第 5 章

接　　口

接口在 Go 语言中扮演着非常重要的角色，它是多态、反射和类型断言等一众动态语言特性的基础。本章将通过反编译、runtime 源码分析等手段，逐步梳理清楚接口的底层实现。

4min

5.1　空接口

这里所谓的空接口并不是 nil，而是指不包含任何方法的接口，也就是 interface{}。在面向对象编程中，接口用来对行为进行抽象，也就是定义对象需要支持的操作，这组操作对应的就是接口中列出的方法。不包含任何方法的接口可以认为不要求对象支持任何操作，因此能够接受任意类型的赋值，所以 Go 语言的 interface{}什么都能装。

5.1.1　一个更好的 void *

如果用 unsafe.Sizeof()函数获取一个 interface{}类型变量的大小，在 64 位平台上是 16 字节，在 32 位平台上是 8 字节。interface{}类型本质上是个 struct，由两个指针类型的成员组成，在 runtime 中可以找到对应的 struct 定义，代码如下：

```
type eface struct {
    _type * _type
    data unsafe.Pointer
}
```

还有一个专门的类型转换函数 efaceOf()，该函数接受的参数是一个 interface{}类型的指针，返回值是一个 eface 类型的指针，内部实际只进行了一下指针类型的转换，也就说明 interface{}类型在内存布局层面与 eface 类型完全等价。efaceOf()函数的代码如下：

```
func efaceOf(ep * interface{}) * eface {
    return ( * eface)(unsafe.Pointer(ep))
}
```

接下来看一下 eface 的两个指针成员，data 字段比较好理解，它是一个 unsafe. Pointer 类型的指针，用来存储实际数据的地址。unsafe. Pointer 在含义上和 C 语言中的 void * 有些类似，只用来表明这是一个指针，并不限定指向的目标数据的类型，可以接受任意类型的地址。至于_type 类型，之前在探索变量逃逸和闭包的时候曾经见到过，当时是作为 runtime. newobject()函数的参数出现的，它在 Go 语言的 runtime 中被用来描述数据类型，笔者习惯称之为类型元数据。eface 的这个_type 字段用来描述 data 的类型元数据，也就是说它给出了 data 的数据类型。

例如，把一个 int 类型变量 n 的地址赋值给一个 interface{}类型的变量 e，代码如下：

```
//第 5 章/code_5_1.go
var n int
var e interface{} = &n
```

如图 5-1 所示，变量 e 的 data 字段存储的是变量 n 的地址，而变量 e 的_type 字段存储的是 * int 类型的类型元数据的地址。

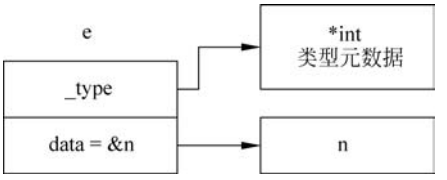

图 5-1 空接口变量 e 与赋值变量 n 的关系

与 void * 相比，interface{}通过多出来的这个_type 字段给出了数据的类型信息，程序在运行阶段可以基于这种类型信息对数据进行特定操作，因此 interface{}就相当于一个增强版的 void * 。

就变量 n 本身而言，它的类型信息只会被编译器使用，编译阶段参考这种类型信息来分配存储空间、生成机器指令，但是并不会把这种类型信息写入最终生成的可执行文件中。从内存布局的角度来讲，变量 n 在 64 位和 32 位平台分别占用 8 字节和 4 字节，占用的这些空间全部用来存放整型的值，没有任何空间被用来存放整型类型信息。

把变量 n 的地址赋值给 interface{}类型的变量 e 的这个操作，意味着编译器要把 * int 的类型元数据生成出来，并把其地址赋给变量 e 的_type 字段，这些类型元数据会被写入最终的可执行文件，程序在运行阶段即取即用。这个简单的赋值操作实际上完成了类型信息的萃取。

为了能够方便地通过反编译进行验证，将第 5 章/code_5_1. go 稍微修改一下，代码如下：

```
//第 5 章/code_5_2.go
func p2e(p * int) (e interface{}) {
    e = p
    return
}
```

然后进行编译和反编译，得到相应的汇编代码如下：

```
$ go tool compile -trimpath="`pwd`=>" -l -p gom eface.go
$ go tool objdump -S -s '^gom.p2e$' eface.o
TEXT gom.p2e(SB) gofile..eface.go
        return
  0x7b0       488d0500000000        LEAQ 0(IP), AX    [3:7]R_PCREL:type.*int
  0x7b7       4889442410            MOVQ AX, 0x10(SP)
  0x7bc       488b442408            MOVQ 0x8(SP), AX
  0x7c1       4889442418            MOVQ AX, 0x18(SP)
  0x7c6       c3                    RET
```

