Go语言中很多设计思想和工具都是传承自Plan9操作系统，Go汇编语言也是基于Plan9汇编演化而来。根据Rob Pike的介绍，大神Ken Thompson在1986年为Plan9系统编写的C语言编译器输出的汇编伪代码就是Plan9汇编的前身。所谓的Plan9汇编语言只是便于以手工方式书写该C语言编译器输出的汇编伪代码而已。

无论高级语言如何发展，作为最接近CPU的汇编语言的地位依然是无法彻底被替代的。只有通过汇编语言才能彻底挖掘CPU芯片的全部功能，因此操作系统的引导过程必须要依赖汇编语言的帮助。只有通过汇编语言才能彻底榨干CPU芯片的性能，因此很多底层的加密解密等对性能敏感的算法会考虑通过汇编语言进行性能优化。

对于每一个严肃的Gopher，Go汇编语言都是一个不可忽视的技术。因为哪怕只懂一点点汇编，也便于更好地理解计算机原理，也更容易理解Go语言中动态栈/接口等高级特性的实现原理。而且掌握了Go汇编语言之后，你将重新站在编程语言鄙视链的顶端，不用担心再被任何其它所谓的高级编程语言用户鄙视。

本章我们将以AMD64为主要开发环境，简单地探讨Go汇编语言的基础用法。

# 1. 快速入门

Go汇编程序始终是幽灵一样的存在。我们将通过分析简单的Go程序输出的汇编代码，然后照猫画虎用汇编实现一个简单的输出程序。

## 1.1. 实现和声明

Go汇编语言并不是一个独立的语言，因为Go汇编程序无法独立使用。Go汇编代码必须以Go包的方式组织，同时包中至少要有一个Go语言文件用于指明当前包名等基本包信息。如果Go汇编代码中定义的变量和函数要被其它Go语言代码引用，还需要通过Go语言代码将汇编中定义的符号声明出来。用于变量的定义和函数的定义Go汇编文件类似于C语言中的.c文件，而用于导出汇编中定义符号的Go源文件类似于C语言的.h文件。

## **3.1.2 定义整数变量**

为了简单，我们先用Go语言定义并赋值一个整数变量，然后查看生成的汇编代码。

首先创建一个pkg.go文件，内容如下：

package pkg

var Id = 9527

代码中只定义了一个int类型的包级变量，并进行了初始化。然后用以下命令查看的Go语言程序对应的伪汇编代码：

$ go tool compile -S pkg.go

"".Id SNOPTRDATA size=8

0x0000 37 25 00 00 00 00 00 00 '.......

其中go tool compile命令用于调用Go语言提供的底层命令工具，其中-S参数表示输出汇编格式。输出的汇编比较简单，其中"".Id对应Id变量符号，变量的内存大小为8个字节。变量的初始化内容为37 25 00 00 00 00 00 00，对应十六进制格式的0x2537，对应十进制为9527。SNOPTRDATA是相关的标志，其中NOPTR表示数据中不包含指针数据。

以上的内容只是目标文件对应的汇编，和Go汇编语言虽然相似当并不完全等价。Go语言官网自带了一个Go汇编语言的入门教程，地址在：[https://golang.org/doc/asm](https://golang.org/doc/asm" \t "http://books.studygolang.com/advanced-go-programming-book/ch3-asm/_blank) 。

Go汇编语言提供了DATA命令用于初始化包变量，DATA命令的语法如下：

DATA symbol+offset(SB)/width, value

其中symbol为变量在汇编语言中对应的标识符，offset是符号开始地址的偏移量，width是要初始化内存的宽度大小，value是要初始化的值。其中当前包中Go语言定义的符号symbol，在汇编代码中对应·symbol，其中“·”中点符号为一个特殊的unicode符号。

我们采用以下命令可以给Id变量初始化为十六进制的0x2537，对应十进制的9527（常量需要以美元符号$开头表示）：

DATA ·Id+0(SB)/1,$0x37

DATA ·Id+1(SB)/1,$0x25

变量定义好之后需要导出以供其它代码引用。Go汇编语言提供了GLOBL命令用于将符号导出：

GLOBL symbol(SB), width

其中symbol对应汇编中符号的名字，width为符号对应内存的大小。用以下命令将汇编中的·Id变量导出：

GLOBL ·Id, $8

现在已经初步完成了用汇编定义一个整数变量的工作。

为了便于其它包使用该Id变量，我们还需要在Go代码中声明该变量，同时也给变量指定一个合适的类型。修改pkg.go的内容如下：

package pkg

var Id int

现状Go语言的代码不再是定义一个变量，语义变成了声明一个变量（声明一个变量时不能再进行初始化操作）。而Id变量的定义工作已经在汇编语言中完成了。

我们将完整的汇编代码放到pkg\_amd64.s文件中：

GLOBL ·Id(SB),$8

DATA ·Id+0(SB)/1,$0x37

DATA ·Id+1(SB)/1,$0x25

DATA ·Id+2(SB)/1,$0x00

DATA ·Id+3(SB)/1,$0x00

DATA ·Id+4(SB)/1,$0x00

DATA ·Id+5(SB)/1,$0x00

DATA ·Id+6(SB)/1,$0x00

DATA ·Id+7(SB)/1,$0x00

文件名pkg\_amd64.s的后缀名表示AMD64环境下的汇编代码文件。

虽然pkg包是用汇编实现，但是用法和之前的Go语言版本完全一样：

package main

import pkg "pkg包的路径"

func main() {

println(pkg.Id)

}

对于Go包的用户来说，用Go汇编语言或Go语言实现并无任何区别。

## **3.1.3 定义字符串变量**

在前一个例子中，我们通过汇编定义了一个整数变量。现在我们提高一点难度，尝试通过汇编定义一个字符串变量。虽然从Go语言角度看，定义字符串和整数变量的写法基本相同，但是字符串底层却有着比单个整数更复杂的数据结构。

实验的流程和前面的例子一样，还是先用Go语言实现类似的功能，然后观察分析生成的汇编代码，最后用Go汇编语言仿写。首先创建pkg.go文件，用Go语言定义字符串：

package pkg

var Name = "gopher"

然后用以下命令查看的Go语言程序对应的伪汇编代码：

$ go tool compile -S pkg.go

go.string."gopher" SRODATA dupok size=6

0x0000 67 6f 70 68 65 72 gopher

"".Name SDATA size=16

0x0000 00 00 00 00 00 00 00 00 06 00 00 00 00 00 00 00 ................

rel 0+8 t=1 go.string."gopher"+0

输出中出现了一个新的符号go.string."gopher"，根据其长度和内容分析可以猜测是对应底层的"gopher"字符串数据。因为Go语言的字符串并不是值类型，Go字符串其实是一种只读的引用类型。如果多个代码中出现了相同的"gopher"只读字符串时，程序链接后可以引用的同一个符号go.string."gopher"。因此，该符号有一个SRODATA标志表示这个数据在只读内存段，dupok表示出现多个相同标识符的数据时只保留一个就可以了。

而真正的Go字符串变量Name对应的大小却只有16个字节了。其实Name变量并没有直接对应“gopher”字符串，而是对应16字节大小的reflect.StringHeader结构体：

type reflect.StringHeader struct {

Data uintptr

Len int

}

从汇编角度看，Name变量其实对应的是reflect.StringHeader结构体类型。前8个字节对应底层真实字符串数据的指针，也就是符号go.string."gopher"对应的地址。后8个字节对应底层真实字符串数据的有效长度，这里是6个字节。

现在创建pkg\_amd64.s文件，尝试通过汇编代码重新定义并初始化Name字符串：

GLOBL ·NameData(SB),$8

DATA ·NameData(SB)/8,$"gopher"

GLOBL ·Name(SB),$16

DATA ·Name+0(SB)/8,$·NameData(SB)

DATA ·Name+8(SB)/8,$6

因为在Go汇编语言中，go.string."gopher"不是一个合法的符号，因此我们无法通过手工创建（这是给编译器保留的部分特权，因为手工创建类似符号可能打破编译器输出代码的某些规则）。因此我们新创建了一个·NameData符号表示底层的字符串数据。然后定义·Name符号内存大小为16字节，其中前8个字节用·NameData符号对应的地址初始化，后8个字节为常量6表示字符串长度。

当用汇编定义好字符串变量并导出之后，还需要在Go语言中声明该字符串变量。然后就可以用Go语言代码测试Name变量了：

package main

import pkg "path/to/pkg"

func main() {

println(pkg.Name)

}

不幸的是这次运行产生了以下错误：

pkgpath.NameData: missing Go //type information for global symbol: size 8

错误提示汇编中定义的NameData符号没有类型信息。其实Go汇编语言中定义的数据并没有所谓的类型，每个符号只不过是对应一块内存而已，因此NameData符号也是没有类型的。但是Go语言是再带垃圾回收器的语言，而Go汇编语言是工作在自动垃圾回收体系框架内的。档Go语言的垃圾回收器在扫描到NameData变量的时候，无法知晓该变量内部是否包含指针，因此就出现了这种错误。错误的根本原因并不是NameData没有类型，而是NameData变量没有标注是否会含有指针信息。

通过给NameData变量增加一个NOPTR标志，表示其中不会包含指针数据可以修复该错误：

#include "textflag.h"

GLOBL ·NameData(SB),NOPTR,$8

通过给·NameData增加NOPTR标志档方式表示其中不含指针数据。我们也可以通过给·NameData变量在Go语言中增加一个不含指针并且大小为8个字节的类型来修改该错误：

package pkg

var NameData [8]bytevar Name string

我们将NameData声明为长度为8的字节数组。编译器可以通过类型分析出该变量不会包含指针，因此汇编代码中可以省略NOPTR标志。现在垃圾回收器在遇到该变量的时候就会停止内部数据的扫描。

在这个实现中，Name字符串底层其实引用的是NameData内存对应的“gopher”字符串数据。因此，如果NameData发生变化，Name字符串的数据也会跟着变化。

func main() {

println(pkg.Name)

pkg.NameData[0] = '?'

println(pkg.Name)

}

当然这和字符串的只读定义是冲突的，正常的代码需要避免出现这种情况。最好的方法是不要导出内部的NameData变量，这样可以避免内部数据被无意破坏。

在用汇编定义字符串时我们可以换一种思维：将底层的字符串数据和字符串头结构体定义在一起，这样可以避免引入NameData符号：

GLOBL ·Name(SB),$24

DATA ·Name+0(SB)/8,$·Name+16(SB)

DATA ·Name+8(SB)/8,$6

DATA ·Name+16(SB)/8,$"gopher"

在新的结构中，Name符号对应的内存从16字节变为24字节，多出的8个字节存放底层的“gopher”字符串。·Name符号前16个字节依然对应reflect.StringHeader结构体：Data部分对应$·Name+16(SB)，表示数据的地址为Name符号往后偏移16个字节的位置；Len部分依然对应6个字节的长度。这是C语言程序员经常使用档技巧。

## **3.1.4 定义main函数**

前面的例子已经展示了如何通过汇编定义整型和字符串类型变量。我们现在将尝试用汇编实现函数，然后输出一个字符串。

先创建main.go文件，创建并初始化字符串变量，同时声明main函数：

package main

var helloworld = "你好, 世界"

func main()

然后创建main\_amd64.s文件，里面对应main函数的实现：

TEXT ·main(SB), $16-0

MOVQ ·helloworld+0(SB), AX; MOVQ AX, 0(SP)

MOVQ ·helloworld+8(SB), BX; MOVQ BX, 8(SP)

CALL runtime·printstring(SB)

CALL runtime·printnl(SB)

RET

TEXT ·main(SB), $16-0用于定义main函数，其中$16-0表示main函数的帧大小是16个字节（对应string头部结构体的大小，用于给runtime·printstring函数传递参数），0表示main函数没有参数和返回值。main函数内部通过调用运行时内部的runtime·printstring(SB)函数来打印字符串。然后调用runtime·printnl打印换行符号。

Go语言函数在函数调用时，完全通过栈传递调用参数和返回值。先通过MOVQ指令，将helloworld对应的字符串头部结构体的16个字节复制到栈指针SP对应的16字节的空间，然后通过CALL指令调用对应函数。最后使用RET指令表示当前函数返回。

## **3.1.5 特殊字符**

Go语言函数或方法符号在编译为目标文件后，目标文件中的每个符号均包含对应包的绝对导入路径。因此目标文件的符号可能非常复杂，比如“path/to/pkg.(\*SomeType).SomeMethod”或“go.string."abc"”等名字。目标文件的符号名中不仅仅包含普通的字母，还可能包含点号、星号、小括弧和双引号等诸多特殊字符。而Go语言的汇编器是从plan9移植过来的二把刀，并不能处理这些特殊的字符，导致了用Go汇编语言手工实现Go诸多特性时遇到种种限制。

Go汇编语言同样遵循Go语言少即是多的哲学，它只保留了最基本的特性：定义变量和全局函数。其中在变量和全局函数等名字中引入特殊等分隔符号支持Go语言等包体系。为了简化Go汇编器的词法扫描程序的实现，特别引入了Unicode中的中点·和大写的除法/，对应的Unicode码点为U+00B7和U+2215。汇编器编译后，中点·会被替换为ASCII中的点“.”，大写的除法会被替换为ASCII码中的除法“/”，比如math/rand·Int会被替换为math/rand.Int。这样可以将中点和浮点数中的小数点、大写的除法和表达式中的除法符号分开，可以简化汇编程序词法分析部分的实现。

即使暂时抛开Go汇编语言设计取舍的问题，在不同的操作系统不同等输入法中如何输入中点·和除法/两个字符就是一个挑战。这两个字符在 [https://golang.org/doc/asm](https://golang.org/doc/asm" \t "http://books.studygolang.com/advanced-go-programming-book/ch3-asm/_blank) 文档中均有描述，因此直接从该页面复制是最简单可靠的方式。

如果是macOS系统，则有以下几种方法输入中点·：在不开输入法时，可直接用 option+shift+9 输入；如果是自带的简体拼音输入法，输入左上角~键对应·，如果是自带的Unicode输入法，则可以输入对应的Unicode码点。其中Unicode输入法可能是最安全可靠等输入方式。

## **3.1.6 没有分号**

Go汇编语言中分号可以用于分隔同一行内的多个语句。下面是用分号混乱排版的汇编代码：

TEXT ·main(SB), $16-0; MOVQ ·helloworld+0(SB), AX; MOVQ ·helloworld+8(SB), BX;

MOVQ AX, 0(SP);MOVQ BX, 8(SP);CALL runtime·printstring(SB);

CALL runtime·printnl(SB);

RET;

和Go语言一样，也可以省略行尾的分号。当遇到末尾时，汇编器会自动插入分号。下面是省略分号后的代码：

TEXT ·main(SB), $16-0

MOVQ ·helloworld+0(SB), AX; MOVQ AX, 0(SP)

MOVQ ·helloworld+8(SB), BX; MOVQ BX, 8(SP)

CALL runtime·printstring(SB)

CALL runtime·printnl(SB)

RET

和Go语言一样，语句之间多个连续的空白字符和一个空格是等价的。

# 2. 计算机结构

汇编语言是直面计算机的编程语言，因此理解计算机结构是掌握汇编语言的前提。当前流行的计算机基本采用的是冯·诺伊曼计算机体系结构（在某些特殊领域还有哈佛体系架构）。冯·诺依曼结构也称为普林斯顿结构，采用的是一种将程序指令和数据存储在一起的存储结构。冯·诺伊曼计算机中的指令和数据存储器其实指的是计算机中的内存，然后在配合CPU处理器就组成了一个最简单的计算机了。

汇编语言其实是一种非常简单的编程语言，因为它面向的计算机模型就是非常简单的。让人觉得汇编语言难学主要有几个原因：不同类型的CPU都有自己的一套指令；即使是相同的CPU，32位和64位的运行模式依然会有差异；不同的汇编工具同样有自己特有的汇编指令；不同的操作系统和高级编程语言和底层汇编的调用规范并不相同。本节将描述几个有趣的汇编语言模型，最后精简出一个适用于AMD64架构的精简指令集，以便于Go汇编语言的学习。

## 2.1. 图灵机和BF语言

图灵机是由图灵提出的一种抽象计算模型。机器有一条无限长的纸带，纸带分成了一个一个的小方格，每个方格有不同的颜色，这类似于计算机中的内存。同时机器有一个探头在纸带上移来移去，类似于通过内存地址来读写内存上的数据。机器头有一组内部计算状态，还有一些固定的程序（更像一个哈佛结构）。在每个时刻，机器头都要从当前纸带上读入一个方格信息，然后根据自己的内部状态和当前要执行的程序指令将信息输出到纸带方格上，同时更新自己的内部状态并进行移动。

图灵机虽然不容易编程，但是非常容易理解。有一种极小化的BrainFuck计算机语言，它的工作模式和图灵机非常相似。BrainFuck由Urban Müller在1993年创建的，简称为BF语言。Müller最初的设计目标是建立一种简单的、可以用最小的编译器来实现的、符合图灵完全思想的编程语言。这种语言由八种状态构成，早期为Amiga机器编写的编译器（第二版）只有240个字节大小！

就象它的名字所暗示的，brainfuck程序很难读懂。尽管如此，brainfuck图灵机一样可以完成任何计算任务。虽然brainfuck的计算方式如此与众不同，但它确实能够正确运行。这种语言基于一个简单的机器模型，除了指令，这个机器还包括：一个以字节为单位、被初始化为零的数组、一个指向该数组的指针（初始时指向数组的第一个字节）、以及用于输入输出的两个字节流。这是一种按照图灵完备的语言，它的主要设计思路是：用最小的概念实现一种“简单”的语言。BrainFuck 语言只有八种符号，所有的操作都由这八种符号的组合来完成。

下面是这八种状态的描述，其中每个状态由一个字符标识：

| **字符** | **C语言类比** | **含义** |
| --- | --- | --- |
| > | ++ptr; | 指针加一 |
| < | --ptr; | 指针减一 |
| + | ++\*ptr; | 指针指向的字节的值加一 |
| - | --\*ptr; | 指针指向的字节的值减一 |
| . | putchar(\*ptr); | 输出指针指向的单元内容（ASCⅡ码） |
| , | \*ptr = getch(); | 输入内容到指针指向的单元（ASCⅡ码） |
| [ | while(\*ptr) {} | 如果指针指向的单元值为零，向后跳转到对应的 ] 指令的次一指令处 |
| ] |  | 如果指针指向的单元值不为零，向前跳转到对应的 [ 指令的次一指令处 |

下面是一个 brainfuck 程序，向标准输出打印"hi"字符串：

++++++++++[>++++++++++<-]>++++.+.

