# 工具

## 工具集

### go build

-gcflags: 传递给编译器的参数

-ldflags: 传递给链接器的参数

-work: 查看编译临时目录

-race: 允许数据竞争检测(仅支持amd64)

-n: 查看但不执行编译指令

-x: 查看并执行编译命令

-a: 强制重新编译所有依赖包

-v: 查看被编译的包名，包括依赖包

-p n:并行编译所使用的CPU数，默认为全部

-o:输出文件名

gcflags:

-B 禁用边界检查

-N 禁用优化

-l 禁用函数内联

-u 禁用unsafe代码

-m 输出优化信息

-S 输出汇编代码

ldflags:

-w 禁用DRAWF调试信息，但不包括符号表

-s 禁用符号表

-X 修改字符串符号值 -X main.VER ‘0.99’ -X main.S ‘abc’

-H 链接文件类型，其中包括windowsgui. cmd/ld/doc.go

更多参数:

go tool compile -h

go tool link -h

### go install

和go build参数相同，将生成文件拷贝到bin,pkg目录，优先使用GOBIN环境变量指定目录

go clean

-n 查看但不执行清理命令

-x 查看并执行清理命令

-i 删除bin,pkg目录下的二进制文件

-r 清理所有依赖包临时目录

### go get

-d 仅下载，不执行安装命令

-t 下载测试所需依赖包

-u 更新包包括其依赖

-v 查看并执行命令

go tool objdump

反汇编可执行文件

go tool objdump -s “main\.\w+” test

## 条件编译

1. GOOS GOARCH

通过runtime.GOOS/GOARCH判断，或使用编译约束标记

// +build darwin linux

🡨-----必须有空行,以区别包文档。

package main

在源文件(.go, .h, .c, .s等)头部添加”+build”注释，指示编译器检查相关环境变量。

多个约束标记会合并处理。其中空格表示OR,逗号AND，感叹号NOT。

// +build darwin linux -------🡪合并结果(darwin OR linux) AND (amd64 AND (NOT cgo))

// +build amd64,!cgo

如果GOOS,GOARCH不符号，编译器会忽略该文件

1. 文件名

还可使用文件名来表示编译约束，比如test\_darwin\_amd64.go.使用文件名拆分多个不同平台源文件，更利于维护。

-rw-r--r--@ 1 zhangxiaoan admin 11545 11 29 05:38 os\_darwin.c

-rw-r--r--@ 1 zhangxiaoan admin 1382 11 29 05:38 os\_darwin.h

-rw-r--r--@ 1 zhangxiaoan admin 6990 11 29 05:38 os\_freebsd.c

-rw-r--r--@ 1 zhangxiaoan admin 791 11 29 05:38 os\_freebsd.h

-rw-r--r--@ 1 zhangxiaoan admin 644 11 29 05:38 os\_freebsd\_arm.c

-rw-r--r--@ 1 zhangxiaoan admin 8624 11 29 05:38 os\_linux.c

-rw-r--r--@ 1 zhangxiaoan admin 1067 11 29 05:38 os\_linux.h

-rw-r--r--@ 1 zhangxiaoan admin 861 11 29 05:38 os\_linux\_386.c

-rw-r--r--@ 1 zhangxiaoan admin 2418 11 29 05:38 os\_linux\_arm.c

⽀支持：\*\_GOOS、\*\_GOARCH、\*\_GOOS\_GOARCH、\*\_GOARCH\_GOOS 格式。

可忽略某个文件，或指定编译器版本号。更多信息参考标准库go/build文档。

// +build ignore

// +build go1.2 🡨------最低需要go1.2编译

1. 自定义约束条件

需使用”go build -tags”参数。

## 跨平台编译

首先得生成与平台相关的工具和标准库

$ cd /usr/local/go/src

$ GOOS=linux GOARCH=amd64 ./make.bash --no-clean

# Building C bootstrap tool.

cmd/dist

# Building compilers and Go bootstrap tool for host, darwin/amd64.

cmd/6l

cmd/6a

cmd/6c

cmd/6g

...