虽然看起来很简单，还是把它转换为等价的伪代码，这样更加直观，代码如下：

```
func p2e(n *int) (e eface) {
    e._type = &type.*int
    e.data = unsafe.Pointer(n)
    return
}
```

其中的 type.*int 就是需要的类型元数据，编译器在生成的指令中为它的地址预留了位置，等到链接阶段生成可执行文件时链接器会填写上实际的地址。

提到变量的类型，一般指的是声明类型，例如变量 n 的声明类型是 int。在 Go 这种强类型语言中，变量的声明类型是不能改变的，即使通过类型转换得到一个新的变量，原变量的类型还是不会改变。对于 interface{}类型的变量 e，它的声明类型是 interface{}，这一点也是不能改变的。变量 e 就像是一个容器，可以装载任意类型的数据，并通过_type 字段记录数据的类型，无论装载什么类型的数据，容器本身的类型不会改变。因为_type 会随着变量 e 装载不同类型的数据而发生改变，所以后文中将它称为变量 e 的动态类型，并相应地把变量 e 的声明类型称为静态类型。

5.1.2 类型元数据

在 C 语言中类型信息主要存在于编译阶段，编译器从源码中得到具体的类型定义，并记录到相应的内存数据结构中，然后根据这些类型信息进行语法检查、生成机器指令等。例如 x86 整数加法和浮点数加法采用完全不同的指令集，编译器根据数据的类型来选择。这些类型信息并不会被写入可执行文件，即使作为符号数据被写入，也是为了方便调试工具，并不会被语言本身所使用。Go 与 C 语言不同的是，在设计之初就支持面向对象编程，还有其他一些动态语言特征，这些都要求运行阶段能够获得类型信息，所以语言的设计者就把类型信息用统一的数据结构来描述，并写入可执行文件中供运行阶段使用，这就是所谓的类型元数据。

既然已经不止一次遇到_type 这种类型元数据类型，这里就来简单看一下它的具体定

义。摘抄自 Go 1.15 版本的 runtime 源码，代码如下：

```
type _type struct {
    size       uintptr
    ptrdata    uintptr
    hash       uint32
    tflag      tflag
    align      uint8
    fieldAlign uint8
    kind       uint8
    equal func(unsafe.Pointer, unsafe.Pointer) bool
    gcdata     *Byte
    str        nameOff
    ptrToThis typeOff
}
```

各个字段的含义及主要用途如表 5-1 所示。

表 5-1 _type 各字段的含义及主要用途

字段	含义及主要用途
size	表示此类型的数据需要占用多少字节的存储空间，runtime 中很多地方会用到它，最典型的就是内存分配的时候，例如 newobject()、mallocgc()
ptrdata	ptrdata 表示数据的前多少字节包含指针，用来在应用写屏障时优化范围大小。例如某个 struct 类型在 64 位平台上占用 32 字节，但是只有第 1 个字段是指针类型，这个值就是 8，剩下的 24 字节就不需要写屏障了。GC 进行位图标记的时候，也会用到该字段
hash	当前类型的哈希值，runtime 基于这个值构建类型映射表，加速类型比较和查找
tflag	额外的类型标识，目前由 4 个独立的二进制位组合而成。tflagUncommon 表明这种类型元数据结构后面有个紧邻的 uncommontype 结构，uncommontype 主要在自定义类型定义方法集时用到。tflagExtraStar 表示类型的名称字符串有个前缀的 *，因为对于程序中的大多数类型 T 而言，*T 也同样存在，复用同一个名称字符串能够节省空间。tflagNamed 表示类型有名称。tflagRegularMemory 表示相等比较和哈希函数可以把该类型的数据当成内存中的单块区间来处理
align	表示当前类型变量的对齐边界
fieldAlign	表示当前类型的 struct 字段的对齐边界
kind	表示当前类型所属的分类，目前 Go 语言的 reflect 包中定义了 26 种有效分类
equal	用来比较两个当前类型的变量是否相等
gcdata	和垃圾回收相关，GC 扫描和写屏障用来追踪指针
str	偏移，通过 str 可以找到当前类型的名称等文本信息
ptrToThis	偏移，假设当前类型为 T，通过它可以找到类型 *T 的类型元数据