理论上我们可以将BF语言当作目标机器语言，将其它高级语言编译为BF语言后就可以在BF机器上运行了。

## 2.2. 人力资源机器游戏

《人力资源机器》（Human Resource Machine）是一款设计精良汇编语言编程游戏。在游戏中，玩家扮演一个职员角色，来模拟人力资源机器的运行。通过完成上司给的每一份任务来实现晋升的目标，完成任务的途径就是用游戏提供的11个机器指令编写正确的汇编程序，最终得到正确的输出结果。人力资源机器的汇编语言可以认为是跨平台、跨操作系统的通用的汇编语言，因为在macOS、Windows、Linux和iOS上该游戏的玩法都是完全一致的。

人力资源机器的机器模型非常简单：INBOX命令对应输入设备，OUTBOX对应输出设备，玩家小人对应一个寄存器，临时存放数据的地板对应内存，然后是数据传输、加减、跳转等基本的指令。总共有11个机器指令:

| **名称** | **解释** |
| --- | --- |
| INBOX | 从输入通道取一个整数数据，放到手中(寄存器) |
| OUTBOX | 将手中（寄存器）的数据放到输出通道，然后手中将没有数据（此时有些指令不能运行） |
| COPYFROM | 将地板上某个编号的格子中的数据复制到手中（手中之前的数据作废），地板格子必须有数据 |
| COPYTO | 将手中（寄存器）的数据复制到地板上某个编号的格子中，手中的数据不变 |
| ADD | 将手中（寄存器）的数据和某个编号对应的地板格子的数据相加，新数据放到手中（手中之前的数据作废） |
| SUB | 将手中（寄存器）的数据和某个编号对应的地板格子的数据相减，新数据放到手中（手中之前的数据作废） |
| BUMP+ | 自加一 |
| BUMP- | 自减一 |
| JUMP | 跳转 |
| JUMP =0 | 为零条件跳转 |
| JUMP <0 | 为负条件跳转 |

除了机器指令外，游戏中有些环节还提供类似寄存器的场所，用于存放临时的数据。人力资源机器游戏的机器指令主要分为以下几类：

* 输入/输出(INBOX, OUTBOX): 输入后手中将只有1份新拿到的数据, 输出后手中将没有数据。
* 数据传输指令(COPYFROM/COPYTO): 主要用于仅有的1个寄存器（手中）和内存之间的数据传输，传输时要确保源数据是有效的
* 算术相关(ADD/SUB/BUMP+/BUMP-)
* 跳转指令: 如果是条件跳转，寄存器中必须要有数据

主流的处理器也有类似的指令。除了基本的算术和逻辑预算指令外，再配合有条件跳转指令就可以实现分支、循环等常见控制流结构了。

下图是某一层的任务：将输入数据的0剔除，非0的数据依次输出，右边部分是解决方案。



*图 3-1 人力资源机器*

整个程序只有一个输入指令、一个输出指令和两个跳转指令共四个指令：

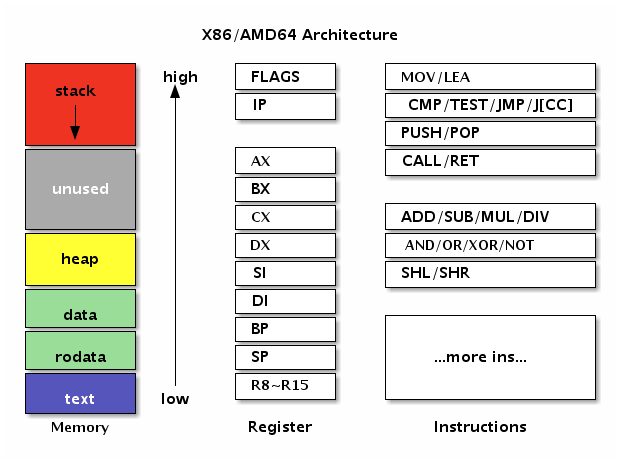
LOOP:  
 INBOX  
 JUMP-if-zero LOOP  
 OUTBOX  
 JUMP LOOP

首先通过INBOX指令读取一个数据包；然后判断包裹的数据是否为0，如果是0的话就跳转到开头继续读取下一个数据包；否则将输出数据包，然后再跳转到开头。以此循环无休止地处理数据包裹，直到任务完成晋升到更高一级的岗位，然后处理类似的但更复杂的任务。

## 2.3. X86-64体系结构

X86其实是是80X86的简称（后面三个字母），包括Intel 8086、80286、80386以及80486等指令集合，因此其架构被称为x86架构。x86-64是AMD公司于1999年设计的x86架构的64位拓展，向后兼容于16位及32位的x86架构。X86-64目前正式名称为AMD64，也就是Go语言中GOARCH环境变量指定的AMD64。如果没有特殊说明的话，本章中的汇编程序都是针对64位的X86-64环境。

在使用汇编语言之前必须要了解对应的CPU体系结构。下面是X86/AMD架构图：



*图 3-2 AMD64架构*

左边是内存部分是常见的内存布局。其中**text**一般对应代码段，用于存储要执行指令数据，代码段一般是只读的。然后是rodata和data数据段，数据段一般用于存放全局的数据，其中rodata是只读的数据段。而**heap段**则用于管理动态的数据，**stack段**用于管理每个函数调用时相关的数据。在汇编语言中一般重点关注text代码段和data数据段，因此Go汇编语言中专门提供了对应TEXT和DATA命令用于定义代码和数据。

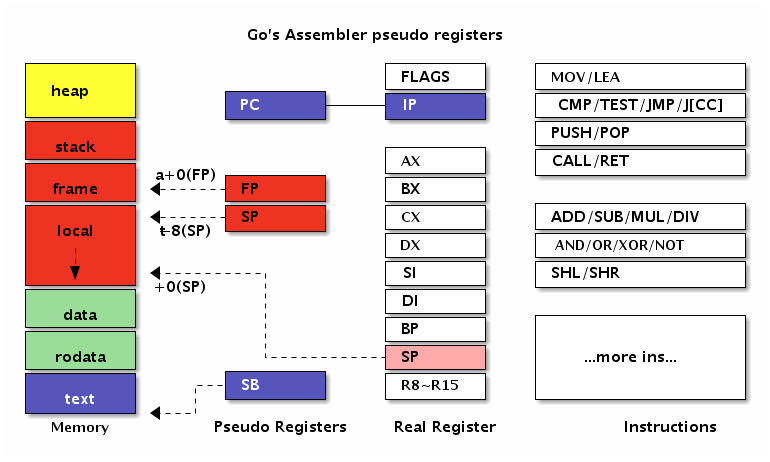
中间是X86提供的寄存器。寄存器是CPU中最重要的资源，每个要处理的内存数据原则上需要先放到寄存器中才能由CPU处理，同时寄存器中处理完的结果需要再存入内存。X86中除了状态寄存器FLAGS和指令寄存器IP两个特殊的寄存器外，还有AX、BX、CX、DX、SI、DI、BP、SP几个通用寄存器。在X86-64中又增加了八个以R8-R15方式命名的通用寄存器。因为历史的原因R0-R7并不是通用寄存器，它们只是X87开始引入的MMX指令专有的寄存器。在通用寄存器中BP和SP是两个比较特殊的寄存器：其中BP用于记录当前函数帧的开始位置，和函数调用相关的指令会隐式地影响BP的值；SP则对应当前栈指针的位置，和栈相关的指令会隐式地影响SP的值；而某些调试工具需要BP寄存器才能正常工作。

右边是**X86的指令集**。CPU是由指令和寄存器组成，指令是每个CPU内置的算法，指令处理的对象就是全部的寄存器和内存。我们可以将每个指令看作是CPU内置标准库中提供的一个个函数，然后基于这些函数构造更复杂的程序的过程就是用汇编语言编程的过程。

## 2.4. Go汇编中的伪寄存器

Go汇编为了简化汇编代码的编写，引入了PC、FP、SP、SB四个伪寄存器。四个伪寄存器加其它的通用寄存器就是Go汇编语言对CPU的重新抽象，该抽象的结构也适用于其它非X86类型的体系结构。

四个伪寄存器和X86/AMD64的内存和寄存器的相互关系如下图：



*图 3-3 Go汇编的伪寄存器*

在AMD64环境，伪PC寄存器其实是IP指令计数器寄存器的别名。伪FP寄存器对应的是函数的帧指针，一般用来访问函数的参数和返回值。伪SP栈指针对应的是当前函数栈帧的底部（不包括参数和返回值部分），一般用于定位局部变量。伪SP是一个比较特殊的寄存器，因为还存在一个同名的SP真寄存器。真SP寄存器对应的是栈的顶部，一般用于定位调用其它函数的参数和返回值。

当需要区分伪寄存器和真寄存器的时候只需要记住一点：伪寄存器一般需要一个标识符和偏移量为前缀，如果没有标识符前缀则是真寄存器。比如(SP)、+8(SP)没有标识符前缀为真SP寄存器，而a(SP)、b+8(SP)有标识符为前缀表示伪寄存器。

## 2.5. X86-64指令集

很多汇编语言的教程都会强调汇编语言是不可移植的。严格来说汇编语言是在不同的CPU类型、或不同的操作系统环境、或不同的汇编工具链下是不可移植的，而在同一种CPU中运行的机器指令是完全一样的。汇编语言这种不可移植性正是其普及的一个极大的障碍。虽然CPU指令集的差异是导致不好移植的较大因素，但是汇编语言的相关工具链对此也有不可推卸的责任。而源自Plan9的Go汇编语言对此做了一定的改进：首先Go汇编语言在相同CPU架构上是完全一致的，也就是屏蔽了操作系统的差异；同时Go汇编语言将一些基础并且类似的指令抽象为相同名字的伪指令，从而减少不同CPU架构下汇编代码的差异（寄存器名字和数量的差异是一直存在的）。本节的目的也是找出一个较小的精简指令集，以简化Go汇编语言的学习。

X86是一个极其复杂的系统，有人统计x86-64中指令有将近一千个之多。不仅仅如此，X86中的很多单个指令的功能也非常强大，比如有论文证明了仅仅一个MOV指令就可以构成一个图灵完备的系统。以上这是两种极端情况，太多的指令和太少的指令都不利于汇编程序的编写，但是也从侧面体现了MOV指令的重要性。

通用的基础机器指令大概可以分为数据传输指令、算术运算和逻辑运算指令、控制流指令和其它指令等几类。因此我们可以尝试精简出一个X86-64指令集，以便于Go汇编语言的学习。

因此我们先看看重要的MOV指令。其中MOV指令可以用于将字面值移动到寄存器、字面值移到内存、寄存器之间的数据传输、寄存器和内存之间的数据传输。需要注意的是，MOV传输指令的内存操作数只能有一个，可以通过某个临时寄存器达到类似目的。最简单的是忽略符号位的数据传输操作，386和AMD64指令一样，不同的1、2、4和8字节宽度有不同的指令：

| **Data Type** | **386/AMD64** | **Comment** |
| --- | --- | --- |
| [1]byte | MOVB | B => Byte |
| [2]byte | MOVW | W => Word |
| [4]byte | MOVL | L => Long |
| [8]byte | MOVQ | Q => Quadword |

MOV指令它不仅仅用于在寄存器和内存之间传输数据，而且还可以用于处理数据的扩展和截断操作。当数据宽度和寄存器的宽度不同又需要处理符号位时，386和AMD64有各自不同的指令：

| **Data Type** | **386** | **AMD64** | **Comment** |
| --- | --- | --- | --- |
| int8 | MOVBLSX | MOVBQSX | sign extend |
| uint8 | MOVBLZX | MOVBQZX | zero extend |
| int16 | MOVWLSX | MOVWQSX | sign extend |
| uint16 | MOVWLZX | MOVWQZX | zero extend |

比如当需要将一个int64类型的数据转为bool类型时，则需要使用MOVBQZX指令处理。

基础算术指令有ADD、SUB、MUL、DIV等指令。其中ADD、SUB、MUL、DIV用于加、减、乘、除运算，最终结果存入目标寄存器。基础的逻辑运算指令有AND、OR和NOT等几个指令，对应逻辑与、或和取反等几个指令。

| **名称** | **解释** |
| --- | --- |
| ADD | 加法 |
| SUB | 减法 |
| MUL | 乘法 |
| DIV | 除法 |
| AND | 逻辑与 |
| OR | 逻辑或 |
| NOT | 逻辑取反 |

其中算术和逻辑指令是顺序编程的基础。通过逻辑比较影响状态寄存器，再结合有条件跳转指令就可以实现更复杂的分支或循环结构。需要注意的是MUL和DIV等乘除法指令可能隐含使用了某些寄存器，指令细节请查阅相关手册。

控制流指令有CMP、JMP-if-x、JMP、CALL、RET等指令。CMP指令用于两个操作数做减法，根据比较结果设置状态寄存器的符号位和零位，可以用于有条件跳转的跳转条件。JMP-if-x是一组有条件跳转指令，常用的有JL、JLZ、JE、JNE、JG、JGE等指令，对应小于、小于等于、等于、不等于、大于和大于等于等条件时跳转。JMP指令则对应无条件跳转，将要跳转的地址设置到IP指令寄存器就实现了跳转。而CALL和RET指令分别为调用函数和函数返回指令。

| **名称** | **解释** |
| --- | --- |
| JMP | 无条件跳转 |
| JMP-if-x | 有条件跳转，JL、JLZ、JE、JNE、JG、JGE |
| CALL | 调用函数 |
| RET | 函数返回 |

无条件和有条件调整指令是实现分支和循环控制流的基础指令。理论上，我们也可以通过跳转指令实现函数的调用和返回功能。不过因为目前函数已经是现代计算机中的一个最基础的抽象，因此大部分的CPU都针对函数的调用和返回提供了专有的指令和寄存器。

其它比较重要的指令有LEA、PUSH、POP等几个。其中LEA指令将标准参数格式中的内存地址加载到寄存器（而不是加载内存位置的内容）。PUSH和POP分别是压栈和出栈指令，通用寄存器中的SP为栈指针，栈是向低地址方向增长的。

| **名称** | **解释** |
| --- | --- |
| LEA | 取地址 |
| PUSH | 压栈 |
| POP | 出栈 |

当需要通过间接索引的方式访问数组或结构体等某些成员对应的内存时，可以用LEA指令先对目前内存取地址，然后在操作对应内存的数据。而栈指令则可以用于函数调整自己的栈空间大小。

最后需要说明的是，Go汇编语言可能并没有支持全部的CPU指令。如果遇到没有支持的CPU指令，可以通过Go汇编语言提供的BYTE命令将真实的CPU指令对应的机器码填充到对应的位置。完整的X86指令在 [https://github.com/golang/arch/blob/master/x86/x86.csv](https://github.com/golang/arch/blob/master/x86/x86.csv" \t "http://books.studygolang.com/advanced-go-programming-book/ch3-asm/_blank) 文件定义。同时Go汇编还正对一些指令定义了别名，具体可以参考这里 [https://golang.org/src/cmd/internal/obj/x86/anames.go](https://golang.org/src/cmd/internal/obj/x86/anames.go" \t "http://books.studygolang.com/advanced-go-programming-book/ch3-asm/_blank) 。

# 3. 常量和全局变量

程序中的一切变量的初始值都直接或间接地依赖常量或常量表达式生成。在Go语言中很多变量是默认零值初始化的，但是Go汇编中定义的变量最好还是手工通过常量初始化。有了常量之后，就可以衍生定义全局变量，并使用常量组成的表达式初始化其它各种变量。本节将简单讨论Go汇编语言中常量和全局变量的用法。

## 3.1. 常量

Go汇编语言中常量以$美元符号为前缀。常量的类型有整数常量、浮点数常量、字符常量和字符串常量等几种类型。以下是几种类型常量的例子：

$1 // 十进制  
$0xf4f8fcff // 十六进制  
$1.5 // 浮点数  
$'a' // 字符  
$"abcd" // 字符串

其中整数类型常量默认是十进制格式，也可以用十六进制格式表示整数常量。所有的常量最终都必须和要初始化的变量内存大小匹配。

对于数值型常量，可以通过常量表达式构成新的常量：

$2+2 // 常量表达式

$3&1<<2 // == $4

$(3&1)<<2 // == $4

其中常量表达式中运算符的优先级和Go语言保持一致。

Go汇编语言中的常量其实不仅仅只有编译时常量，还包含运行时常量。比如包中全局的变量和全局函数在运行时地址也是固定不变的，这里地址不会变量的包变量和函数的地址也是一种汇编常量。

下面是本章第一节用汇编定义的字符串代码：

GLOBL ·NameData(SB),$8

DATA ·NameData(SB)/8,$"gopher"

GLOBL ·Name(SB),$16

DATA ·Name+0(SB)/8,$·NameData(SB)

DATA ·Name+8(SB)/8,$6

其中$·NameData(SB)也是以$美元符号为前缀，因此也可以将它看作是一个常量，它对应的是NameData包变量的地址。在汇编指令中，我们也可以通过LEA指令来获取NameData变量的地址。

## 3.2. 全局变量

在Go语言中，变量根据作用域和生命周期有全局变量和局部变量之分。全局变量是包一级的变量，全局变量一般有着较为固定的内存地址，声明周期跨越整个程序运行时间。而局部变量一般是函数内定义的的变量，只有在函数被执行的时间才被在栈上创建，当函数调用完成后将回收（暂时不考虑闭包对局部变量捕获的问题）。

从Go汇编语言角度来看，全局变量和局部变量有着非常大的差异。在Go汇编中全局变量和全局函数更为相似，都是通过一个人为定义的符号来引用对应的内存，区别只是内存中存放是数据还是要执行的指令。因为在冯诺伊曼系统结构的计算机中指令也是数据，而且指令和数据存放在统一编址的内存中。因为指令和数据并没有本质的差别，因此我们甚至可以像操作数据那样动态生成指令（这是所有JIT技术的原理）。而局部变量则需在了解了汇编函数之后，才能通过SP栈空间来隐式定义。

在Go汇编语言中，内存是通过SB伪寄存器定位。SB是Static base pointer的缩写，意为静态内存的开始地址。我们可以将SB想象为一个和内容容量有相同大小的字节数组，所有的静态全局符号通常可以通过SB加一个偏移量定位，而我们定义的符号其实就是相对于SB内存开始地址偏移量。对于SB伪寄存器，全局变量和全局函数的符号并没有任何区别。

要定义全局变量，首先要声明一个变量对应的符号，以及变量对应的内存大小。导出变量符号的语法如下：

GLOBL symbol(SB), width

GLOBL汇编指令用于定义名为symbol的变量，变量对应的内存宽度为width，内存宽度部分必须用常量初始化。下面的代码通过汇编定义一个int32类型的count变量：

GLOBL ·count(SB),$4

其中符号·count以中点开头表示是当前包的变量，最终符号名为被展开为path/to/pkg.count。count变量的大小是4个字节，常量必须以$美元符号开头。内存的宽度必须是2的指数倍，编译器最终会保证变量的真实地址对齐到机器字倍数。需要注意的是，在Go汇编中我们无法为count变量指定具体的类型。在汇编中定义全局变量时，我们只关心变量的名字和内存大小，变量最终的类型只能在Go语言中声明。

变量定义之后，我们可以通过DATA汇编指令指定对应内存中的数据，语法如下：

DATA symbol+offset(SB)/width, value

具体的含义是从symbol+offset偏移量开始，width宽度的内存，用value常量对应的值初始化。DATA初始化内存时，width必须是1、2、4、8几个宽度之一，因为再大的内存无法一次性用一个uint64大小的值表示。

对于int32类型的count变量来说，我们既可以逐个字节初始化，也可以一次性初始化：

DATA ·count+0(SB)/1,$1  
DATA ·count+1(SB)/1,$2  
DATA ·count+2(SB)/1,$3  
DATA ·count+3(SB)/1,$4  
  
// or  
  
DATA ·count+0(SB)/4,$0x04030201

因为X86处理器是小端序，因此用十六进制0x04030201初始化全部的4个字节，和用1、2、3、4逐个初始化4个字节是一样的效果。

最后还需要在Go语言中声明对应的变量（和C语言头文件声明变量的作用类似），这样垃圾回收器会根据变量的类型来管理其中的指针相关的内存数据。

### 3.2.1. 数组类型

汇编中数组也是一种非常简单的类型。Go语言中数组是一种有着扁平内存结构的基础类型。因此[2]byte类型和[1]uint16类型有着相同的内存结构。只有当数组和结构体结合之后情况才会变的稍微复杂。

