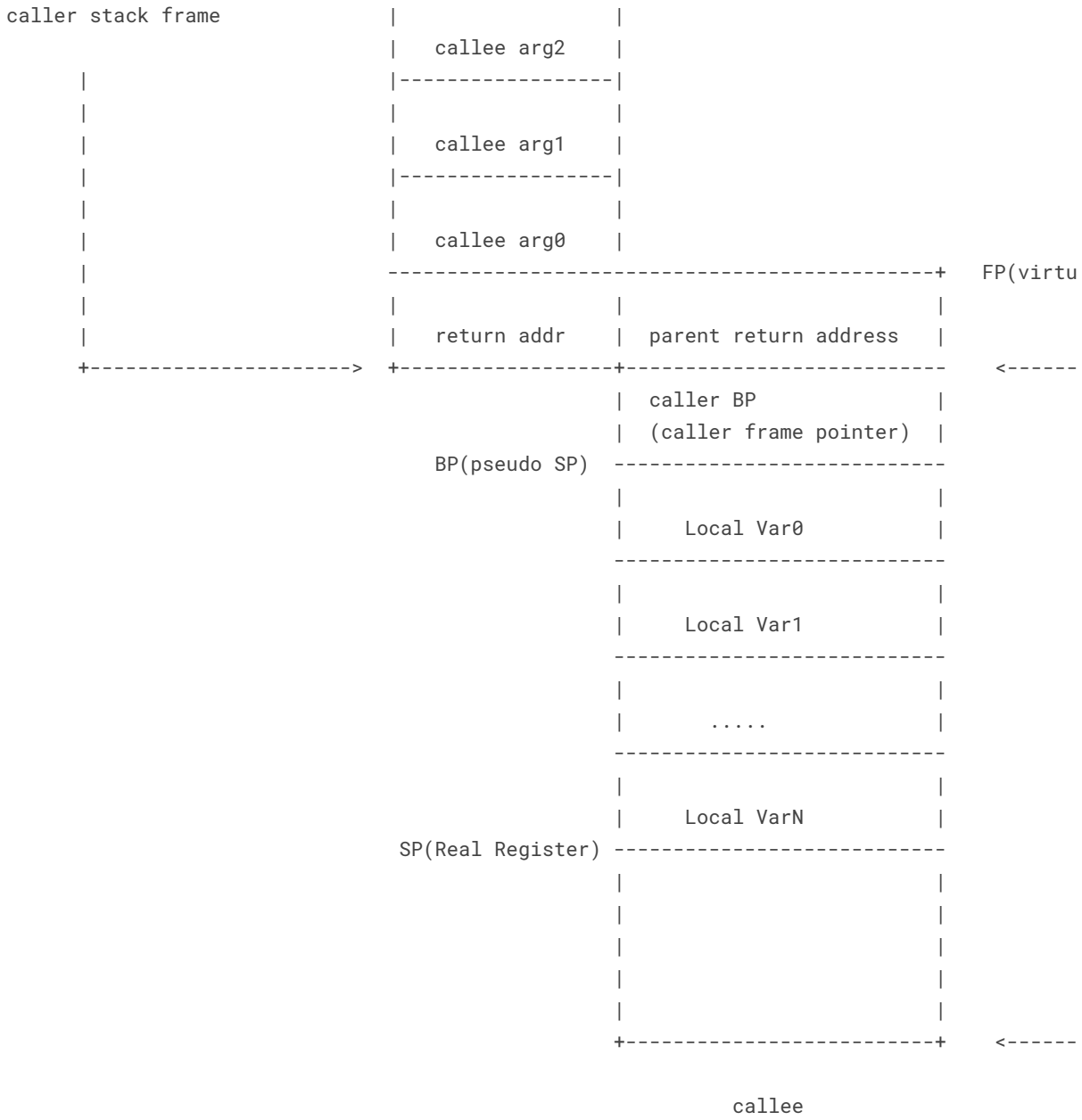
```
                            FP
high ----------------------> low
argN, ... arg3, arg2, arg1, arg0
```

假设所有参数均为 8 字节，这样我们就可以用 symname+0(FP) 访问第一个 参数，symname+8(FP) 访问第二个参数，以此类推。用伪 SP 来引用局部变量，原理上来讲差不多，不过因为伪 SP 指向的是局部变量的底部，所以 symname-8(SP) 表示的是第一个局部变量，symname-16(SP)表示第二个，以此类推。当然，这里假设局部变量都占用 8 个字节。