_type 提供了适用于所有类型的最基本的描述，对于一些更复杂的类型，例如复合类型 slice 和 map 等，runtime 中分别定义了 maptype、slicetype 等对应的结构。例如 slicetype 就是由一个用来描述类型本身的_type 结构和一个指向元素类型的指针组成，代码如下：

```
type slicetype struct {
    typ _type
    elem * _type
}
```

Go 语言允许为自定义类型实现方法，这些方法的相关信息也会被记录到自定义类型的元数据中，一般称为类型的方法集信息。在梳理_type 结构的各个字段时，没有发现任何跟方法集有关的字段，那么 runtime 是如何以_type 为起点来找到方法集信息的呢？

考虑到 Go 语言只允许为自定义类型实现方法，所以要找到元数据中的方法集信息，就要从自定义类型出发。自定义类型，也就是代码中使用 type 关键字定义的类型，示例代码如下：

```
type Integer int
```

在上述代码中，int 本身是内置类型，不允许为其实现方法，而基于 int 定义的 Integer 是个自定义类型。还记得_type 结构的 tflag 字段是几个标志位，当 tflagUncommon 这一位为 1 时，表示类型为自定义类型。从 runtime 的源码可以发现，_type 类型有一个 uncommon()方法，对于自定义类型可以通过此方法得到一个指向 uncommontype 结构的指针，也就是说编译器会为自定义类型生成一个 uncommontype 结构，例如上述自定义类型 Integer 的类型元数据结构如图 5-2 所示。

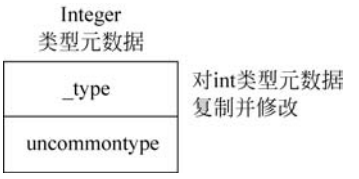

图 5-2 自定义类型 Integer 的类型元数据结构

uncommontype 结构的定义代码如下：

```
type uncommontype struct {
    pkgpath nameOff
    mcount  uint16 //number of methods
    xcount  uint16 //number of exported methods
    moff    uint32 //offset from this uncommontype to [mcount]method
    _       uint32 //unused
}
```

通过 pkgpath 可以知道定义该类型的包名称，mcount 表示该类型共有多少个方法，xcount 表示有多少个方法被导出，也就是首字母大写使包外可访问。moff 是个偏移值，那里就是方法集的元数据，也就是一组 method 结构构成的数组。例如，若为自定义类型 Integer 定义两个方法，它的类型元数据及其 method 数组的内存布局如图 5-3 所示。

method 数组中每个 method 结构对应一个方法，代码如下：

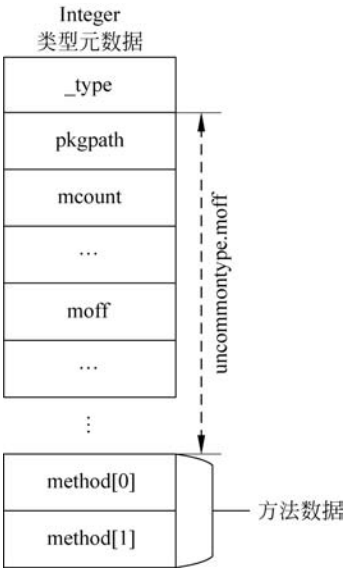

图 5-3 Integer 类型元数据及其 method 数组的内存布局

```
type method struct {
    name nameOff
    mtyp typeOff
    ifn textOff
    tfn textOff
}
```

通过 name 偏移能够找到方法的名称字符串，mtyp 偏移处是方法的类型元数据，进一步可以找到参数和返回值相关的类型元数据。若自定义类型有 A()、B()、C()3 个方法(如图 5-4 所示)，则 method 数组会按照 name 升序排列，运行阶段可以高效地进行二分查找。

ifn 是供接口调用的方法地址，tfn 是正常的方法地址，这两个方法地址有什么不同呢？ifn 的接收者类型一定是指针，而 tfn 的接收者类型跟源代码中的实现一致，这里先不进行过多的解释，在 5.2.3 节中会深入分析这两者的不同。

以上这些类型元数据都是在编译阶段生成的，经过链接器的处理后被写入可执行文件中，runtime 中的类型断言、反射和内存管理等都依赖于这些元数据，本章的后续内容都与这些类型元数据有着密切的关系。

5.1.3 逃逸与装箱

由于 interface{}的 data 字段是个指针，存储的是数据的地址，所以不可避免地也会跟变量逃逸扯上关系。在进行逃逸分析的时候，直接把 interface{}当作原始数据类型的指针来看待即可，效果是等价的，此处不再赘述。

