关于defer中调用recover的一些问题分析

本文分析在go语言中，defer语句调用recover出现不同结果的底层实现，操作系统为linux18.04，架构为amd64。在go中：

```
func X() {
  if err:=recover();err!=nil{
    fmt.Printf("%+v\n",err)
  }
}

func main(){
  defer X()
  panic(123)
}
```

上面这种写法能够让程序捕获panic而不至于终止，但是下面这样的写法却不行：

```
func main(){
  defer recover()
  panic(123)
}
```

下面从源码的分析其中的原因。第二种写法的汇编代码如下：

```
0x47d6cf <main.main+15>:  sub    $0x30,%rsp
   0x47d6d3 <main.main+19>: mov    %rbp,0x28(%rsp)
   0x47d6d8 <main.main+24>: lea    0x28(%rsp),%rbp
   0x47d6dd <main.main+29>: movl   $0x18,(%rsp)
   0x47d6e4 <main.main+36>: lea    0x32bf5(%rip),%rax        # 0x4b02e0
   0x47d6eb <main.main+43>: mov    %rax,0x8(%rsp)
   0x47d6f0 <main.main+48>: lea    0x38(%rsp),%rax
   0x47d6f5 <main.main+53>: mov    %rax,0x10(%rsp)
   0x47d6fa <main.main+58>: callq  0x42cf80 <runtime.deferproc>
   0x47d6ff <main.main+63>: test   %eax,%eax
   0x47d701 <main.main+65>: jne    0x47d721 <main.main+97>
   0x47d703 <main.main+67>: jmp    0x47d705 <main.main+69>
   0x47d705 <main.main+69>: lea    0xd054(%rip),%rax        # 0x48a760
   0x47d70c <main.main+76>: mov    %rax,(%rsp)
   0x47d710 <main.main+80>: lea    0x42f81(%rip),%rax        # 0x4c0698
   0x47d717 <main.main+87>: mov    %rax,0x8(%rsp)
   0x47d71c <main.main+92>: callq  0x42e2f0 <runtime.gopanic>
   0x47d721 <main.main+97>: nop
   0x47d722 <main.main+98>: callq  0x42d840 <runtime.deferreturn>
   0x47d727 <main.main+103>:  mov    0x28(%rsp),%rbp
   0x47d72c <main.main+108>:  add    $0x30,%rsp
   0x47d730 <main.main+112>:  retq
```

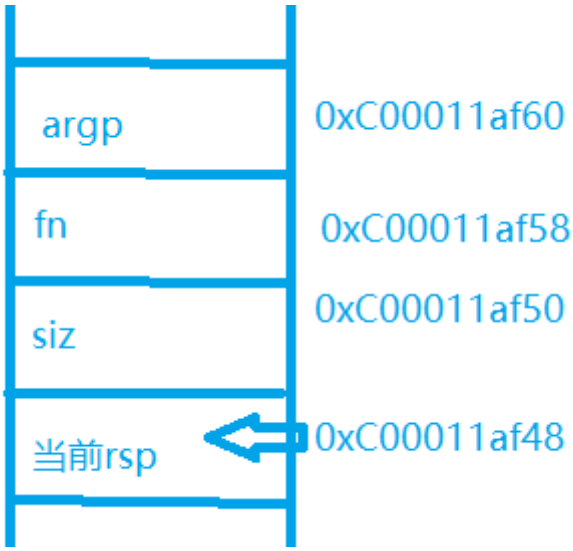
可以看到defer语句被编译成deferproc和deferreturn两条语句处理，而panic被编译成调用runtime.gopanic函数。

```
func deferproc(siz int32, fn *funcval) { // arguments of fn follow fn
```