---

Installed Go for linux/amd64 in /usr/local/go

Installed commands in /usr/local/go/bin

--no-clean参数避免清除其他平台文件

然后回到项目所在目录，设定GOOS,GOARCH环境变量即可编译目标平台文件

$ GOOS=linux GOARCH=amd64 go build -o test

$ file test

learn: ELF 64-bit LSB executable, x86-64, version 1 (SYSV)

$ uname -a

Darwin Kernel Version 12.5.0: RELEASE\_X86\_64 x86\_64

## GO GDB语言调试

gdb不能很好的理解GO程序.即使使用gccgo,栈管理，线程和运行时与GDB期望的执行模式也不完全一样。因此GDB对GO程序并不完全可靠，尤其是在调试并发程序时。

GO程序包含DWARF3调试信息，GDB７.１以上版本可以进行调试。.

传递”-w”选项给链接器会去除调试信息。(比如:go build -ldflags “-w” prog.go)

gc编译器生成的代码包含内联函数和寄存器变量优化，可以通过-gcflags “-N -l”去掉优化

### 常用操作

* 显示代码文件和行号信息，设置断点和查看反汇编:
* (gdb) **list**
* (gdb) **list *line***
* (gdb) **list *file.go*:*line***
* (gdb) **break *line***
* (gdb) **break *file.go*:*line***

(gdb) **disas**

* 显示backtraces和stack frames:
* (gdb) **bt**

(gdb) **frame *n***

* 显示名称，类型和栈上的本地变量，参数和返回值:
* (gdb) **info locals**
* (gdb) **info args**
* (gdb) **p variable**

(gdb) **whatis variable**

* 显示名称，类型和全局变量的位置:

(gdb) **info variables *regexp***

### go扩展

GDB最新的扩展机制允许为二进制程序加载扩展脚本。工具链使用这个机制为GDB扩展一些命令来监控运行代码的内部（如goroutines）和内置map,slice,和通道的清晰打印.

如果GDB不能找到扩展脚本，可以手工执行:

(gdb) source /usr/lib/golang/src/runtime/runtime-gdb.py

* 打印string, slice, map, channel or interface:

(gdb) **p *var***

* strings, slices和maps的$len()和$cap() 函数:

(gdb) **p $len(*var*)**

* 转换接口为它们的动态类型:
* (gdb) **p $dtype(*var*)**

(gdb) **iface *var***

* 查看协程:
* (gdb) **info goroutines**
* (gdb) **goroutine *n* *cmd***

(gdb) **help goroutine**

举例:

(gdb) **goroutine 12 bt**

### 已知问题

字符串的清晰打印只对原始字符串类型有效，从其派生的类型无效

运行库的C代码部分没有类型信息

GDB不理解GO的命名方式， 包中的类型的函数形式为pkg.(\*MyType).Meth

所有的全局变量集中到包”main”

### 教程

通过regexp包的单元测试程序来演示GDB的使用。

构建程序:cd $GOROOT/src/regexp; go test -c

生成可执行程序regexp.test

$ **gdb regexp.test**

GNU gdb (GDB) 7.2-gg8

Copyright (C) 2010 Free Software Foundation, Inc.

License GPLv 3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>

Type "show copying" and "show warranty" for licensing/warranty details.

This GDB was configured as "x86\_64-linux".

Reading symbols from /home/user/go/src/regexp/regexp.test...

done.

Loading Go Runtime support.

(gdb)

 "Loading Go Runtime support" 意味着GDB加载了扩展 $GOROOT/src/runtime/runtime-gdb.py.

如果没有自动加载，可以手工加载:

(gdb) source /usr/lib/go/src/runtime/runtime-gdb.py

Loading Go Runtime support.

#### 查看代码

 使用 "l" 或 "list"命令查看源代码.

(gdb) **l**

通过函数名显示源代码的特定部分(必须包含其包名).