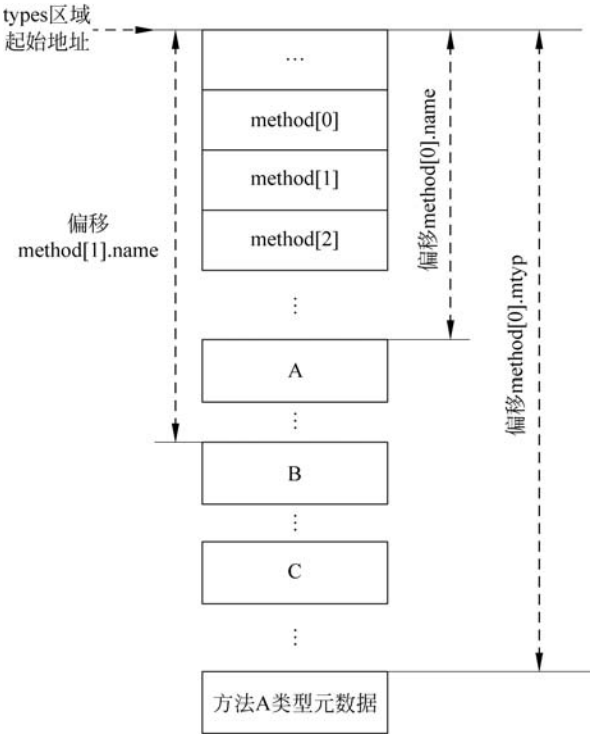

图 5-4 method 数组排序

接下来有一个需要仔细探索的问题：data 字段是个指针，那么它是如何接收来自一个值类型的赋值的呢？示例代码如下：

```
//第 5 章/code_5_3.go
n := 10
var e interface{} = n
```

在上述代码中变量 e 的数据结构如图 5-5 所示。

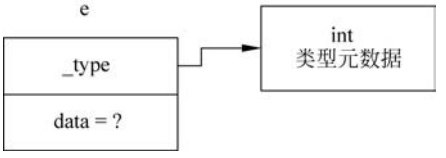

图 5-5 interface{}类型的变量 e 的数据结构

e. data 这里存储的是什么还真不太好猜测，还是直接反编译一下比较简单。依旧把第 5 章/code_5_3.go 放到一个函数中，代码如下：

```
//第 5 章/code_5_4.go
func v2e(n int) (e interface{}) {
    e = n
    return
}
```

反编译后得到汇编代码如下：

```
$ go tool objdump -S -s '^gom.v2e$' eface.o
TEXT gom.v2e(SB) gofile..eface.go
func v2e(n int) (e interface{}) {
  0xb0a       65488b0c2528000000      MOVQ GS:0x28, CX
  0xb13       488b8900000000          MOVQ 0(CX), CX          [3:7]R_TLS_LE
  0xb1a       483b6110                CMPQ 0x10(CX), SP
  0xb1e       763c                    JBE 0xb5c
  0xb20       4883ec18                SUBQ $0x18, SP
  0xb24       48896c2410              MOVQ BP, 0x10(SP)
  0xb29       488d6c2410              LEAQ 0x10(SP), BP
        e = n
  0xb2e       488b442420              MOVQ 0x20(SP), AX
  0xb33       48890424                MOVQ AX, 0(SP)
  0xb37       e800000000              CALL 0xb3c              [1:5]R_CALL:runtime.convT64
  0xb3c       488b442408              MOVQ 0x8(SP), AX
        return
  0xb41       488d0d00000000          LEAQ 0(IP), CX          [3:7]R_PCREL:type.int
  0xb48       48894c2428              MOVQ CX, 0x28(SP)
  0xb4d       4889442430              MOVQ AX, 0x30(SP)
  0xb52       488b6c2410              MOVQ 0x10(SP), BP
  0xb57       4883c418                ADDQ $0x18, SP
  0xb5b       c3                      RET
func v2e(n int) (e interface{}) {
  0xb5c       e800000000              CALL 0xb61
[1:5]R_CALL:runtime.morestack_noctxt
  0xb61       eba7                    JMP gom.v2e(SB)
```

虽然代码篇幅不太长，但还是转换成等价的伪代码比较容易理解，代码如下：

```
func v2e(n int) (e eface) {
entry:
    gp := getg()
    if SP <= gp.stackguard0 {
        goto morestack
    }
    e.data = runtime.convT64(n)
    e._type = &type.int
```

```
    return
morestack:
    runtime.morestack_noctxt()
    goto entry
}
```