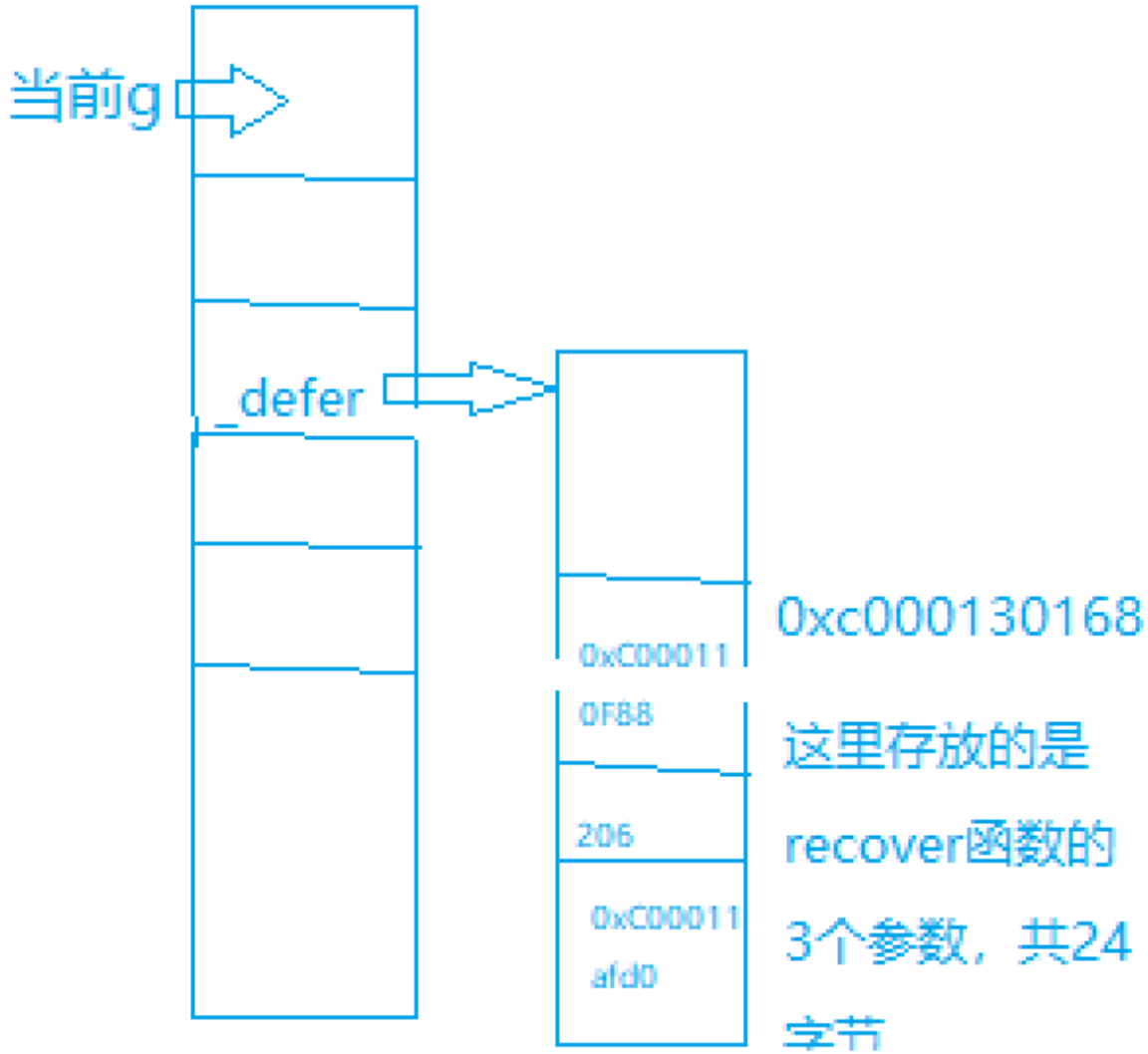
deferproc函数的siz参数为defer语句中调用的函数的参数的大小，单位是字节，recover有3个参数，在64位系统中需要24字节。注释中已经说明，函数的参数是紧随fn参数。因为最终这些参数都要被封装在_defer这个结构体变量中。在_defer这个结构体中，siz变量就是这里的siz参数，而函数的参数则放在_defer结构体的末尾。

```
argp := uintptr(unsafe.Pointer(&fn)) + unsafe.Sizeof(fn)
```

argp被放在了fn的后面，此时fn是在堆栈空间上，所以argp处于fn的上方。顺便说一句，这里的argp也是后面调用gorecover函数的参数。



此时的函数调用堆栈如图。deferproc函数的剩余部分所做的工作只剩new一个_defer变量填充相应的字段。



此时当前g的某些字段分布如上图。接下来函数返回继续执行panic语句，而panic语句被编译成对gopanic函数的调用。调用gopanic的参数e就是panic语句中的参数(整数123，这里的参数类型是接口类型，会被编译成runtime.eface,含有两个字段，放在堆栈空间中)。在gopanic函数中首先声明一个_panic变量，注意这个_panic变量是在栈上分配的。

```
p.arg = e
  p.link = gp._panic
  gp._panic = (*_panic)(noescape(unsafe.Pointer(&p)))
```

当前g的_panic变量是在这里赋值的。这里就是链表操作了，先把p的link指针指向当前g的_panic变量，在把g的_panic变量指向p。而

```
d._panic = (*_panic)(noescape(unsafe.Pointer(&p)))
```

