(2) 将传入的 reflect.Value 参数数组设置到栈上；

(3) 通过函数指针和输入参数调用函数；

(4) 从栈上获取函数的返回值。

我们将按照上面的顺序分析使用 reflect 进行函数调用的几个过程。

1. 参数检查

参数检查是通过反射调用方法的第一步。在参数检查期间我们会从反射对象中取出当前的函数指针 unsafe.Pointer，如果该函数指针是方法，那么我们会通过 reflect.methodReceiver 获取方法的接收者和函数指针：

```
func (v Value) call(op string, in []Value) []Value {
    t := (*funcType)(unsafe.Pointer(v.typ))
    ...
    if v.flag&flagMethod != 0 {
        rcvr = v
        rcvrtype, t, fn = methodReceiver(op, v, int(v.flag)>>flagMethodShift)
    } else {
        ...
    }
    n := t.NumIn()
    if len(in) < n {
        panic("reflect: Call with too few input arguments")
    }
    if len(in) > n {
        panic("reflect: Call with too many input arguments")
    }
    for i := 0; i < n; i++ {
        if xt, targ := in[i].Type(), t.In(i); !xt.AssignableTo(targ) {
            panic("reflect: " + op + " using " + xt.String() + " as type " + targ.String())
        }
    }
```

上述方法还会检查传入参数的个数以及参数的类型与函数签名中的类型是否匹配，任何参数不匹配都会导致整个程序崩溃中止。

2. 准备参数

我们对当前方法的参数完成验证后，就会进入函数调用的下一个阶段——为函数调用准备参数。前面介绍过 Go 语言的函数调用惯例，函数或者方法在调用时，所有参数都会被依次放到栈上。

```
    nout := t.NumOut()
    frametype, _, retOffset, _, framePool := funcLayout(t, rcvrtype)

    var args unsafe.Pointer
    if nout == 0 {
        args = framePool.Get().(unsafe.Pointer)
    } else {
        args = unsafe_New(frametype)
```