忽略与栈增长相关的代码，真正感兴趣的就是为变量 e 的两个成员赋值的这两行代码。先把变量 n 的值作为参数调用 runtime.convT64()函数，并把返回值赋给了 e.data。又把 type.int 的地址赋给了 e._type。后者倒是比较容易理解，因为变量 e 的动态类型是变量 n 的类型，即 int，但这个 runtime.convT64()函数的逻辑还需要再看一下，看一看它的返回值究竟是什么。runtime.convT64()函数的源代码如下：

```
func convT64(val uint64) (x unsafe.Pointer) {
    if val < uint64(len(staticuint64s)) {
        x = unsafe.Pointer(&staticuint64s[val])
    } else {
        x = mallocgc(8, uint64Type, false)
        *(*uint64)(x) = val
    }
    return
}
```

当 val 的值小于 staticuint64s 的长度时，直接返回 staticuint64s 中第 val 项的地址。否则就通过 mallocgc()函数分配一个 uint64，把 val 的值赋给它并返回它的地址。这个 staticuint64s 如图 5-6 所示，是个长度为 256 的 uint64 数组，每个元素的值都跟下标一致，存储了 0～255 这 256 个值，主要用来避免常用数字频繁地进行堆分配。

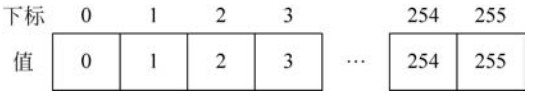

图 5-6　staticuint64s 数组

整体来看 convT64()函数的功能，实际上就是堆分配一个 uint64，并且将 val 参数作为初始值赋给它。由于示例中变量 n 的值为 10，在 staticuint64s 的长度范围内，所以变量 e 的 data 字段存储的就是 staticuint64s 中下标为 10 的存储空间的地址，如图 5-7 所示。

通过 staticuint64s 这种优化方式，能够反向推断出：被 convT64 分配的这个 uint64，它的值在语义层面是不可修改的，是个类似 const 的常量，这样设计主要是为了跟 interface{}配合来模拟装载值。interface{}被设计成一个容器，但它本质上是个指针，可以直接装载地址，用来实现装载值，实际的内存要分配在别的地方，并把内存地址存储在这里。convT64()函数的作用就是分配这个存储值的内存空间，实际上 runtime 中有一系列这类函数，如 convT32()、convTstring()和 convTslice()等。

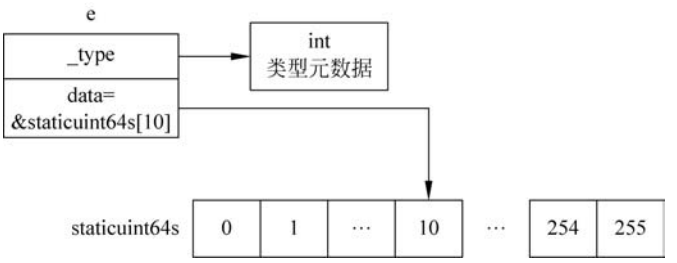

图 5-7 变量 e 的数据结构

至于为什么这个值不可修改，因为 interface{}只是一个容器，它支持把数据装入和取出，但是不支持直接在容器里修改。这有些类似于 Java 和 C＃中的自动装箱，只不过 interface{}是个万能包装类。

那么值类型装箱就一定会进行堆分配吗？这个问题也需要验证。既然已经知道逃逸会造成堆分配，那就构造一个值类型装箱但不逃逸的场景，示例代码如下：

```
//第 5 章/code_5_5.go
func fn(n int) bool {
    return notNil(n)
}

func notNil(a interface{}) bool {
    return a != nil
}
```