则把g的_defer字段中的_panic也指向了当前g的_panic字段。

```
p.argp = unsafe.Pointer(getargp(0))
      reflectcall(nil, unsafe.Pointer(d.fn), deferArgs(d), uint32(d.siz),
uint32(d.siz))
```

这里第一句中，p是_panic类型的变量，当前的rsp寄存器指向0xC00011af08,getargp函数返回的就是这里的位置，即把p的argp字段指向了这里。第二句是反射调用d(_defer类型的变量)的fn函数，调用参数大小为siz，也就是24字节。reflectcall函数runtime/asm_amd64.s汇编文件中定义。该函数被展开后是一系列的跳转语句，根据参数的大小，小于等于32的执行call32，大于32小于等于64的执行call64，以此类推。当前的场景siz=24,当然是执行call32这个函数，函数的部分汇编代码如下图

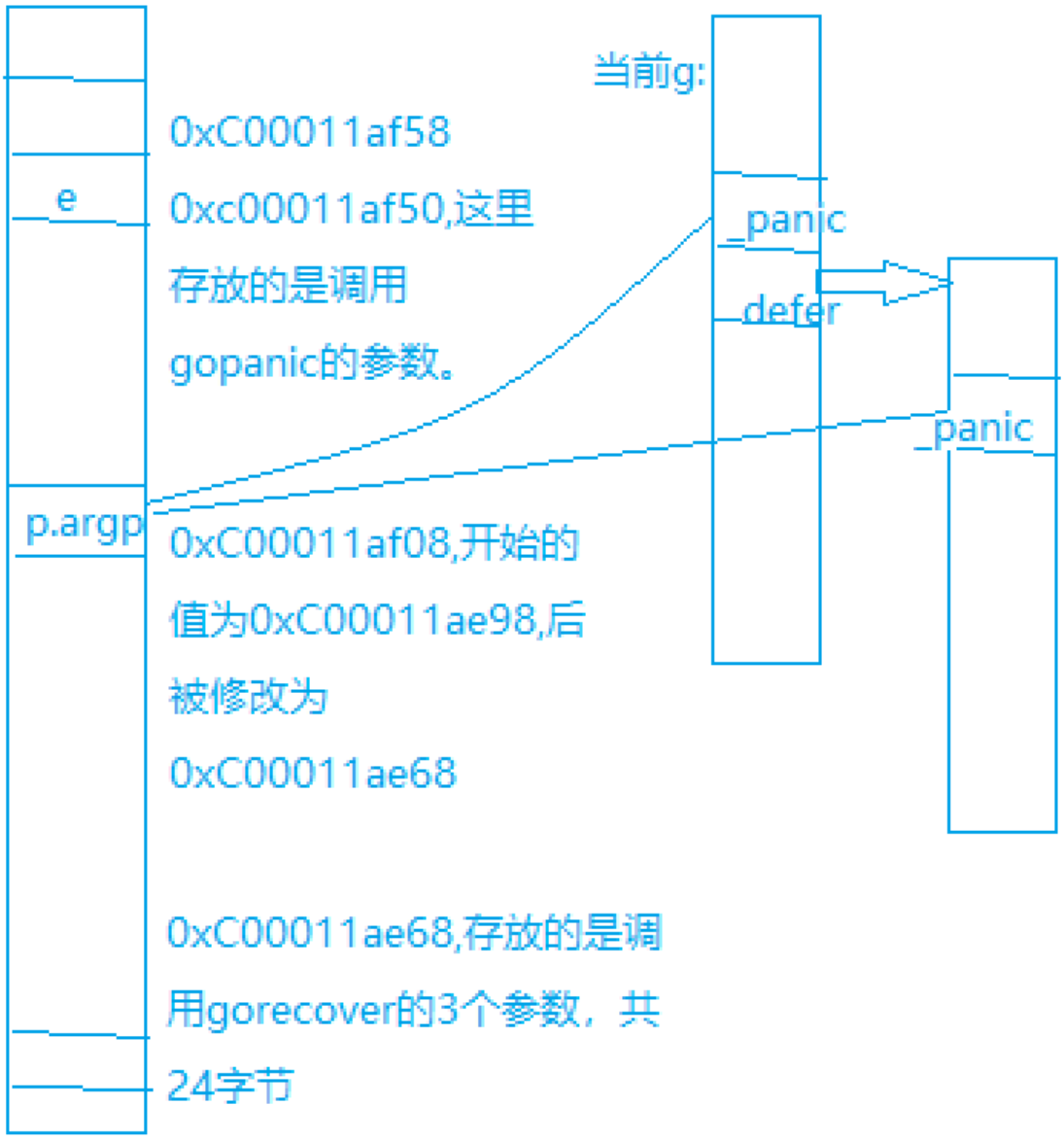
```
0x0000000000459a38      539     CALLFN(·call32, 32)
   0x00000000004599df <runtime.call32+15>:      48 83 ec 28     sub    $0x28,%rsp
   0x00000000004599e3 <runtime.call32+19>:      48 89 6c 24 20  mov    %rbp,0x20(%rsp)
   0x00000000004599e8 <runtime.call32+24>:      48 8d 6c 24 20  lea    0x20(%rsp),%rbp
   0x00000000004599ed <runtime.call32+29>:      48 8b 59 20     mov    0x20(%rcx),%rbx
   0x00000000004599f1 <runtime.call32+33>:      48 85 db        test   %rbx,%rbx
   0x00000000004599f4 <runtime.call32+36>:      75 49   jne    0x459a3f <runtime.call32+111>
   0x00000000004599f6 <runtime.call32+38>:      48 8b 74 24 40  mov    0x40(%rsp),%rsi
   0x00000000004599fb <runtime.call32+43>:      8b 4c 24 48     mov    0x48(%rsp),%ecx
   0x00000000004599ff <runtime.call32+47>:      48 89 e7        mov    %rsp,%rdi
   0x0000000000459a02 <runtime.call32+50>:      f3 a4   rep movsb %ds:(%rsi),%es:(%rdi)
   0x0000000000459a04 <runtime.call32+52>:      48 8b 54 24 38  mov    0x38(%rsp),%rdx
   0x0000000000459a09 <runtime.call32+57>:      ff 12   callq  *(%rdx)
   0x0000000000459a0b <runtime.call32+59>:      48 8b 54 24 30  mov    0x30(%rsp),%rdx