图的最上部的 caller return address 和 current func arg0 都是由 caller 来分配空间的。不算在当前的栈帧内。

因为官方文档本身较模糊，我们来一个函数调用的全景图，来看一下这些真假 SP/FP/BP 到底是个什么关系:





argsize 和 framesize 计算规则

argsize

在函数声明中:

```
TEXT pkgname·add(SB),NOSPLIT,$16-32
```

前面已经说过 $16-32 表示 $framesize-argsize。Go 在函数调用时，参数和返回值都需要由 caller 在其栈帧上备好空间。callee 在声明时仍然需要知道这个 argsize。argsize 的计算方法是，参数大小求和+返回值大小求和，例如入参是 3 个 int64 类型，返回值是 1 个 int64 类型，那么这里的 argsize = sizeof(int64) * 4。

不过真实世界永远没有我们假设的这么美好，函数参数往往混合了多种类型，还需要考虑内存对齐问题。

如果不确定自己的函数签名需要多大的 argsize，可以通过简单实现一个相同签名的空函数，然后 go tool objdump 来逆向查找应该分配多少空间。

framesize

函数的 framesize 就稍微复杂一些了，手写代码的 framesize 不需要考虑由编译器插入的 caller BP，要考虑：

1. 局部变量，及其每个变量的 size。
2. 在函数中是否有对其它函数调用时，如果有的话，调用时需要将 callee 的参数、返回值考虑在内。虽然 return address(rip)的值也是存储在 caller 的 stack frame 上的，但是这个过程是由 CALL 指令和 RET 指令完成 PC 寄存器的保存和恢复的，在手写汇编时，同样也是不需要考虑这个 PC 寄存器在栈上所需占用的 8 个字节的。
3. 原则上来说，调用函数时只要不把局部变量覆盖掉就可以了。稍微多分配几个字节的 framesize 也不会死。
4. 在确保逻辑没有问题的前提下，你愿意覆盖局部变量也没有问题。只要保证进入和退出汇编函数时的 caller 和 callee 能正确拿到返回值就可以。

地址运算

地址运算也是用 lea 指令，英文原意为 `Load Effective Address`，amd64 平台地址都是 8 个字节，所以直接就用 LEAQ 就好:

```
LEAQ (BX)(AX*8), CX
// 上面代码中的 8 代表 scale
// scale 只能是 0、2、4、8
// 如果写成其它值:
// LEAQ (BX)(AX*3), CX
// ./a.s:6: bad scale: 3

// 用 LEAQ 的话，即使是两个寄存器值直接相加，也必须提供 scale
// 下面这样是不行的
// LEAQ (BX)(AX), CX
// asm: asmidx: bad address 0/2064/2067
// 正确的写法是
LEAQ (BX)(AX*1), CX


// 在寄存器运算的基础上，可以加上额外的 offset
LEAQ 16(BX)(AX*1), CX

// 三个寄存器做运算，还是别想了
// LEAQ DX(BX)(AX*8), CX
// ./a.s:13: expected end of operand, found (
```

使用 LEAQ 的好处也比较明显，可以节省指令数。如果用基本算术指令来实现 LEAQ 的功能，需要两~三条以上的计算指令才能实现 LEAQ 的完整功能。

示例

add/sub/mul

math.go:

```
package main

import "fmt"

func add(a, b int) int // 汇编函数声明

func sub(a, b int) int // 汇编函数声明

func mul(a, b int) int // 汇编函数声明

func main() {
    fmt.Println(add(10, 11))
    fmt.Println(sub(99, 15))
    fmt.Println(mul(11, 12))
}
```

math.s:

```
#include "textflag.h" // 因为我们声明函数用到了 NOSPLIT 这样的 flag，所以需要将 textflag.h 包含进来

// func add(a, b int) int
TEXT ·add(SB), NOSPLIT, $0-24
    MOVQ a+0(FP), AX // 参数 a
    MOVQ b+8(FP), BX // 参数 b
    ADDQ BX, AX    // AX += BX
    MOVQ AX, ret+16(FP) // 返回
    RET

// func sub(a, b int) int
TEXT ·sub(SB), NOSPLIT, $0-24
    MOVQ a+0(FP), AX
    MOVQ b+8(FP), BX
    SUBQ BX, AX    // AX -= BX
    MOVQ AX, ret+16(FP)
    RET

// func mul(a, b int) int
TEXT ·mul(SB), NOSPLIT, $0-24
    MOVQ  a+0(FP), AX
    MOVQ  b+8(FP), BX
    IMULQ BX, AX    // AX *= BX
    MOVQ  AX, ret+16(FP)
    RET
    // 最后一行的空行是必须的，否则可能报 unexpected EOF
```