编译时需要禁止内联优化，编译器还能够通过 notNil()函数的代码实现判定有没有发生逃逸，反编译 fn()函数得到的汇编代码如下：

```
$ go tool objdump -S -s '^gom.fn$' eface.o
TEXT gom.fn(SB) gofile..eface.go
func fn(n int) bool {
  0xfd6         65488b0c2528000000      MOVQ GS:0x28, CX
  0xfdf         488b8900000000          MOVQ 0(CX), CX    [3:7]R_TLS_LE
  0xfe6         483b6110                CMPQ 0x10(CX), SP
  0xfea         764a                    JBE 0x1036
  0xfec         4883ec28                SUBQ $0x28, SP
  0xff0         48896c2420              MOVQ BP, 0x20(SP)
  0xff5         488d6c2420              LEAQ 0x20(SP), BP
        return notNil(n)
  0xffa         488b442430              MOVQ 0x30(SP), AX
  0xfff         4889442418              MOVQ AX, 0x18(SP)
  0x1004        488d0500000000          LEAQ 0(IP), AX    [3:7]R_PCREL:type.int
  0x100b        48890424                MOVQ AX, 0(SP)
  0x100f        488d442418              LEAQ 0x18(SP), AX
  0x1014        4889442408              MOVQ AX, 0x8(SP)
```

```
  0x1019        e800000000            CALL 0x101e    [1:5]R_CALL:gom.notNil
  0x101e        0fb6442410            MOVZX 0x10(SP), AX
  0x1023        88442438              MOVB AL, 0x38(SP)
  0x1027        488b6c2420            MOVQ 0x20(SP), BP
  0x102c        4883c428              ADDQ $0x28, SP
  0x1030        c3                    RET
func fn(n int) bool {
  0x1031        0f1f440000            NOPL 0(AX)(AX*1)
  0x1036        e800000000            CALL 0x103b    [1:5]R_CALL:runtime.morestack_noctxt
  0x103b        eb99                  JMP gom.fn(SB)
```

转换为等价的伪代码如下：

```
func fn(n int) bool {
entry:
    gp := getg()
    if SP <= gp.stackguard0 {
        goto morestack
    }
    v := n
    return notNil(eface{_type: &type.int, data: &v})
morestack:
    runtime.morestack_noctxt()
    goto entry
}
```

伪代码中 fn()函数的调用栈如图 5-8 所示，注意局部变量 v，它实际上是被编译器采用隐式方式分配的，被用作变量 n 的值的副本，却并没有分配到堆上。

interface{}在装载值的时候必须单独复制一份，而不能直接让 data 存储原始变量的地址，因为原始变量的值后续可能会发生改变，这就会造成逻辑错误。

上面的例子总算证明了装箱不一定进行堆分配，是否堆分配还是要经过逃逸分析。只有值类型装箱后又涉及逃逸的情况，才会用到 runtime 中的一系列 convT()函数。

关于不包含任何方法的空接口就先研究到这里，下面来看一下包含方法的非空接口。

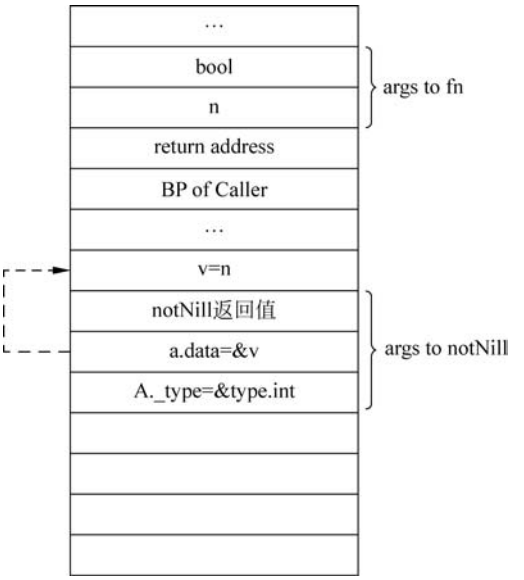

图 5-8　fn()函数的调用栈

5.2 非空接口

与空接口对应，非空接口指的是至少包含一种方法的接口，就像 io. Reader 和 io. Writer。非空接口通过一组方法对行为进行抽象，从而隔离具体实现达到解耦的目的。Go 的接口比 Java 等语言中的接口更加灵活，自定义类型不需要通过 implements 关键字显式地标明自己实现了某个接口，只要实现了接口中所有的方法就实现了该接口。也只有实现了接口中所有的方法，才算是实现了该接口。

本节探索一下 Go 语言中非空接口的底层实现，后文中提到的接口均指非空接口，为了能够加以区分，不再将 interface{}称为空接口，而是直接称为 interface{}类型。

5.2.1 动态派发

在面向对象编程中，接口的一个核心功能是支持多态，实际上就是方法的动态派发。调用接口的某个方法时，调用者不需要知道背后对象的具体类型就能调用对象的指定方法。例如类型 A 和 B 都实现了 fmt. Stringer 接口，示例代码如下：

```
//第 5 章/code_5_6.go
type A struct{}

type B struct{}

func (A) String() string {
    return "This is A"
}

func (B) String() string {
    return "This is B"
}

func toString(o fmt.Stringer) string {
    return o.String()
}

func main() {
    println(toString(A{})) //This is A
    println(toString(B{})) //This is B
}
```