下面我们尝试用汇编定义一个[2]int类型的数组变量num：

var num [2]int

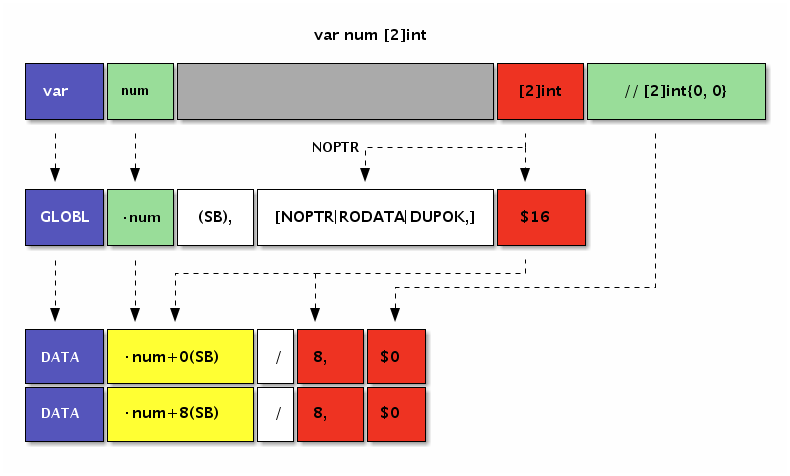
然后在汇编中定义一个对应16字节大小的变量，并用零值进行初始化：

GLOBL ·num(SB),$16

DATA ·num+0(SB)/8,$0

DATA ·num+8(SB)/8,$0

下图是Go语句和汇编语句定义变量时的对应关系：



*图 3-4 变量定义*

汇编代码中并不需要NOPTR标志，因为Go编译器会从Go语言语句声明的[2]int类型中推导出该变量内部没有指针数据。

### 3.2.2. bool型变量

Go汇编语言定义变量无法指定类型信息，因此需要先通过Go语言声明变量的类型。以下是在Go语言中声明的几个bool类型变量：

var (

boolValue bool

trueValue bool

falseValue bool

)

在Go语言中声明的变量不能含有初始化语句。然后下面是amd64环境的汇编定义：

GLOBL ·boolValue(SB),$1 // 未初始化

GLOBL ·trueValue(SB),$1 // var trueValue = true

DATA ·trueValue(SB)/1,$1 // 非 0 均为 true

GLOBL ·falseValue(SB),$1 // var falseValue = true

DATA ·falseValue(SB)/1,$0

bool类型的内存大小为1个字节。并且汇编中定义的变量需要手工指定初始化值，否则将可能导致产生未初始化的变量。当需要将1个字节的bool类型变量加载到8字节的寄存器时，需要使用MOVBQZX指令将不足的高位用0填充。

### 3.2.3. int型变量

所有的整数类型均有类似的定义的方式，比较大的差异是整数类型的内存大小和整数是否是有符号。下面是声明的int32和uint32类型变量：

var int32Value int32

var uint32Value uint32

在Go语言中声明的变量不能含有初始化语句。然后下面是amd64环境的汇编定义：

GLOBL ·int32Value(SB),$4

DATA ·int32Value+0(SB)/1,$0x01 // 第0字节

DATA ·int32Value+1(SB)/1,$0x02 // 第1字节

DATA ·int32Value+2(SB)/2,$0x03 // 第3-4字节

GLOBL ·uint32Value(SB),$4

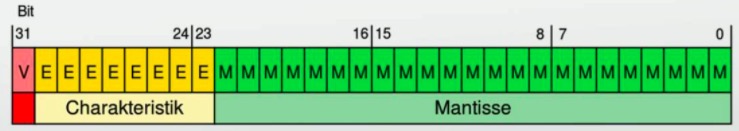
DATA ·uint32Value(SB)/4,$0x01020304 // 第1-4字节

汇编定义变量时初始化数据并不区分整数是否有符号。只有在CPU指令处理该寄存器数据时，才会根据指令的类型来取分数据的类型或者是否带有符号位。

### 3.2.4. float型变量

Go汇编语言通常无法区分变量是否是浮点数类型，与之相关的浮点数机器指令会将变量当作浮点数处理。Go语言的浮点数遵循IEEE754标准，有float32单精度浮点数和float64双精度浮点数之分。

IEEE754标准中，最高位1bit为符号位，然后是指数位（指数为采用移码格式表示），然后是有效数部分（其中小数点左边的一个bit位被省略）。下图是IEEE754中float32类型浮点数的bit布局：



*图 3-5 IEEE754浮点数结构*

IEEE754浮点数还有一些奇妙的特性：比如有正负两个0；除了无穷大和无穷小Inf还有非数NaN；同时如果两个浮点数有序那么对应的有符号整数也是有序的（反之则不一定成立，因为浮点数中存在的非数是不可排序的）。浮点数是程序中最难琢磨的角落，因为程序中很多手写的浮点数字面值常量根本无法精确表达，浮点数计算涉及到的误差舍入方式可能也的随机的。

下面是在Go语言中声明两个浮点数（如果没有在汇编中定义变量，那么声明的同时也会定义变量）。

var float32Value float32

var float64Value float64

然后在汇编中定义并初始化上面声明的两个浮点数：

GLOBL ·float32Value(SB),$4

DATA ·float32Value+0(SB)/4,$1.5 // var float32Value = 1.5

GLOBL ·float64Value(SB),$8

DATA ·float64Value(SB)/8,$0x01020304 // bit 方式初始化

我们在上一节精简的算术指令中都是针对整数，如果要通过整数指令处理浮点数的加减法必须根据浮点数的运算规则进行：先对齐小数点，然后进行整数加减法，最后再对结果进行归一化并处理精度舍入问题。不过在目前的主流CPU中，都提针对浮点数提供了专有的计算指令。

### 3.2.5. string类型变量

从Go汇编语言角度看，字符串只是一种结构体。string的头结构定义如下：

type reflect.StringHeader struct {

Data uintptr

Len int

}

在amd64环境中StringHeader有16个字节大小，因此我们先在Go代码声明字符串变量，然后在汇编中定义一个16字节大小的变量：

var helloworld string

GLOBL ·helloworld(SB),$16

同时我们可以为字符串准备真正的数据。在下面的汇编代码中，我们定义了一个text当前文件内的私有变量（以<>为后缀名），内容为“Hello World!”：

GLOBL text<>(SB),NOPTR,$16

DATA text<>+0(SB)/8,$"Hello Wo"

DATA text<>+8(SB)/8,$"rld!"

虽然text<>私有变量表示的字符串只有12个字符长度，但是我们依然需要将变量的长度扩展为2的指数倍数，这里也就是16个字节的长度。其中NOPTR表示text<>不包含指针数据。

然后使用text私有变量对应的内存地址对应的常量来初始化字符串头结构体中的Data部分，并且手工指定Len部分为字符串的长度：

DATA ·helloworld+0(SB)/8,$text<>(SB) // StringHeader.Data

DATA ·helloworld+8(SB)/8,$12 // StringHeader.Len

需要注意的是，字符串是只读类型，要避免在汇编中直接修改字符串底层数据的内容。

### 3.2.6. slice类型变量

slice变量和string变量相似，只不过是对应的是切片头结构体而已。切片头的结构如下：

type reflect.SliceHeader struct {

Data uintptr

Len int

Cap int

}

对比可以发现，切片的头的前2个成员字符串是一样的。因此我们可以在前面字符串变量的基础上，再扩展一个Cap成员就成了切片类型了：

var helloworld []byte

GLOBL ·helloworld(SB),$24 // var helloworld []byte("Hello World!")

DATA ·helloworld+0(SB)/8,$text<>(SB) // StringHeader.Data

DATA ·helloworld+8(SB)/8,$12 // StringHeader.Len

DATA ·helloworld+16(SB)/8,$16 // StringHeader.Cap

GLOBL text<>(SB),$16

DATA text<>+0(SB)/8,$"Hello Wo" // ...string data...

DATA text<>+8(SB)/8,$"rld!" // ...string data...

因为切片和字符串的相容性，我们可以将切片头的前16个字节临时作为字符串使用，这样可以省去不必要的转换。

### 3.2.7. map/channel类型变量

map/channel等类型并没有公开的内部结构，它们只是一种未知类型的指针，无法直接初始化。在汇编代码中我们只能为类似变量定义并进行0值初始化：

var m map[string]int

var ch chan int

GLOBL ·m(SB),$8 // var m map[string]int

DATA ·m+0(SB)/8,$0

GLOBL ·ch(SB),$8 // var ch chan int

DATA ·ch+0(SB)/8,$0

其实在runtime包中为汇编提供了一些辅助函数。比如在汇编中可以通过runtime.makemap和runtime.makechan内部函数来创建map和chan变量。辅助函数的签名如下：

func makemap(mapType \*byte, hint int, mapbuf \*any) (hmap map[any]any)

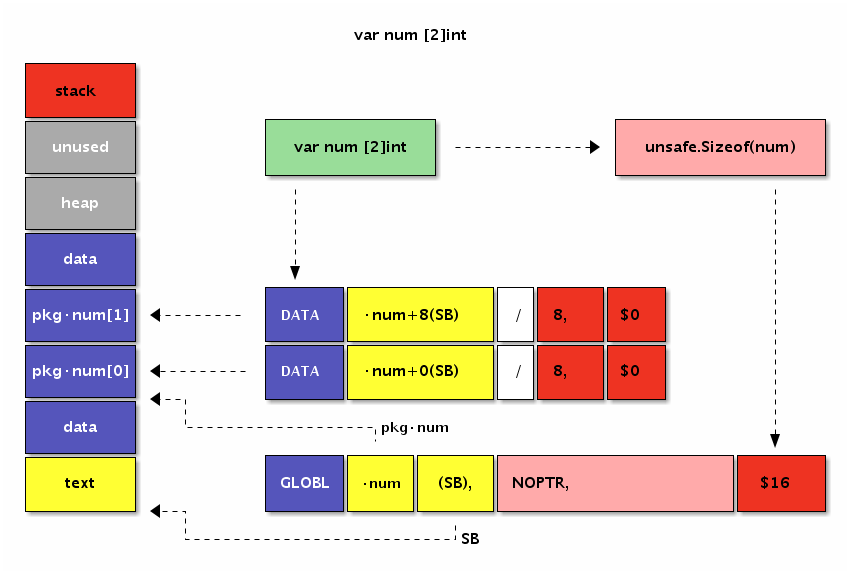
func makechan(chanType \*byte, size int) (hchan chan any)

需要注意的是，makemap是一种范型函数，可以创建不同类型的map，map的具体类型是通过mapType参数指定。

## 3.3. 变量的内存布局

我们已经多次强调，在Go汇编语言中变量是没有类型的。因此在Go语言中有着不同类型的变量，底层可能对应的是相同的内存结构。深刻理解每个变量的内存布局是汇编编程时的必备条件。

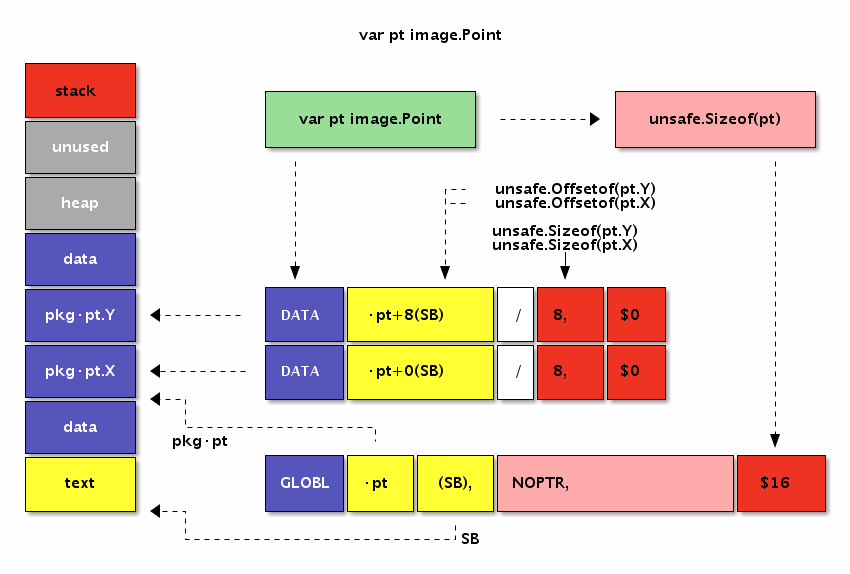
首先查看前面已经见过的[2]int类型数组的内存布局：



*图 3-6 变量定义*

变量在data段分配空间，数组的元素地址依次从低向高排列。

然后再查看下标准库图像包中image.Point结构体类型变量的内存布局：



*图 3-7 结构体变量定义*

变量也时在data段分配空间，变量结构体成员的地址也是依次从低向高排列。

因此[2]int和image.Point类型底层有着近似相同的内存布局。

## 3.4. 标识符规则和特殊标志

Go语言的标识符可以由绝对的包路径加标识符本身定位，因此不同包中的标识符即使同名也不会有问题。Go汇编是通过特殊的符号来表示斜杠和点符号，因为这样可以简化汇编器词法扫描部分代码的编写，只要通过字符串替换就可以了。

下面是汇编中常见的几种标识符的使用方式（通常也适用于函数标识符）：

GLOBL ·pkg\_name1(SB),$1

GLOBL main·pkg\_name2(SB),$1

GLOBL my/pkg·pkg\_name(SB),$1

此外，Go汇编中可以定义仅当前文件可以访问的私有标识符（类似C语言中文件内static修饰的变量），以<>为后缀名：

GLOBL file\_private<>(SB),$1

这样可以减少私有标识符对其它文件内标识符命名的干扰。

此外，Go汇编语言还在"textflag.h"文件定义了一些标志。其中用于变量的标志有DUPOK、RODATA和NOPTR几个。DUPOK表示该变量对应的标识符可能有多个，在链接时只选择其中一个即可（一般用于合并相同的常量字符串，减少重复数据占用的空间）。RODATA标志表示将变量定义在只读内存段，因此后续任何对此变量的修改操作将导致异常（recover也无法捕获）。NOPTR则表示此变量的内部不含指针数据，让垃圾回收器忽略对该变量的扫描。如果变量已经在Go代码中声明过的话，Go编译器会自动分析出该变量是否包含指针，这种时候可以不用手写NOPTR标志。

比如下面的例子是通过汇编来定义一个只读的int类型的变量：

var const\_id int // readonly

#include "textflag.h"

GLOBL ·const\_id(SB),NOPTR|RODATA,$8

DATA ·const\_id+0(SB)/8,$9527

我们使用#include语句包含定义标志的"textflag.h"头文件（和C语言中预处理相同）。然后GLOBL汇编命令在定义变量时，给变量增加了NOPTR和RODATA两个标志（多个标志之间采用竖杠分割），表示变量中没有指针数据同时定义在只读数据段。

变量一般也叫可取地址的值，但是const\_id虽然可以取地址，但是确实不能修改。不能修改的限制并不是由编译器提供，而是因为对该变量的修改会导致对只读内存段进行写，从而导致异常。

## 3.5. 小结

以上我们初步展示了通过汇编定义全局变量的用法。但是真实的环境中我们并不推荐通过汇编定义变量——因为用Go语言定义变量更加简单和安全。在Go语言中定义变量，编译器可以帮助我们计算好变量的大小，生成变量的初始值，同时也包含了足够的类型信息。汇编语言的优势是挖掘机器的特性和性能，用汇编定义变量则无法发挥这些优势。因此在理解了汇编定义变量的用法后，建议大家谨慎使用。

# 4. 函数

终于到函数了！因为Go汇编语言中，可以也建议通过Go语言来定义全局变量，那么剩下的也就是函数了。只有掌握了汇编函数的基本用法，才能真正算是Go汇编语言入门。本章将简单讨论Go汇编中函数的定义和用法。

## 4.1. 基本语法

函数标识符通过TEXT汇编指令定义，表示该行开始的指令定义在TEXT内存段。TEXT语句后的指令一般对应函数的实现，但是对于TEXT指令本身来说并不关心后面是否有指令。因此TEXT和LABEL定义的符号是类似的，区别只是LABEL是用于跳转标号，但是本质上他们都是通过标识符映射一个内存地址。

函数的定义的语法如下：

TEXT symbol(SB), [flags,] $framesize[-argsize]

函数的定义部分由5个部分组成：TEXT指令、函数名、可选的flags标志、函数帧大小和可选的函数参数大小。

其中TEXT用于定义函数符号，函数名中当前包的路径可以省略。函数的名字后面是(SB)，表示是函数名符号相对于SB伪寄存器的偏移量，二者组合在一起最终是绝对地址。作为全局的标识符的全局变量和全局函数的名字一般都是基于SB伪寄存器的相对地址。标志部分用于指示函数的一些特殊行为，标志在textlags.h文件中定义，常见的NOSPLIT主要用于指示叶子函数不进行栈分裂。framesize部分表示函数的局部变量需要多少栈空间，其中包含调用其它函数时准备调用参数的隐式栈空间。最后是可以省略的参数大小，之所以可以省略是因为编译器可以从Go语言的函数声明中推导出函数参数的大小。

我们首先从一个简单的Swap函数开始。Swap函数用于交互输入的两个参数的顺序，然后通过返回值返回交换了顺序的结果。如果用Go语言中声明Swap函数，大概这样的：

package main  
  
//***go:nosplit***func Swap(a, b int) (int, int)

下面是main包中Swap函数在汇编中两种定义方式：

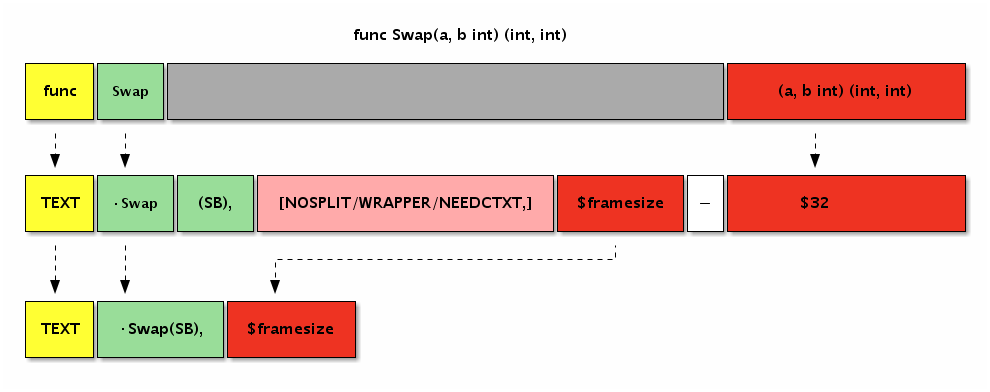
// func Swap(a, b int) (int, int)

TEXT ·Swap(SB), NOSPLIT, $0-32

// func Swap(a, b int) (int, int)

TEXT ·Swap(SB), NOSPLIT, $0

下图是Swap函数几种不同写法的对比关系图：



*图 3-8 函数定义*

第一种是最完整的写法：函数名部分包含了当前包的路径，同时指明了函数的参数大小为32个字节（对应参数和返回值的4个int类型）。第二种写法则比较简洁，省略了当前包的路径和参数的大小。如果有NOSPLIT标注，会禁止汇编器为汇编函数插入栈分裂的代码。NOSPLIT对应Go语言中的//go:nosplit注释。

目前可能遇到的函数标志有NOSPLIT、WRAPPER和NEEDCTXT几个。其中**NOSPLIT**不会生成或包含栈分裂代码，这一般用于没有任何其它函数调用的叶子函数，这样可以适当提高性能。**WRAPPER**标志则表示这个是一个包装函数，在panic或runtime.caller等某些处理函数帧的地方不会增加函数帧计数。最后的**NEEDCTXT**表示需要一个上下文参数，一般用于闭包函数。

需要注意的是函数也没有类型，上面定义的Swap函数签名可以下面任意一种格式：

func Swap(a, b, c int) int

func Swap(a, b, c, d int)

func Swap() (a, b, c, d int)

func Swap() (a []int, d int)

// ...