把这两个文件放在任意目录下，执行 `go build` 并运行就可以看到效果了。

伪寄存器 SP 、伪寄存器 FP 和硬件寄存器 SP

来写一段简单的代码证明伪 SP、伪 FP 和硬件 SP 的位置关系。

spspfp.s:

```
#include "textflag.h"

// func output(int) (int, int, int)
TEXT ·output(SB), $8-48
    MOVQ 24(SP), DX // 不带 symbol，这里的 SP 是硬件寄存器 SP
    MOVQ DX, ret3+24(FP) // 第三个返回值
    MOVQ perhapsArg1+16(SP), BX // 当前函数栈大小 > 0，所以 FP 在 SP 的上方 16 字节处
    MOVQ BX, ret2+16(FP) // 第二个返回值
    MOVQ arg1+0(FP), AX
    MOVQ AX, ret1+8(FP)  // 第一个返回值
    RET
```

spspfp.go:

```
package main

import (
    "fmt"
)

func output(int) (int, int, int) // 汇编函数声明

func main() {
    a, b, c := output(987654321)
    fmt.Println(a, b, c)
}
```

执行上面的代码，可以得到输出:

```
987654321 987654321 987654321
```

和代码结合思考，可以知道我们当前的栈结构是这样的:

```
------
ret2 (8 bytes)
------
ret1 (8 bytes)
------
ret0 (8 bytes)
------
arg0 (8 bytes)
------ FP
ret addr (8 bytes)
------
caller BP (8 bytes)
------ pseudo SP
frame content (8 bytes)
------ hardware SP
```

本小节例子的 framesize 是大于 0 的，读者可以尝试修改 framesize 为 0，然后调整代码中引用伪 SP 和硬件 SP 时的 offset，来研究 framesize 为 0 时，伪 FP，伪 SP 和硬件 SP 三者之间的相对位置。

本小节的例子是为了告诉大家，伪 SP 和伪 FP 的相对位置是会变化的，手写时不应该用伪 SP 和 >0 的 offset 来引用数据，否则结果可能会出乎你的预料。

汇编调用非汇编函数

output.s:

```
#include "textflag.h"

// func output(a,b int) int
TEXT ·output(SB), NOSPLIT, $24-8
    MOVQ a+0(FP), DX // arg a
    MOVQ DX, 0(SP) // arg x
    MOVQ b+8(FP), CX // arg b
    MOVQ CX, 8(SP) // arg y
    CALL ·add(SB) // 在调用 add 之前，已经把参数都通过物理寄存器 SP 搬到了函数的栈顶
    MOVQ 16(SP), AX // add 函数会把返回值放在这个位置
    MOVQ AX, ret+16(FP) // return result
    RET
```

output.go:

```
package main

import "fmt"

func add(x, y int) int {
    return x + y
}

func output(a, b int) int

func main() {
    s := output(10, 13)
    fmt.Println(s)
}
```

汇编中的循环

通过 DECQ 和 JZ 结合，可以实现高级语言里的循环逻辑:

sum.s:

```
#include "textflag.h"

// func sum(sl []int64) int64
TEXT ·sum(SB), NOSPLIT, $0-32
    MOVQ $0, SI
    MOVQ sl+0(FP), BX // &sl[0], addr of the first elem
    MOVQ sl+8(FP), CX // len(sl)
    INCQ CX           // CX++, 因为要循环 len 次

start:
    DECQ CX       // CX--
    JZ   done
```

```
    ADDQ (BX), SI // SI += *BX
    ADDQ $8, BX   // 指针移动
    JMP  start

done:
    // 返回地址是 24 是怎么得来的呢?
    // 可以通过 go tool compile -S math.go 得知
    // 在调用 sum 函数时，会传入三个值，分别为:
    // slice 的首地址、slice 的 len， slice 的 cap
    // 不过我们这里的求和只需要 len，但 cap 依然会占用参数的空间
    // 就是 16(FP)
    MOVQ SI, ret+24(FP)
    RET
```

sum.go:

```
package main

func sum([]int64) int64

func main() {
    println(sum([]int64{1, 2, 3, 4, 5}))
}
```

扩展话题

标准库中的一些数据结构

数值类型

标准库中的数值类型很多:

1. int/int8/int16/int32/int64
2. uint/uint8/uint16/uint32/uint64
3. float32/float64
4. byte/rune
5. uintptr

这些类型在汇编中就是一段存储着数据的连续内存，只是内存长度不一样，操作的时候看好数据长度就行。

slice

前面的例子已经说过了，slice 在传递给函数的时候，实际上会展开成三个参数:

1. 首元素地址
2. slice 的 len
3. slice 的 cap

在汇编中处理时，只要知道这个原则那就很好办了，按顺序还是按索引操作随你开心。

string

```
package main

//go:noinline
func stringParam(s string) {}

func main() {
    var x = "abcc"
    stringParam(x)
}
```

用 go tool compile -S 输出其汇编:

```
0x001d 00029 (stringParam.go:11)    LEAQ    go.string."abcc"(SB), AX  // 获取 RODATA 段中的字
0x0024 00036 (stringParam.go:11)    MOVQ    AX, (SP) // 将获取到的地址放在栈顶，作为第一个参数
0x0028 00040 (stringParam.go:11)    MOVQ    $4, 8(SP) // 字符串长度作为第二个参数
0x0031 00049 (stringParam.go:11)    PCDATA  $0, $0 // gc 相关
0x0031 00049 (stringParam.go:11)    CALL    "".stringParam(SB) // 调用 stringParam 函数
```

在汇编层面 string 就是地址 + 字符串长度。

struct

struct 在汇编层面实际上就是一段连续内存，在作为参数传给函数时，会将其展开在 caller 的栈上传给对应的 callee:

struct.go

```
package main

type address struct {
    lng int
    lat int
}

type person struct {
    age    int
    height int
    addr   address
}

func readStruct(p person) (int, int, int, int)

func main() {
    var p = person{
        age:    99,
        height: 88,
        addr: address{
            lng: 77,
            lat: 66,
        },
    }
    a, b, c, d := readStruct(p)
```

```
    println(a, b, c, d)
}
```

struct.s

```
#include "textflag.h"

TEXT ·readStruct(SB), NOSPLIT, $0-64
    MOVQ arg0+0(FP), AX
    MOVQ AX, ret0+32(FP)
    MOVQ arg1+8(FP), AX
    MOVQ AX, ret1+40(FP)
    MOVQ arg2+16(FP), AX
    MOVQ AX, ret2+48(FP)
    MOVQ arg3+24(FP), AX
    MOVQ AX, ret3+56(FP)
    RET
```

上述的程序会输出 99, 88, 77, 66，这表明即使是内嵌结构体，在内存分布上依然是连续的。

map

通过对下述文件进行汇编(go tool compile -S)，我们可以得到一个 map 在对某个 key 赋值时所需要做的操作:

m.go:

```
package main

func main() {
    var m = map[int]int{}
    m[43] = 1
    var n = map[string]int{}
    n["abc"] = 1
    println(m, n)
}
```

看一看第七行的输出:

```
0x0085 00133 (m.go:7)   LEAQ    type.map[int]int(SB), AX
0x008c 00140 (m.go:7)   MOVQ    AX, (SP)
0x0090 00144 (m.go:7)   LEAQ    ""..autotmp_2+232(SP), AX
0x0098 00152 (m.go:7)   MOVQ    AX, 8(SP)
0x009d 00157 (m.go:7)   MOVQ    $43, 16(SP)
0x00a6 00166 (m.go:7)   PCDATA  $0, $1
0x00a6 00166 (m.go:7)   CALL    runtime.mapassign_fast64(SB)
0x00ab 00171 (m.go:7)   MOVQ    24(SP), AX
0x00b0 00176 (m.go:7)   MOVQ    $1, (AX)
```

前面我们已经分析过调用函数的过程，这里前几行都是在准备 runtime.mapassign_fast64(SB) 的参数。去 runtime 里看看这个函数的签名:

```
func mapassign_fast64(t *maptype, h *hmap, key uint64) unsafe.Pointer {
```

不用看函数的实现我们也大概能推测出函数输入参数和输出参数的关系了，把入参和汇编指令对应的话:

```
t *maptype
=>
LEAQ    type.map[int]int(SB), AX
MOVQ    AX, (SP)

h *hmap
=>
LEAQ    ""..autotmp_2+232(SP), AX
MOVQ    AX, 8(SP)

key uint64
=>
MOVQ    $43, 16(SP)
```

返回参数就是 key 对应的可以写值的内存地址，拿到该地址后我们把想要写的值写进去就可以了:

```
MOVQ    24(SP), AX
MOVQ    $1, (AX)
```

整个过程还挺复杂的，我们手抄一遍倒也可以实现。不过还要考虑，不同类型的 map，实际上需要执行的 runtime 中的 assign 函数是不同的，感兴趣的同学可以汇编本节的示例自行尝试。

整体来讲，用汇编来操作 map 并不是一个明智的选择。

channel

channel 在 runtime 也是比较复杂的数据结构，如果在汇编层面操作，实际上也是调用 runtime 中 chan.go 中的函数，和 map 比较类似，这里就不展开说了。