(gdb) **l main.main**

显示特定文件和行号

(gdb) **l regexp.go:1**

(gdb) *# Hit enter to repeat last command. Here, this lists next 10 lines.*

#### 关于命名

变量和函数名必须携带其所属的包名。regexp包中的Compile函数被GDB当作’regexp.Compile’

方法必须包含其接收着类型。比如，\*Regexp类型的String方法必须写在’regexp.(\*Regexp).String’

引用其它变量的类型在调试信息中被加上了一个数字后缀，被闭包引用的变量将会显示为一个添加&前缀的指针

#### 设置断点

为TestFind函数设置断点:

(gdb) **b 'regexp.TestFind'**

Breakpoint 1 at 0x424908: file /home/user/go/src/regexp/find\_test.go, line 148.

运行程序:

(gdb) **run**

Starting program: /home/user/go/src/regexp/regexp.test

Breakpoint 1, regexp.TestFind (t=0xf8404a89c0) at /home/user/go/src/regexp/find\_test.go:148

148 func TestFind(t \*testing.T) {

执行在断点处暂停。查看哪个协程正在运行:

(gdb) **info goroutines**

1 waiting runtime.gosched

\* 13 running runtime.goexit

标记着 \* 的为当前正在运行的协程

**查看栈.**

在程序暂停的地方查看栈回溯:

(gdb) **bt** *# backtrace*

#0 regexp.TestFind (t=0xf8404a89c0) at /home/user/go/src/regexp/find\_test.go:148

#1 0x000000000042f60b in testing.tRunner (t=0xf8404a89c0, test=0x573720) at /home/user/go/src/testing/testing.go:156

#2 0x000000000040df64 in runtime.initdone () at /home/user/go/src/runtime/proc.c:242

#3 0x000000f8404a89c0 in ?? ()

#4 0x0000000000573720 in ?? ()

#5 0x0000000000000000 in ?? ()

另外一个协程１，阻塞在通道接收:

(gdb) **goroutine 1 bt**

#0 0x000000000040facb in runtime.gosched () at /home/user/go/src/runtime/proc.c:873

#1 0x00000000004031c9 in runtime.chanrecv (c=void, ep=void, selected=void, received=void)

at /home/user/go/src/runtime/chan.c:342

#2 0x0000000000403299 in runtime.chanrecv1 (t=void, c=void) at/home/user/go/src/runtime/chan.c:423

#3 0x000000000043075b in testing.RunTests (matchString={void (struct string, struct string, bool \*, error \*)}

0x7ffff7f9ef60, tests= []testing.InternalTest = {...}) at /home/user/go/src/testing/testing.go:201

#4 0x00000000004302b1 in testing.Main (matchString={void (struct string, struct string, bool \*, error \*)}

0x7ffff7f9ef80, tests= []testing.InternalTest = {...}, benchmarks= []testing.InternalBenchmark = {...})

at /home/user/go/src/testing/testing.go:168

#5 0x0000000000400dc1 in main.main () at /home/user/go/src/regexp/\_testmain.go:98

#6 0x00000000004022e7 in runtime.mainstart () at /home/user/go/src/runtime/amd64/asm.s:78

#7 0x000000000040ea6f in runtime.initdone () at /home/user/go/src/runtime/proc.c:243

#8 0x0000000000000000 in ?? ()

栈桢显示我们当前在执行 regexp.TestFind 函数.

(gdb) **info frame**

Stack level 0, frame at 0x7ffff7f9ff88:

rip = 0x425530 in regexp.TestFind (/home/user/go/src/regexp/find\_test.go:148);

saved rip 0x430233

called by frame at 0x7ffff7f9ffa8

source language minimal.

Arglist at 0x7ffff7f9ff78, args: t=0xf840688b60

Locals at 0x7ffff7f9ff78, Previous frame's sp is 0x7ffff7f9ff88

Saved registers:

rip at 0x7ffff7f9ff80

命令info locals显示函数的所有本地变量， 但是这个命令有一定风险，因为它会尝试打印未初始化的变量。可能造成gdb打印很大的数组.

查看函数参数:

(gdb) **info args**

t = 0xf840688b60

打印参数时显示的时一个指向 Regexp值的指针.

注意GDB错误地将\*放在了类型名的右边.