对于汇编函数来说，只要是函数的名字和参数大小一致就可以是相同的函数了。而且在Go汇编语言中，输入参数和返回值参数是没有任何的区别的。

## 4.2. 函数参数和返回值

对于函数来说，最重要的是函数对外提供的API约定，包含函数的名称、参数和返回值。当这些都确定之后，如何精确计算参数和返回值的大小是第一个需要解决的问题。

比如有一个Swap函数的签名如下：

func Swap(a, b int) (ret0, ret1 int)

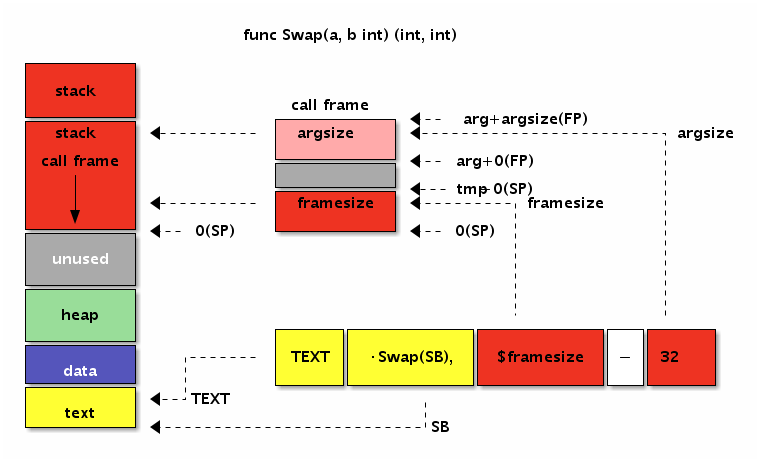
对于这个函数，我们可以轻易看出它需要4个int类型的空间，参数和返回值的大小也就是32个字节：

TEXT ·Swap(SB), $0-32

那么如何在汇编中引用这4个参数呢？为此Go汇编中引入了一个FP伪寄存器，表示函数当前帧的地址，也就是第一个参数的地址。因此我们以通过+0(FP)、+8(FP)、+16(FP)和+24(FP)来分别引用a、b、ret0和ret1四个参数。

但是在汇编代码中，我们并不能直接以+0(FP)的方式来使用参数。为了编写易于维护的汇编代码，Go汇编语言要求，任何通过FP伪寄存器访问的变量必和一个临时标识符前缀组合后才能有效，一般使用参数对应的变量名作为前缀。

下图是Swap函数中参数和返回值在内存中的布局图：



*图 3-9 函数定义*

下面的代码演示了如何在汇编函数中使用参数和返回值：

TEXT ·Swap(SB), $0

MOVQ a+0(FP), AX // AX = a

MOVQ b+8(FP), BX // BX = b

MOVQ BX, ret0+16(FP) // ret0 = BX

MOVQ AX, ret1+24(FP) // ret1 = AX

RET

从代码可以看出a、b、ret0和ret1的内存地址是依次递增的，FP伪寄存器是第一个变量的开始地址。

## 4.3. 参数和返回值的内存布局

如果是参数和返回值类型比较复杂的情况该如何处理呢？下面我们再尝试一个更复杂的函数参数和返回值的计算。比如有以下一个函数：

func Foo(a bool, b int16) (c []byte)

函数的参数有不同的类型，而且返回值中含有更复杂的切片类型。我们该如何计算每个参数的位置和总的大小呢？

其实函数参数和返回值的大小以及对齐问题和结构体的大小和成员对齐问题是一致的，函数的第一个参数和第一个返回值会分别进行一次地址对齐。我们可以用诡代思路将全部的参数和返回值以同样的顺序分别放到两个结构体中，将FP伪寄存器作为唯一的一个指针参数，而每个成员的地址也就是对应原来参数的地址。

用这样的策略可以很容易计算前面的Foo函数的参数和返回值的地址和总大小。为了便于描述我们定义一个Foo\_args\_and\_returns临时结构体类型用于诡代原始的参数和返回值：

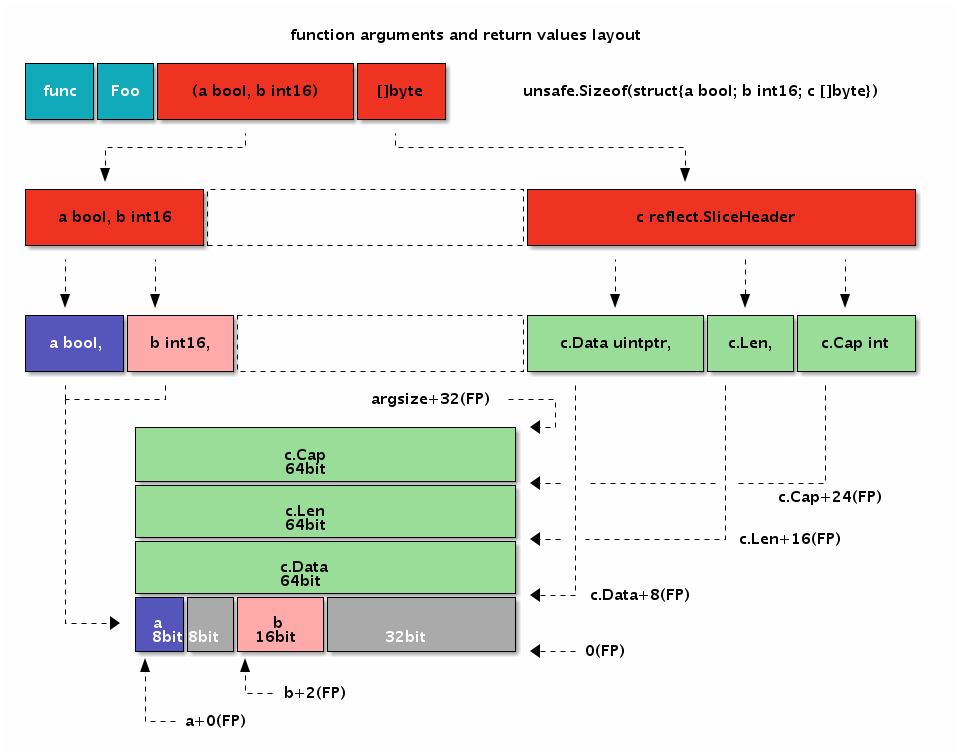
type Foo\_args struct {  
 a bool  
 b int16  
 c []byte  
}  
type Foo\_returns struct {  
 c []byte  
}

然后将Foo原来的参数替换为结构体形式，并且只保留唯一的FP作为参数：

func Foo(FP \*SomeFunc\_args, FP\_ret \*SomeFunc\_returns) {  
 // a = FP + offsetof(&args.a)  
 \_ = unsafe.Offsetof(FP.a) + uintptr(FP) // a  
 // b = FP + offsetof(&args.b)  
  
 // argsize = sizeof(args)  
 argsize = unsafe.Offsetof(FP)  
  
 // c = FP + argsize + offsetof(&return.c)  
 \_ = uintptr(FP) + argsize + unsafe.Offsetof(FP\_ret.c)  
  
 // framesize = sizeof(args) + sizeof(returns)  
 \_ = unsafe.Offsetof(FP) + unsafe.Offsetof(FP\_ret)  
  
 return  
}

代码完全和Foo函数参数的方式类似。唯一的差异是每个函数的偏移量，通过unsafe.Offsetof函数自动计算生成。因为Go结构体中的每个成员已经满足了对齐要求，因此采用通用方式得到每个参数的偏移量也是满足对齐要求的。序言注意的是第一个返回值地址需要重新对齐机器字大小的倍数。

Foo函数的参数和返回值的大小和内存布局：



*图 3-10 函数的参数*

下面的代码演示了Foo汇编函数参数和返回值的定位：

TEXT ·Foo(SB), $0

MOVEQ a+0(FP), AX // a

MOVEQ b+2(FP), BX // b

MOVEQ c\_dat+8\*1(FP), CX // c.Data

MOVEQ c\_len+8\*2(FP), DX // c.Len

MOVEQ c\_cap+8\*3(FP), DI // c.Cap

RET

其中a和b参数之间出现了一个字节的空洞，b和c之间出现了4个字节的空洞。出现空洞的原因是要保证每个参数变量地址都要对齐到相应的倍数。

## 4.4. 函数中的局部变量

从Go语言函数角度讲，局部变量是函数内明确定义的变量，同时也包含函数的参数和返回值变量。但是从Go汇编角度看，局部变量是指函数运行时，在当前函数栈帧所对应的内存内的变量，不包含函数的参数和返回值（因为访问方式有差异）。函数栈帧的空间主要由函数参数和返回值、局部变量和被调用其它函数的参数和返回值空间组成。为了便于理解，我们可以将汇编函数的局部变量类比为Go语言函数中显式定义的变量，不包含参数和返回值部分。

为了便于访问局部变量，Go汇编语言引入了伪SP寄存器，对应当前栈帧的底部。因为在当前栈帧时栈的底部是固定不变的，因此局部变量的相对于伪SP的偏移量也就是固定的，这可以简化局部变量的维护工作。SP真伪寄存器的区分只有一个原则：如果使用SP时有一个临时标识符前缀就是伪SP，否则就是真SP寄存器。比如a(SP)和b+8(SP)有a和b临时前缀，这里都是伪SP，而前缀部分一般用于表示局部变量的名字。而(SP)和+8(SP)没有临时标识符作为前缀，它们都是真SP寄存器。

在X86平台，函数的调用栈是从高地址向低地址增长的，因此伪SP寄存器对应栈帧的底部其实是对应更大的地址。当前栈的顶部对应真实存在的SP寄存器，对应当前函数栈帧的栈顶，对应更小的地址。如果整个内存用Memory数组表示，那么Memory[0(SP):end-0(SP)]就是对应当前栈帧的切片，其中开始位置是真SP寄存器，结尾部分是伪SP寄存器。真SP寄存器一般用于表示调用其它函数时的参数和返回值，真SP寄存器对应内存较低的地址，所以被访问变量的偏移量是正数；而伪SP寄存器对应高地址，对应的局部变量的偏移量都是负数。

为了便于对比，我们将前面Foo函数的参数和返回值变量改成局部变量：

func Foo() {  
 var c []byte  
 var b int16  
 var a bool  
}

然后通过汇编语言重新实现Foo函数，并通过伪SP来定位局部变量：

TEXT ·Foo(SB), $32-0

MOVQ a-32(SP), AX // a

MOVQ b-30(SP), BX // b

MOVQ c\_data-24(SP), CX // c.Data

MOVQ c\_len-16(SP), DX // c.Len

MOVQ c\_cap-8(SP), DI // c.Cap

RET

Foo函数有3个局部变量，但是没有调用其它的函数，因为对齐和填充的问题导致函数的栈帧大小为32个字节。因为Foo函数没有参数和返回值，因此参数和返回值大小为0个字节，当然这个部分可以省略不写。而局部变量中先定义的变量c离伪SP寄存器对应的地址最远，最后定义的变量a离伪SP寄存器最近。有两个因素导致出现这种逆序的结果：一个从Go语言函数角度理解，先定义的c变量地址要比后定义的变量的地址更小；另一个是伪SP寄存器对应栈帧的底部，而X86中栈是从高向地生长的，所以最先定义有着更小地址的c变量离栈的底部伪SP更远。

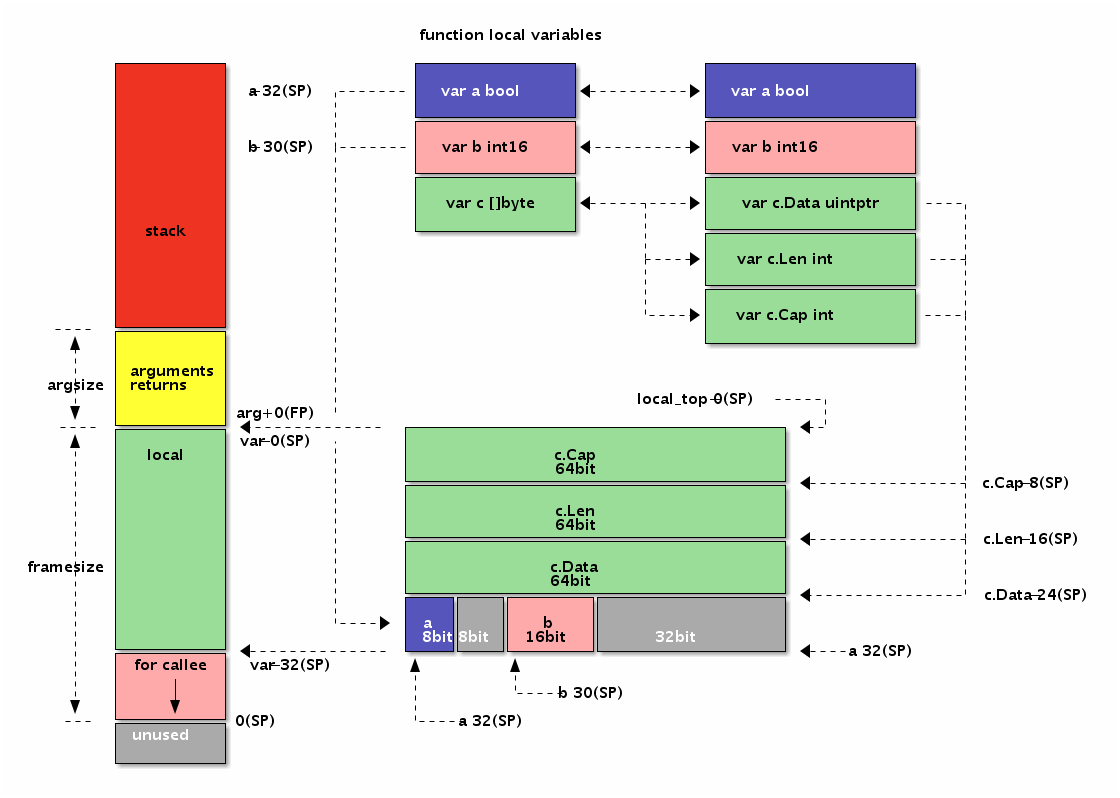
我们同样可以通过结构体来模拟局部变量的布局：

import "unsafe"  
  
func Foo() {  
 var local [1]struct{  
 a bool  
 b int16  
 c []byte  
 }  
 var SP = &local[1];  
  
 \_ = -(unsafe.Sizeof(local)-unsafe.Offsetof(local.a)) + uintptr(&SP) // a  
 \_ = -(unsafe.Sizeof(local)-unsafe.Offsetof(local.b)) + uintptr(&SP) // b  
 \_ = -(unsafe.Sizeof(local)-unsafe.Offsetof(local.c)) + uintptr(&SP) // c  
}

我们将之前的三个局部变量挪到一个结构体中。然后构造一个SP变量对应伪SP寄存器，对应局部变量结构体的顶部。然后根据局部变量总大小和每个变量对应成员的偏移量计算相对于伪SP的距离，最终偏移量是一个负数。

通过这种方式可以处理复杂的局部变量的偏移，同时也能保证每个变量地址的对齐要求。当然，除了地址对齐外，局部变量的布局并没有顺序要求。对于汇编比较熟悉同学可以根据自己的习惯组织变量的布局。

下面是Foo函数的局部变量的大小和内存布局：



*图 3-11 函数的局部变量*

从图中可以看出Foo函数局部变量和前一个例子中参数和返回值的内存布局是完全一样的，这也是我们故意设计的结果。但是参数和返回值是通过伪FP寄存器定位的，FP寄存器对应第一个参数的开始地址（第一个参数地址较低），因此每个变量的偏移量是正数。而局部变量是通过伪SP寄存器定位的，而伪SP寄存器对应的是第一个局部变量的结束地址（第一个局部变量地址较大），因此每个局部变量的偏移量都是负数。

## 4.5. 调用其它函数

常见的用Go汇编实现的函数都是叶子函数，也就是被其它函数调用的函数，但是很少调用其它函数。这主要是因为叶子函数比较简单，可以简化汇编函数的编写；同时一般性能或特性的瓶颈也处于叶子函数。但是能够调用其它函数和能够被其它函数调用同样重要，否则Go汇编就不是一个完整的汇编语言。

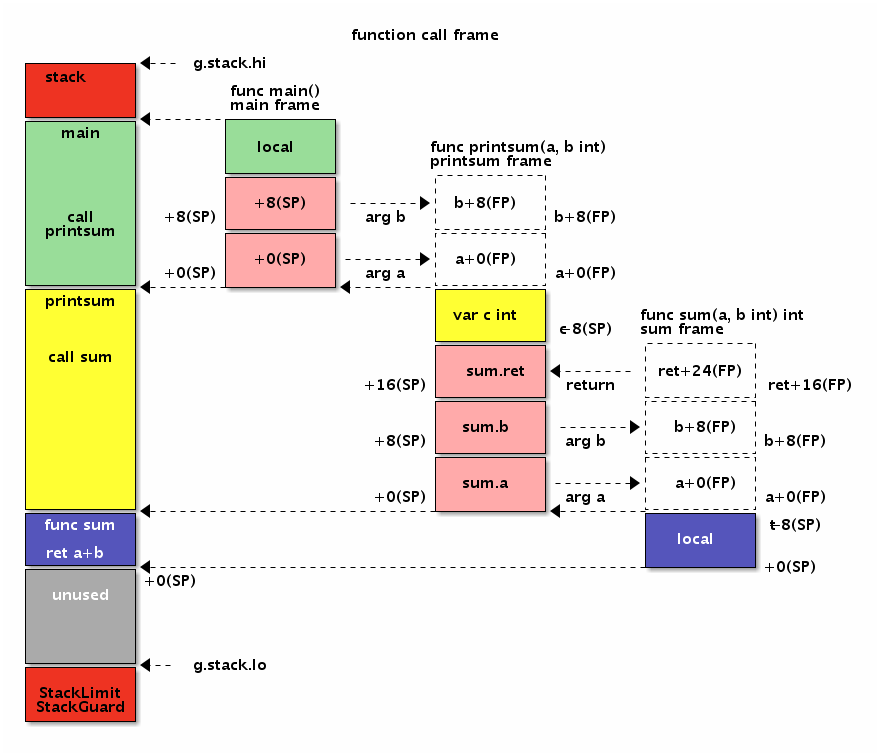
在前文中我们已经学习了一些汇编实现的函数参数和返回值处理的规则。那么一个显然的问题是，汇编函数的参数是从哪里来的？答案同样明显，被调用函数的参数是由调用方准备的：调用方在栈上设置好空间和数据后调用函数，被调用方在返回前将返回值放在对应的位置，函数通过RET指令返回调用方函数之后，调用方再从返回值对应的栈内存位置取出结果。Go语言函数的调用参数和返回值均是通过栈传输的，这样做的优点是函数调用栈比较清晰，缺点是函数调用有一定的性能损耗（Go编译器是通过函数内联来缓解这个问题的影响）。

为了便于展示，我们先使用Go语言来构造三个逐级调用的函数：

func main() {  
 printsum(1, 2)  
}  
  
func printsum(a, b int) {  
 var ret = sum(a, b)  
 println(ret)  
}  
  
func sum(a, b int) int {  
 return a+b  
}

其中main函数通过字面值常量直接调用printsum函数，printsum函数输出两个整数的和。而printsum函数内部又通过调用sum函数计算两个数的和，并最终调用打印函数进行输出。因为printsum既是被调用函数又是调用函数，所以它是我们要重点分析的函数。

下图展示了三个函数逐级调用时内存中函数参数和返回值的布局：



*图 3-12 函数帧*

为了便于理解，我们对真实的内存布局进行了简化。要记住的是调用函数时，被调用函数的参数和返回值内存空间都必须由调用者提供。因此函数的局部变量和为调用其它函数准备的栈空间总和就确定了函数帧的大小。调用其它函数前调用方要选择保存相关寄存器到栈中，并在调用函数返回后选择要恢复的寄存器进行保存。最终通过CALL指令调用函数的过程和调用我们熟悉的调用println函数输出的过程类似。