获取 goroutine id

Go 的 goroutine 是一个叫 g 的结构体，内部有自己的唯一 id，不过 runtime 没有把这个 id 暴露出来，但不知道为什么有很多人就是想把这个 id 得到。于是就有了各种或其 goroutine id 的库。

在 struct 一小节我们已经提到，结构体本身就是一段连续的内存，我们知道起始地址和字段的偏移量的话，很容易就可以把这段数据搬运出来:

go_tls.h:

```
#ifdef GOARCH_arm
#define LR R14
#endif

#ifdef GOARCH_amd64
#define    get_tls(r)    MOVQ TLS, r
#define    g(r)    0(r)(TLS*1)
#endif

#ifdef GOARCH_amd64p32
#define    get_tls(r)    MOVL TLS, r
#define    g(r)    0(r)(TLS*1)
```

```
#endif

#ifdef GOARCH_386
#define    get_tls(r)    MOVL TLS, r
#define    g(r)    0(r)(TLS*1)
#endif
```

goid.go:

```
package goroutineid
import "runtime"
var offsetDict = map[string]int64{
    // ... 省略一些行
    "go1.7":    192,
    "go1.7.1":  192,
    "go1.7.2":  192,
    "go1.7.3":  192,
    "go1.7.4":  192,
    "go1.7.5":  192,
    "go1.7.6":  192,
    // ... 省略一些行
}

var offset = offsetDict[runtime.Version()]

// GetGoID returns the goroutine id
func GetGoID() int64 {
    return getGoID(offset)
}

func getGoID(off int64) int64
```

goid.s:

```
#include "textflag.h"
#include "go_tls.h"

// func getGoID() int64
TEXT ·getGoID(SB), NOSPLIT, $0-16
    get_tls(CX)
    MOVQ g(CX), AX
    MOVQ offset(FP), BX
    LEAQ 0(AX)(BX*1), DX
    MOVQ (DX), AX
    MOVQ AX, ret+8(FP)
    RET
```

这样就实现了一个简单的获取 struct g 中的 goid 字段的小 library，作为玩具放在这里:

> https://github.com/cch123/goroutineid

SIMD

SIMD 是 Single Instruction, Multiple Data 的缩写，在 Intel 平台上的 SIMD 指令集先后为 SSE，AVX，AVX2，AVX512，这些指令集引入了标准以外的指令，和宽度更大的寄存器，例如:

- 128 位的 XMM0~XMM31 寄存器。
- 256 位的 YMM0~YMM31 寄存器。
- 512 位的 ZMM0~ZMM31 寄存器。

这些寄存器的关系，类似 RAX，EAX，AX 之间的关系。指令方面可以同时对多组数据进行移动或者计算，例如:

- movups : 把4个不对准的单精度值传送到xmm寄存器或者内存
- movaps : 把4个对准的单精度值传送到xmm寄存器或者内存

上述指令，当我们将数组作为函数的入参时有很大概率会看到，例如:

arr_par.go:

```
package main

import "fmt"

func pr(input [3]int) {
    fmt.Println(input)
}

func main() {
    pr([3]int{1, 2, 3})
}
```

go compile -S:

```
0x001d 00029 (arr_par.go:10)    MOVQ    "".statictmp_0(SB), AX
0x0024 00036 (arr_par.go:10)    MOVQ    AX, (SP)
0x0028 00040 (arr_par.go:10)    MOVUPS  "".statictmp_0+8(SB), X0
0x002f 00047 (arr_par.go:10)    MOVUPS  X0, 8(SP)
0x0034 00052 (arr_par.go:10)    CALL    "".pr(SB)
```

可见，编译器在某些情况下已经考虑到了性能问题，帮助我们使用 SIMD 指令集来对数据搬运进行了优化。

因为 SIMD 这个话题本身比较广，这里就不展开细说了。

特别感谢

研究过程基本碰到不太明白的都去骚扰卓巨巨了，就是这位 https://mzh.io/ 大大。特别感谢他，给了不少线索和提示。

参考资料

1. https://quasilyte.github.io/blog/post/go-asm-complementary-reference/#external-resources
2. http://davidwong.fr/goasm
3. https://www.doxsey.net/blog/go-and-assembly

4. https://github.com/golang/go/files/447163/GoFunctionsInAssembly.pdf
5. https://golang.org/doc/asm

参考资料[4]需要特别注意，在该 slide 中给出的 callee stack frame 中把 caller 的 return address 也包含进去了，个人认为不是很合适。