其中 toString()函数的实现者并不需要知道参数 o 背后的具体类型，接口机制会在运行阶段自动完成方法调用的动态派发，所以 toString(A{})会调用类型 A 的 String()方法，进而返回字符串"This is A"，而 toString(B{})则返回字符串"This is B"。下面分析一下动

态派发如何实现。

1. 方法地址静态绑定

要进行方法(函数)调用,有两点需要确定:一是方法的地址,也就是在代码段中的指令序列的起始地址;二是参数及调用约定,也就是要传递什么参数及如何传递的问题(通过栈或者寄存器),返回值的读取也包含在调用约定范畴内。调用约定及编译器如何根据调用约定来生成相关指令,在第 3 章已经探索过了,这里的重点是如何确定目标方法的地址。

首先来看一个不使用接口而直接通过自定义类型的对象实例调用其方法的例子,代码如下:

```
//go:noinline
func ReadFile(f *os.File, b []byte) (n int, err error) {
    return f.Read(b)
}
```

上述 ReadFile()函数实际上只调用了 *os.File 类型的 Read()方法,为了方便后续反编译,禁止编译器对该函数进行内联。对 build 得到的可执行文件进行反编译,得到对应的汇编代码如下:

```
$ go tool objdump -S -s '^main.ReadFile$' gom.exe
TEXT main.ReadFile(SB) C:/gopath/src/fengyoulin.com/gom/main.go
func ReadFile(f *os.File, b []byte) (n int, err error) {
  0x4b4240      65488b0c2528000000    MOVQ GS:0x28, CX
  0x4b4249      488b8900000000        MOVQ 0(CX), CX
  0x4b4250      483b6110              CMPQ 0x10(CX), SP
  0x4b4254      7662                  JBE 0x4b42b8
  0x4b4256      4883ec40              SUBQ $0x40, SP
  0x4b425a      48896c2438            MOVQ BP, 0x38(SP)
  0x4b425f      488d6c2438            LEAQ 0x38(SP), BP
        return f.Read(b)
  0x4b4264      488b442448            MOVQ 0x48(SP), AX
  0x4b4269      48890424              MOVQ AX, 0(SP)
  0x4b426d      488b442450            MOVQ 0x50(SP), AX
  0x4b4272      4889442408            MOVQ AX, 0x8(SP)
  0x4b4277      488b442458            MOVQ 0x58(SP), AX
  0x4b427c      4889442410            MOVQ AX, 0x10(SP)
  0x4b4281      488b442460            MOVQ 0x60(SP), AX
  0x4b4286      4889442418            MOVQ AX, 0x18(SP)
  0x4b428b      e87035ffff            CALL os.(*File).Read(SB)
  0x4b4290      488b442420            MOVQ 0x20(SP), AX
  0x4b4295      488b4c2428            MOVQ 0x28(SP), CX
  0x4b429a      488b542430            MOVQ 0x30(SP), DX
  0x4b429f      4889442468            MOVQ AX, 0x68(SP)
  0x4b42a4      48894c2470            MOVQ CX, 0x70(SP)
```

```
  0x4b42a9      4889542478              MOVQ DX, 0x78(SP)
  0x4b42ae      488b6c2438              MOVQ 0x38(SP), BP
  0x4b42b3      4883c440                ADDQ $ 0x40, SP
  0x4b42b7      c3                      RET
func ReadFile(f * os.File, b []byte) (n int, err error) {
  0x4b42b8      e8a3e8faff              CALL runtime.morestack_noctxt(SB)
  0x4b42bd      eb81                    JMP main.ReadFile(SB)
```

可以看到 CALL 指令直接调用了 os.(*File).Read()方法，地址以 Offset 的形式编码在指令中。实际上这个地址是编译器在 OBJ 文件中预留了空间，然后由链接器填写实际的 Offset，有兴趣的读者可以自己反编译 OBJ 文件查看，这里不再赘述。与汇编代码等价的 Go 风格的伪代码如下：

```
func ReadFile(f * os.File, b []byte) (n int, err error) {
entry:
    gp := getg()
    if SP <= gp.stackguard0 {
        goto morestack
    }
    return os.(*File).Read(f, b)
morestack:
    runtime.morestack_noctxt()
    goto entry
}
```

排除掉这些栈增长代码，就剩下一个再普通不过的函数(方法)调用了。从汇编语言的角度来看，上述方法的调用是通过 CALL 指令＋相对地址实现的，方法地址在可执行文件构建阶段就确定了，一般将这种情况称为方法地址的静态绑定。