Go语言中函数调用是一个复杂的问题，因为Go函数不仅仅要了解函数调用参数的布局，还会涉及到栈的跳转，栈上局部变量的生命周期管理。本节只是简单了解函数调用参数的布局规则，在后续的章节中会更详细的讨论函数的细节。

## 4.6. 宏函数

宏函数并不是Go汇编语言所定义，而是Go汇编引入的预处理特性自带的特性。

在C语言中我们可以通过带参数的宏定义一个交换2个数的宏函数：

#define SWAP(x, y) do{ int t = x; x = y; y = t; }while(0)

我们可以用类似的方式定义一个交换两个寄存器的宏：

#define SWAP(x, y, t) MOVQ x, t; MOVQ y, x; MOVQ t, y

因为汇编语言中无法定义临时变量，我们增加一个参数用于临时寄存器。下面是通过SWAP宏函数交换AX和BX寄存器的值，然后返回结果：

// func Swap(a, b int) (int, int)

TEXT ·Swap(SB), $0-32

MOVQ a+0(FP), AX // AX = a

MOVQ b+8(FP), BX // BX = b

SWAP(AX, BX, CX) // AX, BX = b, a

MOVQ AX, ret0+16(FP) // return

MOVQ BX, ret1+24(FP) //

RET

因为预处理器可以通过条件编译针对不同的平台定义宏的实现，这样可以简化平台带来的差异。

# 5. 控制流

程序主要有顺序、分支和循环几种执行流程。本节主要讨论如何将Go语言的控制流比较直观地转译为汇编程序，或者说如何以汇编思维来编写Go语言代码。

## 5.1. 顺序执行

顺序执行是我们比较熟悉的工作模式，类似俗称流水账编程。所有不含分支、循环和goto语句，并且没有递归调用的Go函数一般都是顺序执行的。

比如有如下顺序执行的代码：

func main() {  
 var a = 10  
 println(a)  
  
 var b = (a+a)\*a  
 println(b)  
}

我们尝试用Go汇编的思维改写上述函数。因为X86指令中一般只有2个操作数，因此在用汇编改写时要求出现的变量表达式中最多只能有一个运算符。同时对于一些函数调用，也需要用汇编中可以调用的函数来改写。

第一步改写依然是使用Go语言，只不过是用汇编的思维改写：

import "runtime"  
  
func main() {  
 var a, b int  
  
 a = 10  
 runtime.printint(a)  
 runtime.printnl()  
  
 b = a  
 b += b  
 b \*= a  
 runtime.printint(b)  
 runtime.printnl()  
}

首选模仿C语言的处理方式在函数入口处声明全部的局部变量。然后根据MOV、ADD、MUL等指令的风格，将之前的变量表达式展开为用=、+=和\*=几种运算表达的多个指令。最后用runtime包内部的printint和printnl函数代替之前的println函数输出结果。

经过用汇编的思维改写过后，上述的Go函数虽然看着繁琐了一点，但是还是比较容易理解的。下面我们进一步尝试将改写后的函数继续转译为汇编函数：

TEXT ·main(SB), $24-0

MOVQ $0, a-8\*2(SP) // a = 0

MOVQ $0, b-8\*1(SP) // b = 0

// 将新的值写入a对应内存

MOVQ $10, AX // AX = 10

MOVQ AX, a-8\*2(SP) // a = AX

// 以a为参数调用函数

MOVQ AX, 0(SP)

CALL runtime·printint(SB)

CALL runtime·printnl(SB)

// 函数调用后, AX/BX 寄存器可能被污染, 需要重新加载

MOVQ a-8\*2(SP), AX // AX = a

MOVQ b-8\*1(SP), BX // BX = b

// 计算b值, 并写入内存

MOVQ AX, BX // BX = AX // b = a

ADDQ BX, BX // BX += BX // b += a

IMULQ AX, BX // BX \*= AX // b \*= a

MOVQ BX, b-8\*1(SP) // b = BX

// 以b为参数调用函数

MOVQ BX, 0(SP)

CALL runtime·printint(SB)

CALL runtime·printnl(SB)

RET

汇编实现main函数的第一步是要计算函数栈帧的大小。因为函数内有a、b两个int类型变量，同时调用的runtime·printint函数参数是一个int类型并且没有返回值，因此main函数的栈帧是3个int类型组成的24个字节的栈内存空间。

在函数的开始处先将变量初始化为0值，其中a-8\*2(SP)对应a变量、a-8\*1(SP)对应b变量（因为a变量先定义，因此a变量的地址更小）。

然后给a变量分配一个AX寄存器，并且通过AX寄存器将a变量对应的内存设置为10，AX也是10。为了输出a变量，需要将AX寄存器的值放到0(SP)位置，这个位置的变量将在调用runtime·printint函数时作为它的参数被打印。因为我们之前已经将AX的值保存到a变量内存中了，因此在调用函数前并不需要再进行寄存器的备份工作。

在调用函数返回之后，全部的寄存器将被视为可能被调用的函数修改，因此我们需要从a、b对应的内存中重新恢复寄存器AX和BX。然后参考上面Go语言中b变量的计算方式更新BX对应的值，计算完成后同样将BX的值写入到b对应的内存。

需要说明的是，上面的代码中IMULQ AX, BX使用了IMULQ指令来计算乘法。没有使用MULQ指令的原因是MULQ指令默认使用AX保存结果。读者可以自己尝试用MULQ指令改写上述代码。

最后以b变量作为参数再次调用runtime·printint函数进行输出工作。所有的寄存器同样可能被污染，不过main函数马上就返回了，因此不再需要恢复AX、BX等寄存器了。

重新分析汇编改写后的整个函数会发现里面很多的冗余代码。我们并不需要a、b两个临时变量分配两个内存空间，而且也不需要在每个寄存器变化之后都要写入内存。下面是经过优化的汇编函数：

TEXT ·main(SB), $16-0

// var temp int

// 将新的值写入a对应内存

MOVQ $10, AX // AX = 10

MOVQ AX, temp-8(SP) // temp = AX

// 以a为参数调用函数

CALL runtime·printint(SB)

CALL runtime·printnl(SB)

// 函数调用后, AX 可能被污染, 需要重新加载

MOVQ temp-8\*1(SP), AX // AX = temp

// 计算b值, 不需要写入内存

MOVQ AX, BX // BX = AX // b = a

ADDQ BX, BX // BX += BX // b += a

IMULQ AX, BX // BX \*= AX // b \*= a

// ...

首先是将main函数的栈帧大小从24字节减少到16字节。唯一需要保存的是a变量的值，因此在调用runtime·printint函数输出时全部的寄存器都可能被污染，我们无法通过寄存器备份a变量的值，只有在栈内存中的值才是安全的。然后在BX寄存器并不需要保存到内存。其它部分的代码基本保持不变。

## 5.2. if/goto跳转

Go语言刚刚开源的时候并没有goto语句，后来Go语言虽然增加了goto语句，但是并不推荐在编程中使用。有一个和cgo类似的原则：如果可以不使用goto语句，那么就不要使用goto语句。Go语言中的goto语句是有严格限制的：它无法跨越代码块，并且在被跨越的代码中不能含有变量定义的语句。虽然Go语言不推荐goto语句，但是goto确实每个汇编语言码农的最爱。因为goto近似等价于汇编语言中的无条件跳转指令JMP，配合if条件goto就组成了有条件跳转指令，而有条件跳转指令正是构建整个汇编代码控制流的基石。

为了便于理解，我们用Go语言构造一个模拟三元表达式的If函数：

func If(ok bool, a, b int) int {  
 if ok { return a } else { return b }  
}

比如求两个数最大值的三元表达式(a>b)?a:b用If函数可以这样表达：If(a>b, a, b)。因为语言的限制，用来模拟三元表达式的If函数不支持泛型（可以将a、b和返回类型改为空接口，不过使用会繁琐一些）。

这个函数虽然看似只有简单的一行，但是包含了if分支语句。在改用汇编实现前，我们还是先用汇编的思维来重新审视If函数。在改写时同样要遵循每个表达式只能有一个运算符的限制，同时if语句的条件部分必须只有一个比较符号组成，if语句的body部分只能是一个goto语句。

用汇编思维改写后的If函数实现如下：

func If(ok int, a, b int) int {  
 if ok == 0 { goto L }  
 return a  
L:  
 return b  
}

因为汇编语言中没有bool类型，我们改用int类型代替bool类型（真实的汇编是用byte表示bool类型，可以通过MOVBQZX指令加载byte类型的值，这里做了简化处理）。当ok参数非0时返回变量a，否则返回变量b。我们将ok的逻辑反转下：当ok参数为0时，表示返回b，否则返回变量a。在if语句中，当ok参数为0时goto到L标号指定的语句，也就是返回变量b。如果if条件不满足，也就是ok参数非0，执行后面的语句返回变量a。

上述函数的实现已经非常接近汇编语言，下面是改为汇编实现的代码：

TEXT ·If(SB), NOSPLIT, $0-32

MOVQ ok+8\*0(FP), CX // ok

MOVQ a+8\*1(FP), AX // a

MOVQ b+8\*2(FP), BX // b

CMPQ CX, $0 // test ok

JZ L // if ok == 0, goto L

MOVQ AX, ret+24(FP) // return a

RET

L:

MOVQ BX, ret+24(FP) // return b

RET

首先是将三个参数加载到寄存器中，ok参数对应CX寄存器，a、b分别对应AX、BX寄存器。然后使用CMPQ比较指令将CX寄存器和常数0进行比较。如果比较的结果为0，那么下一条JZ为0时跳转指令将跳转到L标号对应的语句，也就是返回变量b的值。如果比较的结果不为0，那么JZ指令将没有效果，继续执行后面的指令，也就是返回变量a的值。

在跳转指令中，跳转的目标一般是通过一个标号表示。不过在有些通过宏实现的函数中，更希望通过相对位置跳转，这时候可以通过PC寄存器的偏移量来计算临近跳转的位置。

## 5.3. for循环

Go语言的for循环有多种用法，我们这里只选择最经典的for结构来讨论。经典的for循环由初始化、结束条件、迭代步长三个部分组成，再配合循环体内部的if条件语言，这种for结构可以模拟其它各种循环类型。

基于经典的for循环结构，我们定义一个LoopAdd函数，可以用于计算任意等差数列的和：

func LoopAdd(cnt, v0, step int) int {  
 result := v0  
 for i := 0; i < cnt; i++ {  
 result += step  
 }  
 return result  
}

比如1+2+...+100等差数列可以这样计算LoopAdd(100, 1, 1)，而10+8+...+0等差数列则可以这样计算LoopAdd(5, 10, -2)。在用汇编彻底重写之前先采用前面if/goto类似的技术来改造for循环。

新的LoopAdd函数只有if/goto语句构成：

func LoopAdd(cnt, v0, step int) int {

var i = 0

var result = 0

LOOP\_BEGIN:

result = v0

LOOP\_IF:

if i < cnt { goto LOOP\_BODY }

goto LOOP\_END

LOOP\_BODY

i = i+1

result = result + step

goto LOOP\_IF

LOOP\_END:

return result

}

函数的开头先定义两个局部变量便于后续代码使用。然后将for语句的初始化、结束条件、迭代步长三个部分拆分为三个代码段，分别用LOOP\_BEGIN、LOOP\_IF、LOOP\_BODY三个标号表示。其中LOOP\_BEGIN循环初始化部分只会执行一次，因此该标号并不会被引用，可以省略。最后LOOP\_END语句表示for循环的结束。四个标号分隔出的三个代码段分别对应for循环的初始化语句、循环条件和循环体，其中迭代语句被合并到循环体中了。

下面用汇编语言重新实现LoopAdd函数

#include "textflag.h"

// func LoopAdd(cnt, v0, step int) int

TEXT ·LoopAdd(SB), NOSPLIT, $0-32

MOVQ cnt+0(FP), AX // cnt

MOVQ v0+8(FP), BX // v0/result

MOVQ step+16(FP), CX // step

LOOP\_BEGIN:

MOVQ $0, DX // i

LOOP\_IF:

CMPQ DX, AX // compare i, cnt

JL LOOP\_BODY // if i < cnt: goto LOOP\_BODY

JMP LOOP\_END

LOOP\_BODY:

ADDQ $1, DX // i++

ADDQ CX, BX // result += step

JMP LOOP\_IF

LOOP\_END:

MOVQ BX, ret+24(FP) // return result

RET

其中v0和result变量复用了一个BX寄存器。在LOOP\_BEGIN标号对应的指令部分，用MOVQ将DX寄存器初始化为0，DX对应变量i，循环的迭代变量。在LOOP\_IF标号对应的指令部分，使用CMPQ指令比较DX和AX，如果循环没有结束则跳转到LOOP\_BODY部分，否则跳转到LOOP\_END部分结束循环。在LOOP\_BODY部分，更新迭代变量并且执行循环体中的累加语句，然后直接跳转到LOOP\_IF部分进入下一轮循环条件判断。LOOP\_END标号之后就是返回累加结果的语句。

循环是最复杂的控制流，循环中隐含了分支和跳转语句。掌握了循环的写法基本也就掌握了汇编语言的基础写法。更极客的玩法是通过汇编语言打破传统的控制流，比如跨越多层函数直接返回，比如参考基因编辑的手段直接执行一个从C语言构建的代码片段等。总之掌握规律之后，你会发现其实汇编语言编程会变得异常简单和有趣。

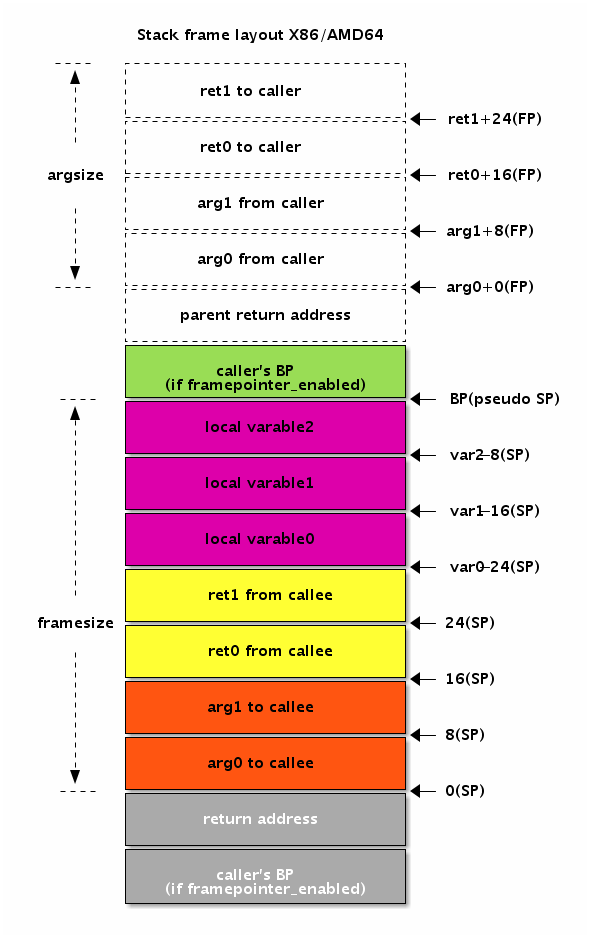
# 6. 再论函数

在前面的章节中我们已经简单讨论过Go的汇编函数，但是那些主要是叶子函数。叶子函数的最大特点是不会调用其他函数，也就是栈的大小是可以预期的，叶子函数也就是可以基本忽略爆栈的问题（如果已经爆了，那也是上级函数的问题）。如果没有爆栈问题，那么也就是不会有栈的分裂问题；如果没有栈的分裂也就不需要移动栈上的指针，也就不会有栈上指针管理的问题。但是是现实中Go语言的函数是可以任意深度调用的，永远不用担心爆栈的风险。那么这些近似黑科技的特性是如何通过低级的汇编语言实现的呢？这些都是本节尝试讨论的问题。

## 6.1. 函数调用规范

在Go汇编语言中CALL指令用于调用函数，RET指令用于从调用函数返回。但是CALL和RET指令并没有处理函数调用时输入参数和返回值的问题。CALL指令类似PUSH IP和JMP somefunc两个指令的组合，首先将当前的IP指令寄存器的值压入栈中，然后通过JMP指令将要调用函数的地址写入到IP寄存器实现跳转。而RET指令则是和CALL相反的操作，基本和POP IP指令等价，也就是将执行CALL指令时保存在SP中的返回地址重新载入到IP寄存器，实现函数的返回。

和C语言函数不同，Go语言函数的参数和返回值完全通过栈传递。下面是Go函数调用时栈的布局图：



*图 3-13 函数调用参数布局*

首先是调用函数前准备的输入参数和返回值空间。然后CALL指令将首先触发返回地址入栈操作。在进入到被调用函数内之后，汇编器自动插入了BP寄存器相关的指令，因此BP寄存器和返回地址是紧挨着的。再下面就是当前函数的局部变量的空间，包含再次调用其它函数需要准备的调用参数空间。被调用的函数执行RET返回指令时，先从栈恢复BP和SP寄存器，接着取出的返回地址跳转到对应的指令执行。

## 6.2. 高级汇编语言

Go汇编语言其实是一种高级的汇编语言。在这里高级一词并没有任何褒义或贬义的色彩，而是要强调Go汇编代码和最终真实执行的代码并不完全等价。Go汇编语言中一个指令在最终的目标代码中可能会被编译为其它等价的机器指令。Go汇编实现的函数或调用函数的指令在最终代码中也会被插入额外的指令。要彻底理解Go汇编语言就需要彻底了解汇编器到底插入了哪些指令。

为了便于分析，我们先构造一个禁止栈分裂的printnl函数。printnl函数内部都通过调用runtime.printnl函数输出换行：

TEXT ·printnl\_nosplit(SB), NOSPLIT, $8

CALL runtime·printnl(SB)

RET

然后通过go tool asm -S main\_amd64.s指令查看编译后的目标代码：

"".printnl\_nosplit STEXT nosplit size=29 args=0xffffffff80000000 locals=0x10

0x0000 00000 (main\_amd64.s:5) TEXT "".printnl\_nosplit(SB), NOSPLIT $16

0x0000 00000 (main\_amd64.s:5) SUBQ $16, SP

0x0004 00004 (main\_amd64.s:5) MOVQ BP, 8(SP)

0x0009 00009 (main\_amd64.s:5) LEAQ 8(SP), BP

0x000e 00014 (main\_amd64.s:6) CALL runtime.printnl(SB)

0x0013 00019 (main\_amd64.s:7) MOVQ 8(SP), BP

0x0018 00024 (main\_amd64.s:7) ADDQ $16, SP

0x001c 00028 (main\_amd64.s:7) RET

输出代码中我们删除了非指令的部分。为了便于讲述，我们将上述代码重新排版，并根据缩进表示相关的功能：

TEXT "".printnl(SB), NOSPLIT, $16

SUBQ $16, SP

MOVQ BP, 8(SP)

LEAQ 8(SP), BP

CALL runtime.printnl(SB)

MOVQ 8(SP), BP

ADDQ $16, SP

RET

第一层是TEXT指令表示函数开始，到RET指令表示函数返回。第二层是SUBQ $16, SP指令为当前函数帧分配16字节的空间，在函数返回前通过ADDQ $16, SP指令回收16字节的栈空间。我们谨慎猜测在第二层是为函数多分配了8个字节的空间。那么为何要多分配8个字节的空间呢？再继续查看第三层的指令：开始部分有两个指令MOVQ BP, 8(SP)和LEAQ 8(SP), BP，首先是将BP寄存器保持到多分配的8字节栈空间，然后将8(SP)地址重新保持到了BP寄存器中；结束部分是MOVQ 8(SP), BP指令则是从栈中恢复之前备份的前BP寄存器的值。最里面第四次层才是我们写的代码，调用runtime.printnl函数输出换行。