(gdb) **p re**

(gdb) p t

$1 = (struct testing.T \*) 0xf840688b60

(gdb) p t

$1 = (struct testing.T \*) 0xf840688b60

(gdb) p \*t

$2 = {errors = "", failed = false, ch = 0xf8406f5690}

(gdb) p \*t->ch

$3 = struct hchan<\*testing.T>

struct hchan<\*testing.T>是通道运行时的内部表示. 它当前为空, 否则gdb会清晰地打印其内容.

继续执行:

(gdb) **n** *# execute next line*

149 for \_, test := range findTests {

(gdb) *# enter is repeat*

150 re := MustCompile(test.pat)

(gdb) **p test.pat**

$4 = ""

(gdb) **p re**

$5 = (struct regexp.Regexp \*) 0xf84068d070

(gdb) **p \*re**

$6 = {expr = "", prog = 0xf840688b80, prefix = "", prefixBytes = []uint8, prefixComplete = true,

prefixRune = 0, cond = 0 '\000', numSubexp = 0, longest = false, mu = {state = 0, sema = 0},

machine = []\*regexp.machine}

(gdb) **p \*re->prog**

$7 = {Inst = []regexp/syntax.Inst = {{Op = 5 '\005', Out = 0, Arg = 0, Rune = []int}, {Op =

6 '\006', Out = 2, Arg = 0, Rune = []int}, {Op = 4 '\004', Out = 0, Arg = 0, Rune = []int}},

Start = 1, NumCap = 2}

我们能够跟进String函数通过调用”s”:

(gdb) **s**

regexp.(\*Regexp).String (re=0xf84068d070, noname=void) at /home/user/go/src/regexp/regexp.go:97

97 func (re \*Regexp) String() string {

通过bt查看我们执行到哪里:

(gdb) **bt**

#0 regexp.(\*Regexp).String (re=0xf84068d070, noname=void)

at /home/user/go/src/regexp/regexp.go:97

#1 0x0000000000425615 in regexp.TestFind (t=0xf840688b60)

at /home/user/go/src/regexp/find\_test.go:151

#2 0x0000000000430233 in testing.tRunner (t=0xf840688b60, test=0x5747b8)

at /home/user/go/src/testing/testing.go:156

#3 0x000000000040ea6f in runtime.initdone () at /home/user/go/src/runtime/proc.c:243

....

查看源代码:

(gdb) **l**

92 mu sync.Mutex

93 machine []\*machine

94 }

95

96 // String returns the source text used to compile the regular expression.

97 func (re \*Regexp) String() string {

98 return re.expr

99 }

100

101 // Compile parses a regular expression and returns, if successful,

#### 清晰打印

GDB's pretty printing mechanism is triggered by regexp matches on type names. An example for slices:

(gdb) **p utf**

$22 = []uint8 = {0 '\000', 0 '\000', 0 '\000', 0 '\000'}

因为slices, arrays 和 strings 不是C指针,GDB不能解释带下标的操作，但是你可以通过下面的方式查看其内容 (使用tab帮助完成):

(gdb) **p slc**

$11 = []int = {0, 0}

(gdb) **p slc->***<TAB>*

array slc len

(gdb) **p slc->array**

$12 = (int \*) 0xf84057af00

(gdb) **p slc->array[1]**

$13 = 0

扩展函数$len 和 $cap支持strings, arrays 和 slices:

(gdb) **p $len(utf)**

$23 = 4

(gdb) **p $cap(utf)**

$24 = 4

通道和maps是引用类型，gdb显示为类似C++的指针类型 hash<int,string>\*.解引用操作将会触发清晰打印

接口在运行时表示为一个指向类型描述的指针和一个指向值的指针.

Go GDB运行扩展对运行时类型解码并自动触发清晰打印.

扩展函数$dtype 为你解码运行时类型 (例子在 regexp.go 的293行.)

(gdb) **p i**

$4 = {str = "cbb"}

(gdb) **whatis i**

type = regexp.input

(gdb) **p $dtype(i)**

$26 = (struct regexp.inputBytes \*) 0xf8400b4930

(gdb) **iface i**

regexp.input: struct regexp.inputBytes \*