Go 中的非类型安全指针

Go 夜读 SIG 小组

2020-04-23

unsafe.Pointer 简介

- 类似 C 语言中的无类型指针 void*
- 借助 unsafe.Pointer 有时候可以挽回 Go 运行时（Go runtime）为了安全而牺牲的一些性能。
- 必须小心按照官方文档中的说明使用 unsafe.Pointer 。稍有不慎，将使 Go 类型系统（不包括非类型安全指针部分）精心设立内存安全壁垒的努力前功尽弃。
- 使用了 unsafe.Pointer 的代码不受 Go 1 兼容性保证。

unsafe.Pointer 相关类型转换编译规则

1. 一个类型安全指针值可以被显式转换为一个非类型安全指针类型（unsafe.Pointer），反之亦然。
2. 一个 uintptr 值可以被显式转换为一个非类型安全指针类型，反之亦然。

注意：这些规则是编译器接收的规则。满足这些规则的代码编译没问题，但并不意味着在运行的时候是安全的。在使用非类型安全指针时，必须遵循一些原则以防止不安全的情况发生。

使用 unsafe.Pointer 的基本运行时原则

1) 保证要使用的值在 unsafe 操作前后时时刻刻要被有效的指针引用着，无论类型安全指针还是非类型安全指针。否则此值有可能被垃圾回收器回收掉。
2) 任何指针都不应该引用未知内存快。

非类型安全指针相关的事实

- 事实一：非类型安全指针值是指针但 uintptr 值是整数。整数从不引用其它值。
- 事实二：不再被使用的内存块的回收时间点是不确定的
- 事实三：某些值的地址在程序运行中可能改变
- 事实四：一个值的生命范围可能并没有代码中看上去的大
- 事实五：*unsafe.Pointer 是一个类型安全指针类型，它的基类型是 unsafe.Pointer

简单介绍一下垃圾回收机制

三色算法：

1) 在每一轮垃圾回收过程的开始，所有的内存块将被标记为白色。然后垃圾回收器将所有开辟在栈和全局内存区上的内存块标记为灰色，并把它们加入一个灰色内存块列表。
2) 循环直到灰色内存块列表直到其为空：从个灰色内存块列表中取出一个内存块，并把它标记为黑色。然后扫描承载在此内存块上的指针值，并通过这些指针找到它们引用着的内存块。 如果一个引用着的内存块为白色的，则将其标记为灰色并加入灰色内存块列表；否则，忽略之。
3) 最后仍然标记为白色的内存快将被视为垃圾而回收掉。

事实一：非类型安全指针值是指针但 uintptr 值是整数

uintptr 值中时常用来存储内存地址值，但是一个 uintptr 值并不引用着存储于其中的地址处的值，所以此 uintptr 值仍在被使用的事实无法防止存储于其中的地址处的值被垃圾回收。

垃圾回收器同等对待类型安全指针和非类型安全指针。只有指针可以直接引用其它值。

事实二：不再被使用的内存块的回收时间点是不确定的

启动一轮新的垃圾回收过程的途径：

- GOGC 环境变量， runtime/debug.SetGCPercent
- 调用 runtime.GC 函数来手动开启
- 最大垃圾回收时间间隔为两分钟

并发，写屏障（ write barrier ）等细节。

事实三：某些值的地址在程序运行中可能改变

为了提高程序性能，每个协程维护着一个栈（一段连续内存块，64 位系统上初始为 2k）。在程序运行时，一个协程的栈的大小可能会根据需要而伸缩。当一个栈的大小改变时，runtime 需要开辟一段新的连续内存块，并把老的连续内存块上的值复制（移动）到新的连续内存块上，从而相应地，开辟在此栈上的指针值中存储的地址可能将改变（如果此地址值处于老的连续内存块上）。

即：目前开辟在栈上的值的地址可能会发生改变；开辟在栈上的的指针值中存储的值可能会自动改变。

事实四：一个值的生命范围可能并没有代码中看上去的大

```
type T struct {x int; y *[1<<23]byte}
func bar() {
    t := T{y: new([1<<23]byte)}
    p := uintptr(unsafe.Pointer(&t.y[0]))
    // 一个聪明的编译器能够觉察到值 t.y 将不会再被用到而回收之。
    *(*byte)(unsafe.Pointer(p)) = 1 // 危险操作！
    println(t.x) // ok 。继续使用值 t，但只使用 t.x 字段。
}
```


事实五：*unsafe.Pointer 是一个类型安全指针类型（这个对指针值的原子操作很有用）

```
func main() {
    type T struct {x int}
    var p *T
    var unsafePPT = (*unsafe.Pointer)(unsafe.Pointer(&p))
    atomic.StorePointer(unsafePPT, unsafe.Pointer(&T{123}))
    fmt.Println(p) // &{123}
}
```