如果去掉NOSPILT标志，再重新查看生成的目标代码，会发现在函数的开头和结尾的地方又增加了新的指令。下面是经过缩进格式化的结果：

TEXT "".printnl\_nosplit(SB), $16

L\_BEGIN:

MOVQ (TLS), CX

CMPQ SP, 16(CX)

JLS L\_MORE\_STK

SUBQ $16, SP

MOVQ BP, 8(SP)

LEAQ 8(SP), BP

CALL runtime.printnl(SB)

MOVQ 8(SP), BP

ADDQ $16, SP

L\_MORE\_STK:

CALL runtime.morestack\_noctxt(SB)

JMP L\_BEGIN

RET

其中开头有三个新指令，MOVQ (TLS), CX用于加载g结构体指针，然后第二个指令CMPQ SP, 16(CX)SP栈指针和g结构体中stackguard0成员比较，如果比较的结果小于0则跳转到结尾的L\_MORE\_STK部分。当获取到更多栈空间之后，通过JMP L\_BEGIN指令跳转到函数的开始位置重新进行栈空间的检测。

g结构体在$GOROOT/src/runtime/runtime2.go文件定义，开头的结构成员如下：

type g struct {  
 // Stack parameters.  
 stack stack // offset known to runtime/cgo  
 stackguard0 uintptr // offset known to liblink  
 stackguard1 uintptr // offset known to liblink  
  
 ...  
}

第一个成员是stack类型，表示当前栈的开始和结束地址。stack的定义如下：

// Stack describes a Go execution stack.  
// The bounds of the stack are exactly [lo, hi),  
// with no implicit data structures on either side.  
type stack struct {  
 lo uintptr  
 hi uintptr  
}

在g结构体中的stackguard0成员是出现爆栈前的警戒线。stackguard0的偏移量是16个字节，因此上述代码中的CMPQ SP, 16(AX)表示将当前的真实SP和爆栈警戒线比较，如果超出警戒线则表示需要进行栈扩容，也就是跳转到L\_MORE\_STK。在L\_MORE\_STK标号处，先调用runtime·morestack\_noctxt进行栈扩容，然后又跳回到函数的开始位置，此时此刻函数的栈已经调整了。然后再进行一次栈大小的检测，如果依然不足则继续扩容，直到栈足够大为止。

以上是栈的扩容，但是栈的收缩是在何时处理的呢？我们知道Go运行时会定期进行垃圾回收操作，这其中包含栈的回收工作。如果栈使用到比例小于一定到阈值，则分配一个较小到栈空间，然后将栈上面到数据移动到新的栈中，栈移动的过程和栈扩容的过程类似。

## 6.3. PCDATA和FUNCDATA

Go语言中有个runtime.Caller函数可以获取当前函数的调用者列表。我们可以非常容易在运行时定位每个函数的调用位置，以及函数的调用链。因此在panic异常或用log输出信息时，可以精确定位代码的位置。

比如以下代码可以打印程序的启动流程：

import (  
 "fmt"  
 "runtime"  
)  
  
func main() {  
 for skip := 0; ; skip++ {  
 pc, file, line, ok := runtime.Caller(skip)  
 if !ok {  
 break  
 }  
  
 p := runtime.FuncForPC(pc)  
 fnfile, fnline := p.FileLine(0)  
  
 fmt.Printf("skip = %d, pc = 0x%08X\n", skip, pc)  
 fmt.Printf(" func: file = %s, line = L%03d, name = %s, entry = 0x%08X\n", fnfile, fnline, p.Name(), p.Entry())  
 fmt.Printf(" call: file = %s, line = L%03d\n", file, line)  
 }  
}

其中runtime.Caller先获取当时的PC寄存器值，以及文件和行号。然后根据PC寄存器表示的指令位置，通过runtime.FuncForPC函数获取函数的基本信息。Go语言是如何实现这种特性的呢？

Go语言作为一门静态编译型语言，在执行时每个函数的地址都是固定的，函数的每条指令也是固定的。如果针对每个函数和函数的每个指令生成一个地址表格（也叫PC表格），那么在运行时我们就可以根据PC寄存器的值轻松查询到指令当时对应的函数和位置信息。而Go语言也是采用类似的策略，只不过地址表格经过裁剪，舍弃了不必要的信息。因为要在运行时获取任意一个地址的位置，必然是要有一个函数调用，因此我们只需要为函数的开始和结束位置，以及每个函数调用位置生成地址表格就可以了。同时地址是有大小顺序的，在排序后可以通过只记录增量来减少数据的大小；在查询时可以通过二分法加快查找的速度。

在汇编中有个PCDATA用于生成PC表格，PCDATA的指令用法为：PCDATA tableid, tableoffset。PCDATA有个两个参数，第一个参数为表格的类型，第二个是表格的地址。在目前的实现中，有PCDATA\_StackMapIndex和PCDATA\_InlTreeIndex两种表格类型。两种表格的数据是类似的，应该包含了代码所在的文件路径、行号和函数的信息，只不过PCDATA\_InlTreeIndex用于内联函数的表格。

此外对于汇编函数中返回值包含指针的类型，在返回值指针被初始化之后需要执行一个GO\_RESULTS\_INITIALIZED指令：

#define GO\_RESULTS\_INITIALIZED PCDATA $PCDATA\_StackMapIndex, $1

GO\_RESULTS\_INITIALIZED记录的也是PC表格的信息，表示PC指针越过某个地址之后返回值才完成被初始化的状态。

Go语言二进制文件中除了有PC表格，还有FUNC表格用于记录函数的参数、局部变量的指针信息。FUNCDATA指令和PCDATA的格式类似：FUNCDATA tableid, tableoffset，第一个参数为表格的类型，第二个是表格的地址。目前的实现中定义了三种FUNC表格类型：FUNCDATA\_ArgsPointerMaps表示函数参数的指针信息表，FUNCDATA\_LocalsPointerMaps表示局部指针信息表，FUNCDATA\_InlTree表示被内联展开的指针信息表。通过FUNC表格，Go语言的垃圾回收器可以跟踪全部指针的生命周期，同时根据指针指向的地址是否在被移动的栈范围来确定是否要进行指针移动。

在前面递归函数的例子中，我们遇到一个NO\_LOCAL\_POINTERS宏。它的定义如下：

#define FUNCDATA\_ArgsPointerMaps 0 /\* garbage collector blocks \*/#define FUNCDATA\_LocalsPointerMaps 1#define FUNCDATA\_InlTree 2

#define NO\_LOCAL\_POINTERS FUNCDATA $FUNCDATA\_LocalsPointerMaps, runtime·no\_pointers\_stackmap(SB)

因此NO\_LOCAL\_POINTERS宏表示的是FUNCDATA\_LocalsPointerMaps对应的局部指针表格，而runtime·no\_pointers\_stackmap是一个空的指针表格，也就是表示函数没有指针类型的局部变量。

PCDATA和FUNCDATA的数据一般是由编译器自动生成的，手工编写并不现实。如果函数已经有Go语言声明，那么编译器可以自动输出参数和返回值的指针表格。同时所有的函数调用一般是对应CALL指令，编译器也是可以辅助生成PCDATA表格的。编译器唯一无法自动生成是函数局部变量的表格，因此我们一般要在汇编函数的局部变量中谨慎使用指针类型。

对于PCDATA和FUNCDATA细节感兴趣的同学可以尝试从debug/gosym包入手，参考包的实现和测试代码。

## 6.4. 方法函数

Go语言中方法函数和全局函数非常相似，比如有以下的方法：

type MyInt int  
  
func (v MyInt) Twice() int {  
 return int(v)\*2  
}  
  
func MyInt\_Twice(v MyInt) int {  
 return int(v)\*2  
}

其中MyInt类型的Twice方法和MyInt\_Twice函数的类型是完全一样的，只不过Twice在目标文件中被修饰为main.MyInt.Twice名称。我们可以用汇编实现该方法函数：

// func (v MyInt) Twice() int

TEXT ·MyInt·Twice(SB), NOSPLIT, $0-16

MOVQ a+0(FP), AX // v

ADDQ AX, AX // AX \*= 2

MOVQ AX, ret+8(FP) // return v

RET

不过这只是接收非指针类型的方法函数。现在增加一个接收参数是指针类型的Ptr方法，函数返回传入的指针：

func (p \*MyInt) Ptr() \*MyInt {  
 return p  
}

在目标文件中，Ptr方法名被修饰为main.(\*MyInt).Ptr，也就是对应汇编中的·(\*MyInt)·Ptr。不过在Go汇编语言中，星号和小括弧都无法用作函数名字，也就是无法用汇编直接实现接收参数是指针类型的方法。

在最终的目标文件中的标识符名字中还有很多Go汇编语言不支持的特殊符号（比如type.string."hello"中的双引号），这导致了无法通过手写的汇编代码实现全部的特性。或许是Go语言官方故意限制了汇编语言的特性。

## 6.5. 递归函数: 1到n求和

递归函数是比较特殊的函数，递归函数通过调用自身并且在栈上保存状态，这可以简化很多问题的处理。Go语言中递归函数的强大之处是不用担心爆栈问题，因为栈可以根据需要进行扩容和收缩。

首先通过Go递归函数实现一个1到n的求和函数：

// sum = 1+2+...+n  
// sum(100) = 5050  
func sum(n int) int {  
 if n > 0 { return n+sum(n-1) } else { return 0 }  
}

然后通过if/goto重构上面的递归函数，以便于转义为汇编版本：

func sum(n int) (result int) {

var AX = n

var BX int

if n > 0 { goto L\_STEP\_TO\_END }

goto L\_END

L\_STEP\_TO\_END:

AX -= 1

BX = sum(AX)

AX = n // 调用函数后, AX重新恢复为n

BX += AX

return BX

L\_END:

return 0

}

在改写之后，递归调用的参数需要引入局部变量，保存中间结果也需要引入局部变量。而通过栈来保存中间的调用状态正是递归函数的核心。因为输入参数也在栈上，所以我们可以通过输入参数来保存少量的状态。同时我们模拟定义了AX和BX寄存器，寄存器在使用前需要初始化，并且在函数调用后也需要重新初始化。

下面继续改造为汇编语言版本：

// func sum(n int) (result int)

TEXT ·sum(SB), NOSPLIT, $16-16

MOVQ n+0(FP), AX // n

MOVQ result+8(FP), BX // result

CMPQ AX, $0 // test n - 0

JG L\_STEP\_TO\_END // if > 0: goto L\_STEP\_TO\_END

JMP L\_END // goto L\_STEP\_TO\_END

L\_STEP\_TO\_END:

SUBQ $1, AX // AX -= 1

MOVQ AX, 0(SP) // arg: n-1

CALL ·sum(SB) // call sum(n-1)

MOVQ 8(SP), BX // BX = sum(n-1)

MOVQ n+0(FP), AX // AX = n

ADDQ AX, BX // BX += AX

MOVQ BX, result+8(FP) // return BX

RET

L\_END:

MOVQ $0, result+8(FP) // return 0

RET

在汇编版本函数中并没有定义局部变量，只有用于调用自身的临时栈空间。因为函数本身的参数和返回值有16个字节，因此栈帧的大小也为16字节。L\_STEP\_TO\_END标号部分用于处理递归调用，是函数比较复杂的部分。L\_END用于处理递归终结的部分。

调用sum函数的参数在0(SP)位置，调用结束后的返回值在8(SP)位置。在函数调用之后要需要重新为需要的寄存器注入值，因为被调用的函数内部很可能会破坏了寄存器的状态。同时调用函数的参数值也是不可信任的，输入参数值也可能在被调用函数内部被修改了。

总得来说用汇编实现递归函数和普通函数并没有什么区别，当然是在没有考虑爆栈的前提下。我们的函数应该可以对较小的n进行求和，但是当n大到一定程度，也就是栈达到一定的深度，必然会出现爆栈的问题。爆栈是C语言的特性，不应该在哪怕是Go汇编语言中出现。

Go语言的编译器在生成函数的机器代码时，会在开头插入一小段代码。因为sum函数也需要深度递归调用，因此我们删除了NOSPLIT标志，让汇编器为我们自动生成一个栈扩容的代码：

// func sum(n int) int

TEXT ·sum(SB), $16-16

NO\_LOCAL\_POINTERS

// 原来的代码

除了去掉了NOSPLIT标志，我们还在函数开头增加了一个NO\_LOCAL\_POINTERS语句，该语句表示函数没有局部指针变量。栈的扩容必然要涉及函数参数和局部编指针的调整，如果缺少局部指针信息将导致扩容工作无法进行。不仅仅是栈的扩容需要函数的参数和局部指针标记表格，在GC进行垃圾回收时也将需要。函数的参数和返回值的指针状态可以通过在Go语言中的函数声明中获取，函数的局部变量则需要手工指定。因为手工指定指针表格是一个非常繁琐的工作，因此一般要避免在手写汇编中出现局部指针。

喜欢深究的读者可能会有一个问题：如果进行垃圾回收或栈调整时，寄存器中的指针是如何维护的？前文说过，Go语言的函数调用是通过栈进行传递参数的，并没有使用寄存器传递参数。同时函数调用之后所有的寄存器视为失效。因此在调整和维护指针时，只需要扫描内存中的指针数据，寄存器中的数据在垃圾回收器函数返回后都需要重新加载，因此寄存器是不需要扫描的。

## 6.6. 闭包函数

闭包函数是最强大的函数，因为闭包函数可以捕获外层局部作用域的局部变量，因此闭包函数本身就具有了状态。从理论上来说，全局的函数也是闭包函数的子集，只不过全局函数并没有捕获外层变量而已。

为了理解闭包函数如何工作，我们先构造如下的例子：

package main  
  
func NewTwiceFunClosure(x int) func() int {  
 return func() int {  
 x \*= 2  
 return x  
 }  
}  
  
func main() {  
 fnTwice := NewTwiceFunClosure(1)  
  
 println(fnTwice()) // 1\*2 => 2  
 println(fnTwice()) // 2\*2 => 4  
 println(fnTwice()) // 4\*2 => 8  
}

其中NewTwiceFunClosure函数返回一个闭包函数对象，返回的闭包函数对象捕获了外层的x参数。返回的闭包函数对象在执行时，每次将捕获的外层变量乘以2之后再返回。在main函数中，首先以1作为参数调用NewTwiceFunClosure函数构造一个闭包函数，返回的闭包函数保存在fnTwice闭包函数类型的变量中。然后每次调用fnTwice闭包函数将返回翻倍后的结果，也就是：2，4，8。

上述的代码，从Go语言层面是非常容易理解的。但是闭包函数在汇编语言层面是如何工作的呢？下面我们尝试手工构造闭包函数来展示闭包的工作原理。首先是构造FunTwiceClosure结构体类型，用来表示闭包对象：

import "unsafe"  
  
type FunTwiceClosure struct {  
 F uintptr  
 X int  
}  
  
func NewTwiceFunClosure(x int) func() int {  
 var p = &FunTwiceClosure{  
 F: asmFunTwiceClosureAddr(),  
 X: x,  
 }  
 return ptrToFunc(unsafe.Pointer(p))  
}

FunTwiceClosure结构体包含两个成员，第一个成员F表示闭包函数的函数指令的地址，第二个成员X表示闭包捕获的外部变量。如果闭包函数捕获了多个外部变量，那么FunTwiceClosure结构体也要做相应的调整。然后构造FunTwiceClosure结构体对象，其实也就是闭包函数对象。其中asmFunTwiceClosureAddr函数用于辅助获取闭包函数的函数指令的地址，采用汇编语言实现。最后通过ptrToFunc辅助函数将结构体指针转为闭包函数对象返回，该函数也是通过汇编语言实现。

汇编语言实现了以下三个辅助函数：

func ptrToFunc(p unsafe.Pointer) func() int

func asmFunTwiceClosureAddr() uintptrfunc asmFunTwiceClosureBody() int

其中ptrToFunc用于将指针转化为func() int类型的闭包函数，asmFunTwiceClosureAddr用于返回闭包函数机器指令的开始地址（类似全局函数的地址），asmFunTwiceClosureBody是闭包函数对应的全局函数的实现。

然后用Go汇编语言实现以上三个辅助函数：

#include "textflag.h"

TEXT ·ptrToFunc(SB), NOSPLIT, $0-16

MOVQ ptr+0(FP), AX // AX = ptr

MOVQ AX, ret+8(FP) // return AX

RET

TEXT ·asmFunTwiceClosureAddr(SB), NOSPLIT, $0-8

LEAQ ·asmFunTwiceClosureBody(SB), AX // AX = ·asmFunTwiceClosureBody(SB)

MOVQ AX, ret+0(FP) // return AX

RET

TEXT ·asmFunTwiceClosureBody(SB), NOSPLIT|NEEDCTXT, $0-8

MOVQ 8(DX), AX

ADDQ AX , AX // AX \*= 2

MOVQ AX , 8(DX) // ctx.X = AX

MOVQ AX , ret+0(FP) // return AX

RET

其中·ptrToFunc和·asmFunTwiceClosureAddr函数的实现比较简单，我们不再详细描述。最重要的是·asmFunTwiceClosureBody函数的实现：它有一个NEEDCTXT标志。采用NEEDCTXT标志定义的汇编函数表示需要一个上下文环境，在AMD64环境下是通过DX寄存器来传递这个上下文环境指针，也就是对应FunTwiceClosure结构体的指针。函数首先从FunTwiceClosure结构体对象取出之前捕获的X，将X乘以2之后写回内存，最后返回修改之后的X的值。

如果是在汇编语言中调用闭包函数，也需要遵循同样的流程：首先为构造闭包对象，其中保存捕获的外层变量；在调用闭包函数时首先要拿到闭包对象，用闭包对象初始化DX，然后从闭包对象中取出函数地址并用通过CALL指令调用。

# 7. 汇编语言的威力

汇编语言的真正威力来自两个维度：一是突破框架限制，实现看似不可能的任务；二是突破指令限制，通过高级指令挖掘极致的性能。对于第一个问题，我们将演示如何通过Go汇编语言直接访问系统调用，和直接调用C语言函数。对于第二个问题，我们将演示X64指令中AVX等高级指令的简单用法。

## 7.1. 系统调用

系统调用是操作系统为外提供的公共接口。因为操作系统彻底接管了各种底层硬件设备，因此操作系统提供的系统调用成了实现某些操作的唯一方法。从另一个角度看，系统调用更像是一个RPC远程过程调用，不过信道是寄存器和内存。在系统调用时，我们向操作系统发送调用的编号和对应的参数，然后阻塞等待系统调用地返回。因为涉及到阻塞等待，因此系统调用期间的CPU利用率一般是可以忽略的。另一个和RPC地远程调用类似的地方是，操作系统内核处理系统调用时不会依赖用户的栈空间，一般不会导致爆栈发生。因此系统调用是最简单安全的一种调用了。

系统调用虽然简单，但是它是操作系统对外的接口，因此不同的操作系统调用规范可能有很大地差异。我们先看看Linux在AMD64架构上的系统调用规范，在syscall/asm\_linux\_amd64.s文件中有注释说明：

//  
// System calls for AMD64, Linux  
//  
  
// func Syscall(trap int64, a1, a2, a3 uintptr) (r1, r2, err uintptr);  
// Trap # in AX, args in DI SI DX R10 R8 R9, return in AX DX  
// Note that this differs from "standard" ABI convention, which  
// would pass 4th arg in CX, not R10.

