**GDB调试Go 程序**

说明：作为一门静态语言，似乎支持调试是必须的，而且，Go初学者喜欢问的问题也是：大家都用什么IDE？怎么调试？

其实，Go是为多核和并发而生，真正的项目，你用单步调试，原本没问题的，可能会调出有问题。更好的调试方式是跟PHP这种语言一样，用打印的方式（日志或print）。

当然，简单的小程序，如果单步调试，可以看到一些内部的运行机理，对于学习还是挺有好处的。下面介绍一下用GDB调试Go程序：（目前IDE支持调试Go程序，用的也是GDB。要求GDB 7.1以上）

以下内容来自雨痕的《Go语言学习笔记》（[下载Go资源](http://bbs.studygolang.com/forum.php?mod=viewthread&tid=10&extra=page%3D1)）：

默认情况下，编译过的二进制文件已经包含了 DWARFv3 调试信息，只要 GDB7.1 以上版本都可以进行调试。 在OSX下，如无法执行调试指令，可尝试用sudo方式执行gdb。

删除调试符号：go build -ldflags “-s -w”

* -s: 去掉符号信息。
* -w: 去掉DWARF调试信息。

关闭内联优化：go build -gcflags “-N -l”

调试相关函数：

* runtime.Breakpoint()：触发调试器断点。
* runtime/debug.PrintStack()：显示调试堆栈。
* log：适合替代 print显示调试信息。

GDB 调试支持：

* 参数载入：gdb -d $GCROOT 。
* 手工载入：source pkg/runtime/runtime-gdb.py。

更多细节，请参考: http://golang.org/doc/gdb

调试演示：(OSX 10.8.2, Go1.0.3, GDB7.5.1)

|  |  |  |
| --- | --- | --- |
| 1 | package main | |
| 2 |  |

|  |  |
| --- | --- |
| 3 | import ( |
| 4 | "fmt" | |

|  |  |  |
| --- | --- | --- |
| 5 | "runtime" | |
| 6 | ) |

|  |  |
| --- | --- |
| 7 |  |
| 8 | | | func test(s string, x int) (r string) { |

|  |  |  |  |
| --- | --- | --- | --- |
| 9 | | r = fmt.Sprintf("test: %s %d", s, x) | |
| 10 | runtime.Breakpoint() | |

|  |  |  |
| --- | --- | --- |
| 11 | return r | |
| 12 | } |

|  |  |
| --- | --- |
| 13 | func main() { |
| 14 | s := "haha" | |

|  |  |
| --- | --- |
| 15 | i := 1234 |
| 16 | println(test(s, i)) | |

|  |  |
| --- | --- |
| 17 | } |

$ **go build -gcflags “-N -l”** // 编译，关闭内联优化。

$ s**udo gdb demo** // 启动 gdb 调试器，手工载入 Go Runtime 。  
GNU gdb (GDB) 7.5.1  
Reading symbols from demo…done.  
(gdb) source /usr/local/go/src/pkg/runtime/runtime-gdb.py  
Loading Go Runtime support.

(gdb)**l main.main** // 以 .方式查看源码。  
9 r = fmt.Sprintf(“test: %s %d”, s, x)  
10 runtime.Breakpoint()  
11 return r  
12 }  
13  
14 func main() {  
15 s := “haha”  
16 i := 1234  
17 println(test(s, i))  
18 }

(gdb)**l main.go:8** // 以 :方式查看源码。  
3 import (  
4 “fmt”  
5 “runtime”  
6 )  
7  
8 func test(s string, x int) (r string) {  
9 r = fmt.Sprintf(“test: %s %d”, s, x)  
10 runtime.Breakpoint()  
11 return r  
12 }

(gdb) **b main.main** // 以 .方式设置断点。  
Breakpoint 1 at 0×2131: file main.go, line 14.

(gdb) **b main.go:17** // 以 :方式设置断点。  
Breakpoint 2 at 0×2167: file main.go, line 17.