使用模式 1：将类型 *T1 的一个值转换为非类型安全指针值，然后将此非类型安全指针值转换为类型 *T2 （要求：T1 的尺寸不小于 T2）

```
func Float64bits(f float64) uint64 {
    return *(*uint64)(unsafe.Pointer(&f))
}

func Float64frombits(b uint64) float64 {
    return *(*float64)(unsafe.Pointer(&b))
}
```

math 标准库包

使用模式 1：将类型 *T1 的一个值转换为非类型安全指针值，然后将此非类型安全指针值转换为类型 *T2 （要求：T1 的尺寸不小于 T2）

```
func ByteSlice2String(bs []byte) string {
    return *(*string)(unsafe.Pointer(&bs))
}
```

strings 标准库包中 strings.Builder.String() 方法的实现

标准库不是 100% 标准，轻微依赖于官方编译器和运行时（runtime）的实现。

使用模式 1：将类型 *T1 的一个值转换为非类型安全指针值，然后将此非类型安全指针值转换为类型 *T2 （要求：T1 的尺寸不小于 T2 ）

```
func String2ByteSlice(str string) []byte {
        return *(*[]byte)(unsafe.Pointer(&str))
}

x := String2ByteSlice("abc")
fmt.Println(x)              // [97 98 99]
fmt.Println(len(x), cap(x)) // 3 824634327128
```

```
type slice struct {
        data     unsafe.Pointer
        len, cap int
}
type string struct {
        data unsafe.Pointer
        len  int
}
```

违背了此模式

使用模式 1：将类型 *T1 的一个值转换为非类型安全指针值，然后将此非类型安全指针值转换为类型 *T2 （要求：T1 的尺寸不小于 T2）

```
type StringEx struct {string; cap int}

func String2ByteSlice(str string) []byte {
    se := StringEx{string: str, cap:len(str)}
    return *(*[]byte)(unsafe.Pointer(&se))
}
```

使用模式 2 ：将一个非类型安全指针值转换为一个 uintptr 值，然后使用此 uintptr 值

不是太有用

```
func main() {
    type T struct{a int}
    var t T
    fmt.Printf("%p\n", &t)                    // 0xc6233120a8
    println(&t)                              // 0xc6233120a8
    fmt.Printf("%x\n", uintptr(unsafe.Pointer(&t))) // c6233120a8
}
```

使用模式 3：将一个非类型安全指针转换为一个 uintptr 值，然后此 uintptr 值参与各种算术运算，再将算术运算的结果 uintptr 值转回非类型安全指针

有点 C 的感觉了

```
ptr2 = unsafe.Pointer(uintptr(ptr1) + offset)
ptr2 = unsafe.Pointer(uintptr(ptr1) &^ 7) // 8 字节对齐
```

要求：

1)转换前后的非类型安全指针（这里的 ptr1 和 ptr2 ）必须指向同一个内存块。

2)两次转换必须在同一条语句中 。

使用模式 3 ：例子 1

```
type T struct {x bool; y [3]int16}
const N = unsafe.Offsetof(T{}.y)
const M = unsafe.Sizeof(T{}.y[0])
func main() {
    t := T{y: [3]int16{123, 456, 789}}
    p := unsafe.Pointer(&t)
    ty2 := (*int16)(unsafe.Pointer(uintptr(p)+N+M+M))
    fmt.Println(*ty2) // 789
}
```


使用模式 3 ：例子 2

```
type T struct {x bool; y [3]int16}
var t = T{y: [3]int16{123, 456, 789}}
const N = unsafe.Offsetof(T{}.y)
func y2t(yAddr unsafe.Pointer) *T {
    return (*T)(unsafe.Pointer(uintptr(yAddr)-N))
}
func main() {
    fmt.Println(y2t(unsafe.Pointer(&t.y))) // &{false [123 456 789]}
}
```

使用模式 3：错误使用 1

```
type T struct {x bool; y [3]int16}
const N = unsafe.Offsetof(T{}.y)
const M = unsafe.Sizeof(T{}.y[0])
func main() {
    t := T{y: [3]int16{123, 456, 789}}
    p := unsafe.Pointer(&t)
    addr := uintptr(p)+N+M+M
    ty2 := (*int16)(unsafe.Pointer(addr))
    ...
```

有一处例外，
后面会讲

使用模式 3：错误使用 2

```
type T struct {x bool; y [3]int16}
const N = unsafe.Offsetof(T{}.y)
const M = unsafe.Sizeof(T{}.y[0])
func main() {
    t := T{y: [3]int16{123, 456, 789}}
    p := unsafe.Pointer(&t)
    ty2 := (*int16)(unsafe.Pointer(uintptr(p)+N+M+M+M))
    ...
// https://gfw.go101.org/article/unofficial-faq.html#final-zero-size-field
```