这是syscall.Syscall函数的内部注释，简要说明了Linux系统调用的规范。系统调用的前6个参数直接由DI、SI、DX、R10、R8和R9寄存器传输，结果由AX和DX寄存器返回。macOS等类UINX系统调用的参数传输大多数都采用类似的规则。

macOS的系统调用编号在/usr/include/sys/syscall.h头文件，Linux的系统调用号在/usr/include/asm/unistd.h头文件。虽然在UNIX家族中是系统调用的参数和返回值的传输规则类似，但是不同操作系统提供的系统调用却不是完全相同的，因此系统调用编号也有很大的差异。以UNIX系统中著名的write系统调用为例，在macOS的系统调用编号为4，而在Linux的系统调用编号却是1。

我们将基于write系统调用包装一个字符串输出函数。下面的代码是macOS版本：

// func SyscallWrite\_Darwin(fd int, msg string) int

TEXT ·SyscallWrite\_Darwin(SB), NOSPLIT, $0

MOVQ $(0x2000000+4), AX // #define SYS\_write 4

MOVQ fd+0(FP), DI

MOVQ msg\_data+8(FP), SI

MOVQ msg\_len+16(FP), DX

SYSCALL

MOVQ AX, ret+0(FP)

RET

其中第一个参数是输出文件的文件描述符编号，第二个参数是字符串的头部。字符串头部是由reflect.StringHeader结构定义，第一成员是8字节的数据指针，第二个成员是8字节的数据长度。在macOS系统中，执行系统调用时还需要将系统调用的编号加上0x2000000后再行传入AX。然后再将fd、数据地址和长度作为write系统调用的三个参数输入，分别对应DI、SI和DX三个寄存器。最后通过SYSCALL指令执行系统调用，系统调用返回后从AX获取返回值。

这样我们就基于系统调用包装了一个定制的输出函数。在UNIX系统中，标准输入stdout的文件描述符编号是1，因此我们可以用1作为参数实现字符串的输出：

import "runtime"  
  
func SyscallWrite\_Darwin(fd int, msg string) int  
  
func main() {  
 if runtime.*GOOS* == "darwin" {  
 SyscallWrite\_Darwin(1, "hello syscall!\n")  
 }  
}

如果是Linux系统，只需要将编号改为write系统调用对应的1即可。而Windows的系统调用则有另外的参数传输规则。在X64环境Windows的系统调用参数传输规则和默认的C语言规则非常相似，在后续的直接调用C函数部分再行讨论。

## 7.2. 直接调用C函数

在计算机的发展的过程中，C语言和UNIX操作系统有着不可替代的作用。因此操作系统的系统调用、汇编语言和C语言函数调用规则几个技术是密切相关的。

在X86的32位系统时代，C语言一般默认的是用栈传递参数并用AX寄存器返回结果，称为cdecl调用约定。Go语言函数和cdecl调用约定非常相似，它们都是以栈来传递参数并且返回地址和BP寄存器的布局都是类似的。但是Go语言函数将返回值也通过栈返回，因此Go语言函数可以支持多个返回值。我们可以将Go语言函数看作是没有返回值的C语言函数，同时将Go语言函数中的返回值挪到C语言函数参数的尾部，这样栈不仅仅用于传入参数也用于返回多个结果。

在X64时代，AMD架构增加了8个通用寄存器，为了提高效率C语言也默认改用寄存器来传递参数。在X64系统，默认有System V AMD64 ABI和Microsoft x64两种C语言函数调用规范。其中System V的规范适用于Linux、FreeBSD、macOS等诸多类UNIX系统，而Windows则是用自己特有的调用规范。

在理解了C语言函数的调用规范之后，汇编代码就可以绕过CGO技术直接调用C语言函数。为了便于演示，我们先用C语言构造一个简单的加法函数myadd：

#include <stdint.h>

int64\_t myadd(int64\_t a, int64\_t b) {

return a+b;

}

然后我们需要实现一个asmCallCAdd函数：

func asmCallCAdd(cfun uintptr, a, b int64) int64

因为Go汇编语言和CGO特性不能同时在一个包中使用（因为CGO会调用gcc，而gcc会将Go汇编语言当做普通的汇编程序处理，从而导致错误），我们通过一个参数传入C语言myadd函数的地址。asmCallCAdd函数的其余参数和C语言myadd函数的参数保持一致。

我们只实现System V AMD64 ABI规范的版本。在System V版本中，寄存器可以最多传递六个参数，分别对应DI、SI、DX、CX、R8和R9六个寄存器（如果是浮点数则需要通过XMM寄存器传送），返回值依然通过AX返回。通过对比系统调用的规范可以发现，系统调用的第四个参数是用R10寄存器传递，而C语言函数的第四个参数是用CX传递。

下面是System V AMD64 ABI规范的asmCallCAdd函数的实现：

// System V AMD64 ABI

// func asmCallCAdd(cfun uintptr, a, b int64) int64

TEXT ·asmCallCAdd(SB), NOSPLIT, $0

MOVQ cfun+0(FP), AX // cfun

MOVQ a+8(FP), DI // a

MOVQ b+16(FP), SI // b

CALL AX

MOVQ AX, ret+24(FP)

RET

首先是将第一个参数表示的C函数地址保存到AX寄存器便于后续调用。然后分别将第二和第三个参数加载到DI和SI寄存器。然后CALL指令通过AX中保持的C语言函数地址调用C函数。最后从AX寄存器获取C函数的返回值，并通过asmCallCAdd函数返回。

Win64环境的C语言调用规范类似。不过Win64规范中只有CX、DX、R8和R9四个寄存器传递参数（如果是浮点数则需要通过XMM寄存器传送），返回值依然通过AX返回。虽然是可以通过寄存器传输参数，但是调用这依然要为前四个参数准备栈空间。需要注意的是，Windows x64的系统调用和C语言函数可能是采用相同的调用规则。因为没有Windows测试环境，我们这里就不提供了Windows版本的代码实现了，Windows用户可以自己尝试实现类似功能。

然后我们就可以使用asmCallCAdd函数直接调用C函数了：

/\*  
#include <stdint.h>  
  
int64\_t myadd(int64\_t a, int64\_t b) {  
 return a+b;  
}  
\*/  
import "C"  
  
import (  
 asmpkg "path/to/asm"  
 "runtime"  
 "unsafe"  
)  
  
func main() {  
 if runtime.*GOOS* != "windows" {  
 println(asmpkg.asmCallCAdd(  
 uintptr(unsafe.Pointer(C.myadd)),  
 123, 456,  
 ))  
 }  
}

在上面的代码中，通过C.myadd获取C函数的地址，然后转换为合适的类型再传人asmCallCAdd函数。在这个例子中，汇编函数假设调用的C语言函数需要的栈很小，可以直接复用Go函数中多余的空间。如果C语言函数可能需要较大的栈，可以尝试像CGO那样切换到系统线程的栈上运行。

## 7.3. AVX指令

从Go1.11开始，Go汇编语言引入了AVX512指令的支持。AVX指令集是属于Intel家的SIMD指令集中的一部分。AVX512的最大特点是数据有512位宽度，可以一次计算8个64位数或者是等大小的数据。因此AVX指令可以用于优化矩阵或图像等并行度很高的算法。不过并不是每个X86体系的CPU都支持了AVX指令，因此首要的任务是如何判断CPU支持了哪些高级指令。

在Go语言标准库的internal/cpu包提供了CPU是否支持某些高级指令的基本信息，但是只有标准库才能引用这个包（因为internal路径的限制）。该包底层是通过X86提供的CPUID指令来识别处理器的详细信息。最简便的方法是直接将internal/cpu包克隆一份。不过这个包为了避免复杂的依赖没有使用init函数自动初始化，因此需要根据情况手工调整代码执行doinit函数初始化。

internal/cpu包针对X86处理器提供了以下特性检测：

package cpu  
  
var X86 x86  
  
// The booleans in x86 contain the correspondingly named cpuid feature bit.  
// HasAVX and HasAVX2 are only set if the OS does support XMM and YMM registers  
// in addition to the cpuid feature bit being set.  
// The struct is padded to avoid false sharing.  
type x86 struct {  
 HasAES bool  
 HasADX bool  
 HasAVX bool  
 HasAVX2 bool  
 HasBMI1 bool  
 HasBMI2 bool  
 HasERMS bool  
 HasFMA bool  
 HasOSXSAVE bool  
 HasPCLMULQDQ bool  
 HasPOPCNT bool  
 HasSSE2 bool  
 HasSSE3 bool  
 HasSSSE3 bool  
 HasSSE41 bool  
 HasSSE42 bool  
}

因此我们可以用以下的代码测试运行时的CPU是否支持AVX2指令集：

import (  
 cpu "path/to/cpu"  
)  
  
func main() {  
 if cpu.X86.HasAVX2 {  
 // support AVX2  
 }  
}

AVX512是比较新的指令集，只有高端的CPU才会提供支持。为了主流的CPU也能运行代码测试，我们选择AVX2指令来构造例子。AVX2指令每次可以处理32字节的数据，可以用来提升数据复制的工作的效率。

下面的例子是用AVX2指令复制数据，每次复制数据32字节倍数大小的数据：

// func CopySlice\_AVX2(dst, src []byte, len int)

TEXT ·CopySlice\_AVX2(SB), NOSPLIT, $0

MOVQ dst\_data+0(FP), DI

MOVQ src\_data+24(FP), SI

MOVQ len+32(FP), BX

MOVQ $0, AX

LOOP:

VMOVDQU 0(SI)(AX\*1), Y0

VMOVDQU Y0, 0(DI)(AX\*1)

ADDQ $32, AX

CMPQ AX, BX

JL LOOP

RET

其中VMOVDQU指令先将0(SI)(AX\*1)地址开始的32字节数据复制到Y0寄存器中，然后再复制到0(DI)(AX\*1)对应的目标内存中。VMOVDQU指令操作的数据地址可以不用对齐。

AVX2共有16个Y寄存器，每个寄存器有256bit位。如果要复制的数据很多，可以多个寄存器同时复制，这样可以利用更高效的流水特性优化性能。

# 8. 例子：Goroutine ID

在操作系统中，每个进程都会有一个唯一的进程编号，每个线程也有自己唯一的线程编号。同样在Go语言中，每个Goroutine也有自己唯一的Go程编号，这个编号在panic等场景下经常遇到。虽然Goroutine有内在的编号，但是Go语言却刻意没有提供获取该编号的接口。本节我们尝试通过Go汇编语言获取Goroutine ID。

## 8.1. 故意设计没有goid

根据官方的相关资料显示，Go语言刻意没有提供goid的原因是为了避免被滥用。因为大部分用户在轻松拿到goid之后，在之后的编程中会不自觉地编写出强依赖goid的代码。强依赖goid将导致这些代码不好移植，同时也会导致并发模型复杂化。同时，Go语言中可能同时存在海量的Goroutine，但是每个Goroutine何时被销毁并不好实时监控，这也会导致依赖goid的资源无法很好地自动回收（需要手工回收）。不过如果你是Go汇编语言用户，则完全可以忽略这些借口。

## 8.2. 纯Go方式获取goid

为了便于理解，我们先尝试用纯Go的方式获取goid。使用纯Go的方式获取goid的方式虽然性能较低，但是代码有着很好的移植性，同时也可以用于测试验证其它方式获取的goid是否正确。

每个Go语言用户应该都知道panic函数。调用panic函数将导致Goroutine异常，如果panic在传递到Goroutine的根函数还没有被recover函数处理掉，那么运行时将打印相关的异常和栈信息并退出Goroutine。

下面我们构造一个简单的例子，通过panic来输出goid：

package main  
func main() {  
 panic("goid")  
}

运行后将输出以下信息：

panic: goid

goroutine 1 [running]:

main.main()

/path/to/main.go:4 +0x40

我们可以猜测Panic输出信息goroutine 1 [running]中的1就是goid。但是如何才能在程序中获取panic的输出信息呢？其实上述信息只是当前函数调用栈帧的文字化描述，runtime.Stack函数提供了获取该信息的功能。

我们基于runtime.Stack函数重新构造一个例子，通过输出当前栈帧的信息来输出goid：

package main  
import "runtime"  
func main() {  
 var buf = make([]byte, 64)  
 var stk = buf[:runtime.Stack(buf, false)]  
 print(string(stk))  
}

运行后将输出以下信息：

goroutine 1 [running]:

main.main()

/path/to/main.g

因此从runtime.Stack获取的字符串中就可以很容易解析出goid信息：

package main  
  
import (  
 "fmt"  
 "runtime"  
 "strconv"  
 "strings"  
)  
  
func GetGoid() int64 {  
 var (  
 buf [64]byte  
 n = runtime.Stack(buf[:], false)  
 stk = strings.TrimPrefix(string(buf[:n]), "goroutine ")  
 )  
  
 idField := strings.Fields(stk)[0]  
 id, err := strconv.Atoi(idField)  
 if err != nil {  
 panic(fmt.Errorf("can not get goroutine id: %v", err))  
 }  
  
 return int64(id)  
}

GetGoid函数的细节我们不再赘述。需要补充说明的是runtime.Stack函数不仅仅可以获取当前Goroutine的栈信息，还可以获取全部Goroutine的栈信息（通过第二个参数控制）。同时在Go语言内部的 [net/http2.curGoroutineID](https://github.com/golang/net/blob/master/http2/gotrack.go" \t "http://books.studygolang.com/advanced-go-programming-book/ch3-asm/_blank) 函数正是采用类似方式获取的goid。

## 8.3. 从g结构体获取goid

根据官方的Go汇编语言文档，每个运行的Goroutine结构的g指针保存在当前运行Goroutine的系统线程的局部存储TLS中。可以先获取TLS线程局部存储，然后再从TLS中获取g结构的指针，最后从g结构中取出goid。

下面是参考runtime包中定义的get\_tls宏获取g指针：

get\_tls(CX)

MOVQ g(CX), AX // Move g into AX.

其中get\_tls是一个宏函数，在 [runtime/go\_tls.h](https://github.com/golang/go/blob/master/src/runtime/go_tls.h" \t "http://books.studygolang.com/advanced-go-programming-book/ch3-asm/_blank) 头文件中定义。

对于AMD64平台，get\_tls宏函数定义如下：

#ifdef GOARCH\_amd64

#define get\_tls(r) MOVQ TLS, r

#define g(r) 0(r)(TLS\*1)

#endif

将get\_tls宏函数展开之后，获取g指针的代码如下：

MOVQ TLS, CX

MOVQ 0(CX)(TLS\*1), AX

其实TLS类似线程局部存储的地址，地址对应的内存里的数据才是g指针。我们还可以更直接一点:

MOVQ (TLS), AX

基于上述方法可以包装一个getg函数，用于获取g指针：

// func getg() unsafe.Pointer

TEXT ·getg(SB), NOSPLIT, $0-8

MOVQ (TLS), AX

MOVQ AX, ret+0(FP)

RET

然后在Go代码中通过goid成员在g结构体中的偏移量来获取goid的值：

import "unsafe"  
  
const *g\_goid\_offset* = 152 // Go1.10  
  
func GetGroutineId() int64 {  
 g := getg()  
 p := (\*int64)(unsafe.Pointer(uintptr(g) + *g\_goid\_offset*))  
 return \*p  
}

其中 g\_goid\_offset 是 goid 成员的偏移量，g 结构参考 [runtime/runtime2.go](https://github.com/golang/go/blob/master/src/runtime/runtime2.go" \t "http://books.studygolang.com/advanced-go-programming-book/ch3-asm/_blank)。

在Go1.10版本，goid的偏移量是152字节。因此上述代码只能正确运行在goid偏移量也是152字节的Go版本中。根据汤普森大神的神谕，枚举和暴力穷举是解决一切疑难杂症的万金油。我们也可以将goid的偏移保存到表格中，然后根据Go版本号查询goid的偏移量。

下面是改进后的代码：

import (  
 "runtime"  
 "strings"  
)  
  
var offsetDictMap = map[string]int64{  
 "go1.10": 152,  
 "go1.9": 152,  
 "go1.8": 192,  
}  
  
var g\_goid\_offset = func() int64 {  
 goversion := runtime.Version()  
 for key, off := range offsetDictMap {  
 if goversion == key || strings.HasPrefix(goversion, key) {  
 return off  
 }  
 }  
 panic("unsupport go verion:"+goversion)  
}()

现在的goid偏移量已经终于可以自动适配已经发布的Go语言版本。

## 8.4. 获取g结构体对应的接口对象

枚举和暴力穷举虽然够直接，但是对于正在开发中的未发布的Go版本支持并不好，我们无法提前知晓开发中的某个版本的goid成员的偏移量。

如果是在runtime包内部，我们可以通过unsafe.OffsetOf(g.goid)直接获取成员的偏移量。也可以通过反射获取g结构体的类型，然后通过类型查询某个成员的偏移量。因为g结构体是一个内部类型，Go代码无法从外部包获取g结构体的类型信息。但是在Go汇编语言中，我们是可以看到全部的符号的，因此理论上我们也可以获取g结构体的类型信息。

在任意的类型被定义之后，Go语言都会为该类型生成对应的类型信息。比如g结构体会生成一个type·runtime·g标识符表示g结构体的值类型信息，同时还有一个type·\*runtime·g标识符表示指针类型的信息。如果g结构体带有方法，那么同时还会生成go.itab.runtime.g和go.itab.\*runtime.g类型信息，用于表示带方法的类型信息。

如果我们能够拿到表示g结构体类型的type·runtime·g和g指针，那么就可以构造g对象的接口。下面是改进的getg函数，返回g指针对象的接口：

// func getg() interface{}

TEXT ·getg(SB), NOSPLIT, $32-16

// get runtime.g

MOVQ (TLS), AX

// get runtime.g type

MOVQ $type·runtime·g(SB), BX

// convert (\*g) to interface{}

MOVQ AX, 8(SP)

MOVQ BX, 0(SP)

CALL runtime·convT2E(SB)

MOVQ 16(SP), AX

MOVQ 24(SP), BX

// return interface{}

MOVQ AX, ret+0(FP)

MOVQ BX, ret+8(FP)

RET

其中AX寄存器对应g指针，BX寄存器对应g结构体的类型。然后通过runtime·convT2E函数将类型转为接口。因为我们使用的不是g结构体指针类型，因此返回的接口表示的g结构体值类型。理论上我们也可以构造g指针类型的接口，但是因为Go汇编语言的限制，我们无法使用type·\*runtime·g标识符。

基于g返回的接口，就可以容易获取goid了：

import "reflect"  
  
func GetGoid() int64 {  
 g := getg()  
 gid := reflect.ValueOf(g).FieldByName("goid").Int()  
 return goid  
}

上述代码通过反射直接获取goid，理论上只要反射的接口和goid成员的名字不发生变化，代码都可以正常运行。经过实际测试，以上的代码可以在Go1.8、Go1.9和Go1.10版本中正确运行。乐观推测，如果g结构体类型的名字不发生变化，Go语言反射的机制也不发生变化，那么未来Go语言版本应该也是可以运行的。

反射虽然具备一定的灵活性，但是反射的性能一直是被大家诟病的地方。一个改进的思路是通过反射获取goid的偏移量，然后通过g指针和偏移量获取goid，这样反射只需要在初始化阶段执行一次。

下面是g\_goid\_offset变量的初始化代码：

import "reflect"  
  
var g\_goid\_offset uintptr = func() uintptr {  
 g := GetGroutine()  
 if f, ok := reflect.TypeOf(g).FieldByName("goid"); ok {  
 return f.Offset  
 }  
 panic("can not find g.goid field")  
}()

有了正确的goid偏移量之后，采用前面讲过的方式获取goid：

import (  
 "unsafe"  
)  
  
func GetGroutineId() int64 {  
 g := getg()  
 p := (\*int64)(unsafe.Pointer(uintptr(g) + g\_goid\_offset))  
 return \*p  
}