   0x0000000000459a10 <runtime.call32+64>:      48 8b 7c 24 40  mov    0x40(%rsp),%rdi
   0x0000000000459a15 <runtime.call32+69>:      8b 4c 24 48     mov    0x48(%rsp),%ecx
   0x0000000000459a19 <runtime.call32+73>:      8b 5c 24 4c     mov    0x4c(%rsp),%ebx
   0x0000000000459a1d <runtime.call32+77>:      48 89 e6        mov    %rsp,%rsi
   0x0000000000459a20 <runtime.call32+80>:      48 01 df        add    %rbx,%rdi
   0x0000000000459a23 <runtime.call32+83>:      48 01 de        add    %rbx,%rsi
   0x0000000000459a26 <runtime.call32+86>:      48 29 d9        sub    %rbx,%rcx
   0x0000000000459a29 <runtime.call32+89>:      e8 72 ff ff ff  callq  0x4599a0 <callRet>
   0x0000000000459a2e <runtime.call32+94>:      48 8b 6c 24 20  mov    0x20(%rsp),%rbp
   0x0000000000459a33 <runtime.call32+99>:      48 83 c4 28     add    $0x28,%rsp
   0x0000000000459a37 <runtime.call32+103>:     c3      retq
=> 0x0000000000459a38 <runtime.call32+104>:     e8 73 fd ff ff  callq  0x4597b0 <runtime.morestack_noctxt>
   0x0000000000459a3d <runtime.call32+109>:     eb 91   jmp    0x4599d0 <runtime.call32>
   0x0000000000459a3f <runtime.call32+111>:     48 8d 7c 24 30  lea    0x30(%rsp),%rdi
   0x0000000000459a44 <runtime.call32+116>:     48 39 3b        cmp    %rdi,(%rbx)
   0x0000000000459a47 <runtime.call32+119>:     75 ad   jne    0x4599f6 <runtime.call32+38>
   0x0000000000459a49 <runtime.call32+121>:     48 89 23        mov    %rsp,(%rbx)
   0x0000000000459a4c <runtime.call32+124>:     eb a8   jmp    0x4599f6 <runtime.call32+38>
   0x0000000000459a4e:  cc      int3
   0x0000000000459a4f:  cc      int3
```