使用模式 3：错误使用 2

```
// 假设此函数不会被内联。
func DoSomething(addr uintptr) {
    ptr := unsafe.Poiner(addr)
    // 对处于传递进来的地址处的值进行读写 ...
    *(*int)(ptr) = 123
}

var a int
DoSomething(unsafe.Pointer(&a))
```


使用模式 4 ：需要将 reflect.Value.Pointer 或者 reflect.Value.UnsafeAddr 方法的 uintptr 返回值转换为非类型安全指针

设计目的：避免不引用 unsafe 包就可以将这两个方法的返回值（如果是 unsafe.Pointer 类型）转换为任何类型安全指针类型。

不立即转换为 unsafe.Pointer，将出现一个可能导致处于返回的地址处的内存块被回收掉的时间窗。

使用模式 4 ：需要将 reflect.Value.Pointer 或者 reflect.Value.UnsafeAddr 方法的 uintptr 返回值转换为非类型安全指针

```
p := (*int)(unsafe.Pointer(reflect.ValueOf(new(int)).Pointer())) // ok


// 而下面这样使用是危险的：
u := reflect.ValueOf(new(int)).Pointer()
// 在这个时刻，处于存储在 u 中的地址处的内存块可能会被回收掉。
p := (*int)(unsafe.Pointer(u))
```

使用模式 5 ： reflect.SliceHeader 或者 reflect.StringHeader 值的 Data 字段和非类型安全指针之间的相互转换

```
type SliceHeader struct {
    Data uintptr
    Len  int
    Cap  int
}
```

```
type StringHeader struct {
    Data uintptr
    Len  int
}
```

reflect.SliceHeader 和切片的内部结构一致；
reflect.StringHeader 和字符串的内部结构一致。
使用原则：不要凭空生成 SliceHeader 和 StringHeader，要从切片和字符串转换出它们。

使用模式 5 ：例子 1

```
a := [...]byte{'G', 'o', 'l', 'a', 'n', 'g'}
s := "Java"
hdr := (*reflect.StringHeader)(unsafe.Pointer(&s))
hdr.Data = uintptr(unsafe.Pointer(&a))
hdr.Len = len(a)
fmt.Println(s) // Golang
// 现在，字符串 s 和切片 a 共享着底层的 byte 字节序列
a[2], a[3], a[4], a[5] = 'o', 'g', 'l', 'e'
fmt.Println(s) // Google
```

编译没问题，也符合基本运行时原则。

但是不推荐这么做，因为这打破了对字符串的不变性的预期。

结果字符串不应传递给外部使用。

使用模式 5 ：例子 2

```
func String2ByteSlice(str string) (bs []byte) {
    strHdr := (*reflect.StringHeader)(unsafe.Pointer(&str))
    sliceHdr := (*reflect.SliceHeader)(unsafe.Pointer(&bs))
    sliceHdr.Data = strHdr.Data
    sliceHdr.Cap = strHdr.Len
    sliceHdr.Len = strHdr.Len
    runtime.KeepAlive(&str) //  这里的 KeepAlive 是必要的。
    return
}
```


使用模式 5 ：例子 2

```
type SliceHeader struct {
    Data unsafe.Pointer
    Len  int
    Cap  int
}
```

```
type StringHeader struct {
    Data unsafe.Pointer
    Len  int
}
```

则上一页中的 runtime.KeepAlive 调用不再必要

https://github.com/golang/go/issues/19367

使用模式 6：将非类型安全指针值转换为 uintptr 值并传递给 syscall.Syscall 函数调用

```
func DoSomething(addr uintptr) {...} // 危险


// syscall 标准库包
func Syscall(trap, a1, a2, a3 uintptr) (r1, r2 uintptr, err Errno) // ok
```

这是 syscall.Syscall 这样的函数的特权，编译器将对它们的调用特殊处理。此使用模式不适用于普通自定义函数。此使用模式也适用于 Windows 系统中的 syscall.Proc.Call 和 syscall.LazyProc.Call 系统调用。

总结

- 非类型安全机制可以帮助我们写出运行效率更高的代码
- 但是使用不当，将造成一些重现几率非常低的微妙的 bug
- 我们应该知晓当前的非类型安全机制规则和使用模式可能在以后的 Go 版本中完全失效。 当然，目前没有任何迹象表明这种变化将很快会来到。但是，一旦发生这种变化，前面列出的当前是正确的代码将变得不再安全甚至编译不通过。

参考资料

1. Go 101 项目： https://github.com/golang101/golang101
2. Go 101 官网： https://gfw.go101.org
3. Go 101 公众号