至此我们获取goid的实现思路已经足够完善了，不过汇编的代码依然有严重的安全隐患。

虽然getg函数是用NOSPLIT标志声明的禁止栈分裂的函数类型，但是getg内部又调用了更为复杂的runtime·convT2E函数。runtime·convT2E函数如果遇到栈空间不足，可能触发栈分裂的操作。而栈分裂时，GC将要挪动栈上所有函数的参数和返回值和局部变量中的栈指针。但是我们的getg函数并没有提供局部变量的指针信息。

下面是改进后的getg函数的完整实现：

// func getg() interface{}

TEXT ·getg(SB), NOSPLIT, $32-16

NO\_LOCAL\_POINTERS

MOVQ $0, ret\_type+0(FP)

MOVQ $0, ret\_data+8(FP)

GO\_RESULTS\_INITIALIZED

// get runtime.g

MOVQ (TLS), AX

// get runtime.g type

MOVQ $type·runtime·g(SB), BX

// convert (\*g) to interface{}

MOVQ AX, 8(SP)

MOVQ BX, 0(SP)

CALL runtime·convT2E(SB)

MOVQ 16(SP), AX

MOVQ 24(SP), BX

// return interface{}

MOVQ AX, ret\_type+0(FP)

MOVQ BX, ret\_data+8(FP)

RET

其中NO\_LOCAL\_POINTERS表示函数没有局部指针变量。同时对返回的接口进行零值初始化，初始化完成后通过GO\_RESULTS\_INITIALIZED告知GC。这样可以在保证栈分裂时，GC能够正确处理返回值和局部变量中的指针。

## 8.5. goid的应用: 局部存储

有了goid之后，构造Goroutine局部存储就非常容易了。我们可以定义一个gls包提供goid的特性：

package gls  
  
import "sync"  
  
var gls struct {  
 m map[int64]map[interface{}]interface{}  
 sync.Mutex  
}  
  
func init() {  
 gls.m = make(map[int64]map[interface{}]interface{})  
}

gls包变量简单包装了map，同时通过sync.Mutex互斥量支持并发访问。

然后定义一个getMap内部函数，用于获取每个Goroutine字节的map：

func init() {  
 gls.m = make(map[int64]map[interface{}]interface{})  
}  
  
func getMap() map[interface{}]interface{} {  
 gls.Lock()  
 defer gls.Unlock()  
  
 goid := GetGoid()  
 if m, \_ := gls.m[goid]; m != nil {  
 return m  
 }  
  
 m := make(map[interface{}]interface{})  
 gls.m[goid] = m  
 return m  
}

获取到Goroutine私有的map之后，就是正常的增、删、改操作接口了：

func Get(key interface{}) interface{} {  
 return getMap()[key]  
}  
func Put(key interface{}, v interface{}) {  
 getMap()[key] = v  
}  
func Delete(key interface{}) {  
 delete(getMap(), key)  
}

最后我们再提供一个Clean函数，用于释放Goroutine对应的map资源：

func Clean() {  
 gls.Lock()  
 defer gls.Unlock()  
  
 delete(gls.m, GetGoid())  
}

这样一个极简的Goroutine局部存储gls对象就完成了。

下面是使用局部存储简单的例子：

package main  
  
import (  
 "fmt"  
 gls "path/to/gls"  
 "sync"  
)  
  
func main() {  
 var wg sync.WaitGroup  
 for i := 0; i < 5; i++ {  
 wg.Add(1)  
 go func(idx int) {  
 defer wg.Done()  
 defer gls.Clean()  
  
 defer func() {  
 fmt.Printf("%d: number = %d\n", idx, gls.Get("number"))  
 }()  
 gls.Put("number", idx+100)  
 }(i)  
 }  
 wg.Wait()  
}

通过Goroutine局部存储，不同层次函数之间可以共享存储资源。同时为了避免资源泄漏，需要在Goroutine的根函数中，通过defer语句调用gls.Clean()函数释放资源。

# 9. Delve调试器

目前Go语言支持GDB、LLDB和Delve几种调试器。其中GDB是最早支持的调试工具，LLDB是macOS系统推荐的标准调试工具。但是GDB和LLDB对Go语言的专有特性都缺乏很大支持，而只有Delve是专门为Go语言设计开发的调试工具。而且Delve本身也是采用Go语言开发，对Windows平台也提供了一样的支持。本节我们基于Delve简单解释如何调试Go汇编程序。

## 9.1 Delve入门

首先根据官方的文档正确安装Delve调试器。我们会先构造一个简单的Go语言代码，用于熟悉下Delve的简单用法。

创建main.go文件，main函数先通过循初始化一个切片，然后输出切片的内容：

package main  
  
import (  
 "fmt"  
)  
  
func main() {  
 nums := make([]int, 5)  
 for i := 0; i < len(nums); i++ {  
 nums[i] = i \* i  
 }  
 fmt.Println(nums)  
}

命令行进入包所在目录，然后输入dlv debug命令进入调试：

$ dlv debug

Type 'help' for list of commands.

(dlv)

输入help命令可以查看到Delve提供的调试命令列表：

(dlv) help

The following commands are available:

args ------------------------ Print function arguments.

break (alias: b) ------------ Sets a breakpoint.

breakpoints (alias: bp) ----- Print out info for active breakpoints.

clear ----------------------- Deletes breakpoint.

clearall -------------------- Deletes multiple breakpoints.

condition (alias: cond) ----- Set breakpoint condition.

config ---------------------- Changes configuration parameters.

continue (alias: c) --------- Run until breakpoint or program termination.

disassemble (alias: disass) - Disassembler.

down ------------------------ Move the current frame down.

exit (alias: quit | q) ------ Exit the debugger.

frame ----------------------- Set the current frame, or execute command...

funcs ----------------------- Print list of functions.

goroutine ------------------- Shows or changes current goroutine

goroutines ------------------ List program goroutines.

help (alias: h) ------------- Prints the help message.

list (alias: ls | l) -------- Show source code.

locals ---------------------- Print local variables.

next (alias: n) ------------- Step over to next source line.

on -------------------------- Executes a command when a breakpoint is hit.

print (alias: p) ------------ Evaluate an expression.

regs ------------------------ Print contents of CPU registers.

restart (alias: r) ---------- Restart process.

set ------------------------- Changes the value of a variable.

source ---------------------- Executes a file containing a list of delve...

sources --------------------- Print list of source files.

stack (alias: bt) ----------- Print stack trace.

step (alias: s) ------------- Single step through program.

step-instruction (alias: si) Single step a single cpu instruction.

stepout --------------------- Step out of the current function.

thread (alias: tr) ---------- Switch to the specified thread.

threads --------------------- Print out info for every traced thread.

trace (alias: t) ------------ Set tracepoint.

types ----------------------- Print list of types

up -------------------------- Move the current frame up.

vars ------------------------ Print package variables.

whatis ---------------------- Prints type of an expression.

Type help followed by a command for full documentation.

(dlv)

每个Go程序的入口是main.main函数，我们可以用break在此设置一个断点：

(dlv) break main.main

Breakpoint 1 set at 0x10ae9b8 for main.main() ./main.go:7

然后通过breakpoints查看已经设置的所有断点：

(dlv) breakpoints

Breakpoint unrecovered-panic at 0x102a380 for runtime.startpanic()

/usr/local/go/src/runtime/panic.go:588 (0)

print runtime.curg.\_panic.arg

Breakpoint 1 at 0x10ae9b8 for main.main() ./main.go:7 (0)

我们发现除了我们自己设置的main.main函数断点外，Delve内部已经为panic异常函数设置了一个断点。

通过vars命令可以查看全部包级的变量。因为最终的目标程序可能含有大量的全局变量，我们可以通过一个正则参数选择想查看的全局变量：

(dlv) vars main

main.initdone· = 2

runtime.main\_init\_done = chan bool 0/0

runtime.mainStarted = true

(dlv)

然后就可以通过continue命令让程序运行到下一个断点处：

(dlv) continue

> main.main() ./main.go:7 (hits goroutine(1):1 total:1) (PC: 0x10ae9b8)

2:

3: import (

4: "fmt"

5: )

6:

=> 7: func main() {

8: nums := make([]int, 5)

9: for i := 0; i < len(nums); i++ {

10: nums[i] = i \* i

11: }

12: fmt.Println(nums)

(dlv)

输入next命令单步执行进入main函数内部：

(dlv) next

> main.main() ./main.go:8 (PC: 0x10ae9cf)

3: import (

4: "fmt"

5: )

6:

7: func main() {

=> 8: nums := make([]int, 5)

9: for i := 0; i < len(nums); i++ {

10: nums[i] = i \* i

11: }

12: fmt.Println(nums)

13: }

(dlv)

进入函数之后可以通过args和locals命令查看函数的参数和局部变量：

(dlv) args

(no args)

(dlv) locals

nums = []int len: 842350763880, cap: 17491881, nil

因为main函数没有参数，因此args命令没有任何输出。而locals命令则输出了局部变量nums切片的值：此时切片还未完成初始化，切片的底层指针为nil，长度和容量都是一个随机数值。

再次输入next命令单步执行后就可以查看到nums切片初始化之后的结果了：

(dlv) next

> main.main() ./main.go:9 (PC: 0x10aea12)

4: "fmt"

5: )

6:

7: func main() {

8: nums := make([]int, 5)

=> 9: for i := 0; i < len(nums); i++ {

10: nums[i] = i \* i

11: }

12: fmt.Println(nums)

13: }

(dlv) locals

nums = []int len: 5, cap: 5, [...]

i = 17601536

(dlv)

此时因为调试器已经到了for语句行，因此局部变量出现了还未初始化的循环迭代变量i。

下面我们通过组合使用break和condition命令，在循环内部设置一个条件断点，当循环变量i等于3时断点生效：

(dlv) break main.go:10

Breakpoint 2 set at 0x10aea33 for main.main() ./main.go:10

(dlv) condition 2 i==3

(dlv)

然后通过continue执行到刚设置的条件断点，并且输出局部变量：

(dlv) continue

> main.main() ./main.go:10 (hits goroutine(1):1 total:1) (PC: 0x10aea33)

5: )

6:

7: func main() {

8: nums := make([]int, 5)

9: for i := 0; i < len(nums); i++ {

=> 10: nums[i] = i \* i

11: }

12: fmt.Println(nums)

13: }

(dlv) locals

nums = []int len: 5, cap: 5, [...]

i = 3

(dlv) print nums

[]int len: 5, cap: 5, [0,1,4,0,0]

(dlv)

我们发现当循环变量i等于3时，nums切片的前3个元素已经正确初始化。

我们还可以通过stack查看当前执行函数的栈帧信息：

(dlv) stack

0 0x00000000010aea33 in main.main

at ./main.go:10

1 0x000000000102bd60 in runtime.main

at /usr/local/go/src/runtime/proc.go:198

2 0x0000000001053bd1 in runtime.goexit

at /usr/local/go/src/runtime/asm\_amd64.s:2361

(dlv)

或者通过goroutine和goroutines命令查看当前Goroutine相关的信息：

(dlv) goroutine

Thread 101686 at ./main.go:10

Goroutine 1:

Runtime: ./main.go:10 main.main (0x10aea33)

User: ./main.go:10 main.main (0x10aea33)

Go: /usr/local/go/src/runtime/asm\_amd64.s:258 runtime.rt0\_go (0x1051643)

Start: /usr/local/go/src/runtime/proc.go:109 runtime.main (0x102bb90)

(dlv) goroutines

[4 goroutines]

\* Goroutine 1 - User: ./main.go:10 main.main (0x10aea33) (thread 101686)

Goroutine 2 - User: /usr/local/go/src/runtime/proc.go:292 \

runtime.gopark (0x102c189)

Goroutine 3 - User: /usr/local/go/src/runtime/proc.go:292 \

runtime.gopark (0x102c189)

Goroutine 4 - User: /usr/local/go/src/runtime/proc.go:292 \

runtime.gopark (0x102c189)

(dlv)

最后完成调试工作后输入quit命令退出调试器。至此我们已经掌握了Delve调试器器的简单用法。

## 9.2. 调试汇编程序

用Delve调试Go汇编程序的过程比调试Go语言程序更加简单。调试汇编程序时，我们需要时刻关注寄存器的状态，如果涉及函数调用或局部变量或参数还需要重点关注栈寄存器SP的状态。

为了编译演示，我们重新实现一个更简单的main函数：

package main  
  
func main() { asmSayHello() }  
  
func asmSayHello()

在main函数中调用汇编语言实现的asmSayHello函数输出一个字符串。

asmSayHello函数在main\_amd64.s文件中实现：

#include "textflag.h"

#include "funcdata.h"

// "Hello World!\n"

DATA text<>+0(SB)/8,$"Hello Wo"

DATA text<>+8(SB)/8,$"rld!\n"

GLOBL text<>(SB),NOPTR,$16

// func asmSayHello()

TEXT ·asmSayHello(SB), $16-0

NO\_LOCAL\_POINTERS

MOVQ $text<>+0(SB), AX

MOVQ AX, (SP)

MOVQ $16, 8(SP)

CALL runtime·printstring(SB)

RET

参考前面的调试流程，在执行到main函数断点时，可以disassemble反汇编命令查看main函数对应的汇编代码：

(dlv) break main.main

Breakpoint 1 set at 0x105011f for main.main() ./main.go:3

(dlv) continue

> main.main() ./main.go:3 (hits goroutine(1):1 total:1) (PC: 0x105011f)

1: package main

2:

=>3: func main() { asmSayHello() }

4:

5: func asmSayHello()

(dlv) disassemble

TEXT main.main(SB) /path/to/pkg/main.go

main.go:3 0x1050110 65488b0c25a0080000 mov rcx, qword ptr g [0x8a0]

main.go:3 0x1050119 483b6110 cmp rsp, qword ptr [r +0x10]

main.go:3 0x105011d 761a jbe 0x1050139

=>main.go:3 0x105011f\* 4883ec08 sub rsp, 0x8

main.go:3 0x1050123 48892c24 mov qword ptr [rsp], rbp

main.go:3 0x1050127 488d2c24 lea rbp, ptr [rsp]

main.go:3 0x105012b e880000000 call $main.asmSayHello

main.go:3 0x1050130 488b2c24 mov rbp, qword ptr [rsp]

main.go:3 0x1050134 4883c408 add rsp, 0x8

main.go:3 0x1050138 c3 ret

main.go:3 0x1050139 e87288ffff call $runtime.morestack\_noctxt

main.go:3 0x105013e ebd0 jmp $main.main

(dlv)

虽然main函数内部只有一行函数调用语句，但是却生成了很多汇编指令。在函数的开头通过比较rsp寄存器判断栈空间是否不足，如果不足则跳转到0x1050139地址调用runtime.morestack函数进行栈扩容，然后跳回到main函数开始位置重新进行栈空间测试。而在asmSayHello函数调用之前，先扩展rsp空间用于临时存储rbp寄存器的状态，在函数返回后通过栈恢复rbp的值并回收临时栈空间。通过对比Go语言代码和对应的汇编代码，我们可以加深对Go汇编语言的理解。

从汇编语言角度深刻Go语言各种特性的工作机制对调试工作也是一个很大的帮助。如果希望在汇编指令层面调试Go代码，Delve还提供了一个step-instruction单步执行汇编指令的命令。

现在我们依然用break命令在asmSayHello函数设置断点，并且输入continue命令让调试器执行到断点位置停下：

(dlv) break main.asmSayHello

Breakpoint 2 set at 0x10501bf for main.asmSayHello() ./main\_amd64.s:10

(dlv) continue

> main.asmSayHello() ./main\_amd64.s:10 (hits goroutine(1):1 total:1) (PC: 0x10501bf)

5: DATA text<>+0(SB)/8,$"Hello Wo"

6: DATA text<>+8(SB)/8,$"rld!\n"

7: GLOBL text<>(SB),NOPTR,$16

8:

9: // func asmSayHello()

=> 10: TEXT ·asmSayHello(SB), $16-0

11: NO\_LOCAL\_POINTERS

12: MOVQ $text<>+0(SB), AX

13: MOVQ AX, (SP)

14: MOVQ $16, 8(SP)

15: CALL runtime·printstring(SB)

(dlv)

此时我们可以通过regs查看全部的寄存器状态：

(dlv) regs

rax = 0x0000000001050110

rbx = 0x0000000000000000

rcx = 0x000000c420000300

rdx = 0x0000000001070be0

rdi = 0x000000c42007c020

rsi = 0x0000000000000001

rbp = 0x000000c420049f78

rsp = 0x000000c420049f70

r8 = 0x7fffffffffffffff

r9 = 0xffffffffffffffff

r10 = 0x0000000000000100

r11 = 0x0000000000000286

r12 = 0x000000c41fffff7c

r13 = 0x0000000000000000

r14 = 0x0000000000000178

r15 = 0x0000000000000004

rip = 0x00000000010501bf

rflags = 0x0000000000000206

...

(dlv)

因为AMD64的各种寄存器非常多，项目的信息中刻意省略了非通用的寄存器。如果再单步执行到13行时，可以发现AX寄存器值的变化。

(dlv) regs

rax = 0x00000000010a4060

rbx = 0x0000000000000000

rcx = 0x000000c420000300

...

(dlv)

因此我们可以推断汇编程序内部定义的text<>数据的地址为0x00000000010a4060。我们可以用过print命令来查看该内存内的数据：

(dlv) print \*(\*[5]byte)(uintptr(0x00000000010a4060))

[5]uint8 [72,101,108,108,111]

(dlv)

我们可以发现输出的[5]uint8 [72,101,108,108,111]刚好是对应“Hello”字符串。通过类似的方法，我们可以通过查看SP对应的栈指针位置，然后查看栈中局部变量的值。

至此我们就掌握了Go汇编程序的简单调试技术。

# 10. 补充说明

如果是纯粹学习汇编语言，则可以从《深入理解程序设计：使用Linux汇编语言》开始，该书讲述了如何以C语言的思维变现汇编程序。如果是学习X86汇编，则可以从《汇编语言：基于x86处理器》一开始，然后再结合《现代x86汇编语言程序设计》学习AVX等高级汇编指令的使用。

Go汇编语言的官方文档非常匮乏。其中“A Quick Guide to Go's Assembler”是唯一的一篇系统讲述Go汇编语言的官方文章，该文章中又引入了另外两篇Plan9的文档：A Manual for the Plan 9 assembler 和 Plan 9 C Compilers。Plan9的两篇文档分别讲述了汇编语言以及和汇编有关联的C语言编译器的细节。看过这几篇文档之后会对Go汇编语言有了一些模糊的概念，剩下的就是在实战中通过代码学习了。

Go语言的编译器和汇编器都带了一个-S参数，可以查看生成的最终目标代码。通过对比目标代码和原始的Go语言或Go汇编语言代码的差异可以加深对底层实现的理解。同时Go语言连接器的实现代码也包含了很多相关的信息。Go汇编语言是依托Go语言的语言，因此理解Go语言的工作原理是也是必要的。比较重要的部分是Go语言runtime和reflect包的实现原理。如果读者了解CGO技术，那么对Go汇编语言的学习也是一个巨大的帮助。最后是要了解syscall包是如何实现系统调用的。

得益于Go语言的设计，Go汇编语言的优势也非常明显：跨操作系统、不同CPU之间的用法也非常相似、支持C语言预处理器、支持模块。同时Go汇编语言也存在很多不足：它不是一个独立的语言，底层需要依赖Go语言甚至操作系统；很多高级特性很难通过手工汇编完成。虽然Go语言官方尽量保持Go汇编语言简单，但是汇编语言是一个比较大的话题，大到足以写一本Go汇编语言的教程。本章的目的是让大家对Go汇编语言简单入门，在看到底层汇编代码的时候不会一头雾水，在某些遇到性能受限制的场合能够通过Go汇编突破限制。