(gdb) **info breakpoints** // 查看所有断点。  
Num Type Disp Enb Address What  
1 breakpoint keep y 0×0000000000002131 in main.main at main.go:14  
2 breakpoint keep y 0×0000000000002167 in main.main at main.go:17

(gdb)**r** // 启动进程，触发第一个断点。  
Starting program: demo  
[New Thread 0x1c03 of process 4088]  
[Switching to Thread 0x1c03 of process 4088]  
Breakpoint 1, main.main () at main.go:14  
14 func main() {

(gdb) **info goroutines** // 查看 goroutines 信息。  
\* 1 running runtime.gosched  
\* 2 syscall runtime.entersyscall

(gdb) **goroutine 1 bt** // 查看指定序号的 goroutine 调用堆栈。  
#0 0x000000000000f6c0 in runtime.gosched () at pkg/runtime/proc.c:927  
#1 0x000000000000e44c in runtime.main () at pkg/runtime/proc.c:244  
#2 0x000000000000e4ef in schedunlock () at pkg/runtime/proc.c:267  
#3 0×0000000000000000 in ?? ()

(gdb) **goroutine 2 bt** // 这个 goroutine 貌似跟 GC 有关。  
#0 runtime.entersyscall () at pkg/runtime/proc.c:989  
#1 0x000000000000d01d in runtime.MHeap\_Scavenger () at pkg/runtime/mheap.c:363  
#2 0x000000000000e4ef in schedunlock () at pkg/runtime/proc.c:267  
#3 0×0000000000000000 in ?? ()

(gdb) **c** / / 继续执行，触发下一个断点。  
Continuing.  
Breakpoint 2, main.main () at main.go:17  
17! ! println(test(s, i))

(gdb) **info goroutines** // 当前 goroutine 序号为 1。  
\* 1 running runtime.gosched  
2 runnable runtime.gosched

(gdb) **goroutine 1 bt** // 当前 goroutine 调用堆栈。  
#0 0x000000000000f6c0 in runtime.gosched () at pkg/runtime/proc.c:927  
#1 0x000000000000e44c in runtime.main () at pkg/runtime/proc.c:244  
#2 0x000000000000e4ef in schedunlock () at pkg/runtime/proc.c:267  
#3 0×0000000000000000 in ?? ()

(gdb) **bt** // 查看当前调⽤堆栈，可以与当前 goroutine 调用堆栈对比。  
#0 main.main () at main.go:17  
#1 0x000000000000e44c in runtime.main () at pkg/runtime/proc.c:244  
#2 0x000000000000e4ef in schedunlock () at pkg/runtime/proc.c:267  
#3 0×0000000000000000 in ?? ()

(gdb) **info frame** // 堆栈帧信息。  
Stack level 0, frame at 0x442139f88:  
rip = 0×2167 in main.main (main.go:17); saved rip 0xe44c  
called by frame at 0x442139fb8  
source language go.  
Arglist at 0x442139f28, args:  
Locals at 0x442139f28, Previous frame’s sp is 0x442139f88  
Saved registers:  
rip at 0x442139f80

(gdb) **info locals** // 查看局部变量。  
i = 1234  
s = “haha”

(gdb) p s // 以 Pretty-Print 方式查看变量。  
$1 = “haha”

(gdb) **p $len(s)** // 获取对象长度($cap)  
$2 = 4

(gdb) **whatis i** // 查看对象类型。  
type = int

(gdb) **c** // 继续执行，触发 breakpoint() 断点。  
Continuing.  
Program received signal SIGTRAP, Trace/breakpoint trap.  
runtime.breakpoint () at pkg/runtime/asm\_amd64.s:81  
81 RET

(gdb) **n** // 从 breakpoint() 中出来，执行源码下一行代码。  
main.test (s=”haha”, x=1234, r=”test: haha 1234″) at main.go:11  
11 return r

(gdb) **info args** // 从参数信息中，我们可以看到命名返回参数的值。  
s = “haha”  
x = 1234  
r = “test: haha 1234″

(gdb) **x/3xw &r** // 查看 r 内存数据。(指针 8 + 长度 4)  
0x442139f48: 0×42121240 0×00000000 0x0000000f  
(gdb) x/15xb 0×42121240 // 查看字符串字节数组  
0×42121240: 0×74 0×65 0×73 0×74 0x3a 0×20 0×68 0×61  
0×42121248: 0×68 0×61 0×20 0×31 0×32 0×33 0×34

(gdb) **c** // 继续执行，进程结束。

Continuing.  
test: haha 1234  
[Inferior 1 (process 4088) exited normally]

(gdb) **q** // 退出 GDB