0x459a09这句就是调用对应的defer函数，这里当然是对gorecover的调用。call32函数的所做的包括：

> 1.如果当前g的_panic不为nil，并且其argp的值与上一个函数栈帧(gopanic函数的调用)中的rsp的指向不等则跳转，否则会将_panic的argp指向当前的rsp。而从图中看出显然两者相等，rbx存放的是_panic，不为空，其指向的位置的值是0xc00011ae98，上一个函数栈帧中的rsp也是0xc00011ae98。当前rsp为0xc00011ae68，这是在调用call32的时候0x4599df处做的。
>
> 2.argp改变了，相应的参数也要复制过来，从0xC000130168处，复制siz=24字节。
>
> 3.调用对应的defer函数(0x459a09)

```
rax             0x0      0
rbx             0xc00011af08     824634879752
rcx             0xc000000180     824633721216
rdx             0x0      0
rsi             0x0      0
rdi             0xc00011ae98     824634879640
rbp             0xc00011ae88     0xc00011ae88
rsp             0xc00011ae68     0xc00011ae68
r8              0x7fffd1492a01   140736704621057
r9              0x203000 2109440
r10             0x8      8
r11             0x77     119
r12             0xf7     247
r13             0x0      0
r14             0x4be70a 4974346
r15             0x0      0
rip             0x459a44 0x459a44 <runtime.call32+116>
```

此时整个函数调用栈大致为：



剩下的就是对gorecover的调用，在gorecover函数中

```
if p != nil && !p.goexit && !p.recovered && argp == uintptr(p.argp) {
    p.recovered = true
    return p.arg
  }
```

argp为0xc00011af88，而p.argp为0xc00011ae68,显然不相等，所以这里p的recovered变量没有被赋值，所以在gopanic函数走到了对fatalpanic函数的调用是g结束。那为什么第一种写法没有问题呢？第一种写法对应的汇编代码为：

```
0x493360 <main.main>:        mov    %fs:0xfffffffffffffff8,%rcx
0x493369 <main.main+9>:      cmp    0x10(%rcx),%rsp
0x49336d <main.main+13>:     jbe    0x4933d1 <main.main+113>
0x49336f <main.main+15>:     sub    $0x68,%rsp
0x493373 <main.main+19>:     mov    %rbp,0x60(%rsp)
0x493378 <main.main+24>:     lea    0x60(%rsp),%rbp
0x49337d <main.main+29>:     movl   $0x0,0x10(%rsp)
0x493385 <main.main+37>:     lea    0x36114(%rip),%rax        # 0x4c94a0
0x49338c <main.main+44>:     mov    %rax,0x28(%rsp)
0x493391 <main.main+49>:     lea    0x10(%rsp),%rax
0x493396 <main.main+54>:     mov    %rax,(%rsp)
0x49339a <main.main+58>:     callq  0x42d3e0 <runtime.deferprocStack>
0x49339f <main.main+63>:     test   %eax,%eax
0x4933a1 <main.main+65>:     jne    0x4933c1 <main.main+97>
0x4933a3 <main.main+67>:     jmp    0x4933a5 <main.main+69>
0x4933a5 <main.main+69>:     lea    0xe034(%rip),%rax        # 0x4a13e0
0x4933ac <main.main+76>:     mov    %rax,(%rsp)
0x4933b0 <main.main+80>:     lea    0x48a19(%rip),%rax        # 0x4dbdd0
0x4933b7 <main.main+87>:     mov    %rax,0x8(%rsp)
0x4933bc <main.main+92>:     callq  0x42e690 <runtime.gopanic>
0x4933c1 <main.main+97>:     nop
0x4933c2 <main.main+98>:     callq  0x42dbe0 <runtime.deferreturn>
0x4933c7 <main.main+103>:    mov    0x60(%rsp),%rbp
0x4933cc <main.main+108>:    add    $0x68,%rsp
0x4933d0 <main.main+112>:    retq
0x4933d1 <main.main+113>:    callq  0x459c00 <runtime.morestack_noctxt>
0x4933d6 <main.main+118>:    jmp    0x493360 <main.main>
```

可以看到，这里与第二种不同的是，对defer语句的处理变成了对deferprocStack函数的调用(当然这不是使第一种写法能够正常结束的原因，只不过这里的_defer变量在栈上分配，相对于deferproc在堆上new一块内存更快)。

```
0x000000000049321f <main.X+47>:      48 8d 84 24 b0 00 00 00 lea    0xb0(%rsp),%rax
0x0000000000493227 <main.X+55>:      48 89 04 24     mov    %rax,(%rsp)
0x000000000049322b <main.X+59>:      e8 b0 bb f9 ff  callq  0x42ede0 <runtime.gorecover>
```

第二中写法里面，在call32函数中完成参数的复制之后直接对defer语句函数进行调用，call32函数没有重新开辟栈空间，直接调用了gorecover，由上图可以看出，在第一种写法里面，call32调用的是main.X函数，在X函数中开辟了新的栈空间，在调用gorecover时使用的都是上一个函数(这里是main.X)中开辟的栈传递参数， argp == uintptr(p.argp)条件自然成立，p.recovered被设置为true。在gopanic中走到了对recovery函数的调用，从而不会panic。