显而易见，这种地址静态绑定的方式无法支持方法调用的动态派发，因为编译阶段并不知道对象的具体类型，所以无法确定要绑定到何种方法。对于动态派发来讲，编译阶段能够确定的是要调用的方法的名字，以及方法的原型(参数与返回值列表)。以第 5 章/code_5_6.go 中的 toString()函数为例，要调用的方法名字是 String，没有入参，有一个 string 类型的返回值。实际上有这些信息就足够了，运行阶段根据这些信息就能完成动态派发。

2. 动态查询类型元数据

至于动态派发的代码实现，可以有很多种不同版本。先不去管 Go 语言到底是如何实现的，如果让我们来设计，可以怎么做呢？

我们假设非空接口的数据结构与 eface 相同，同样包含一个类型元数据指针和一个数据指针。5.1 节已经简单地分析了与类型元数据相关的数据结构，知道自定义类型的类型元数据中存有方法集信息，方法集信息是一组 method 结构构成的数组，通过它可以找到对应方法的方法名、参数和返回值的类型，以及代码的地址。method 结构的代码如下：

```
type method struct {
    name nameOff
    mtyp typeOff
    ifn textOff
    tfn textOff
}
```

类型元数据中的 method 数组是按照方法名升序排列的，可以直接应用二分法查找。运行阶段利用这些信息就可以根据方法名和原型动态绑定方法地址了。假如现在有一个 io. Reader 类型的接口变量 r，其背后动态类型是 * os. File，代码如下：

```
var r io.Reader = f
n, err := r.Read(buf)
```

首先，可以通过变量 r 得到 * os. File 的类型元数据，如图 5-9 所示，然后根据方法名称 Read 以二分法查找匹配的 method 结构，找到后再根据 method. mtyp 得到方法本身的类型元数据，最后对比方法原型是否一致（参数和返回值的类型、顺序是否一致）。如果原型一致，就找到了目标方法，通过 method. ifn 字段得到方法的地址，然后就像调用普通函数一样调用就可以了。

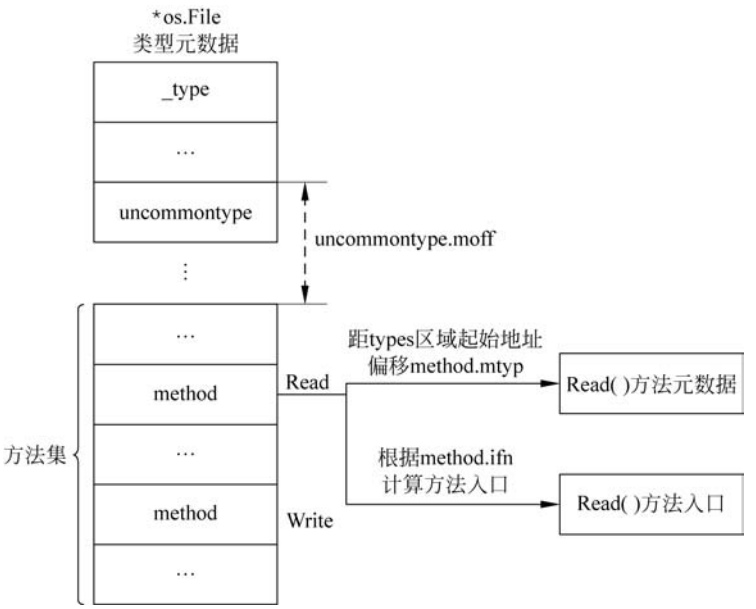

图 5-9 * os. File 的类型元数据

单就动态派发而言，这种方式确实可以实现，但是有一个明显的问题，那就是效率低，或者说性能差。跟地址静态绑定的方法调用比起来，原本一条 CALL 指令完成的事情，这里又多出

了一次二分查找加方法原型匹配，增加的开销不容小觑，可能会造成动态派发的方法比静态绑定的方法多一倍开销甚至更多，所以必须进行优化。不能在每次方法调用前都到元数据中去查找，尽量做到一次查找、多次使用，这里可以一定程度上参考C++的虚函数表实现。

3. C++虚函数机制

C++中的虚函数机制跟接口的思想很相似，编程语言允许父类指针指向子类对象，当通过父类的指针来调用虚函数时，就能实现动态派发。具体实现原理就是，编译器为每个包含虚函数的类都生成一张虚函数表，实际上是个地址数组，按照虚函数声明的顺序存储了各个虚函数的地址。此外还会在类对象的头部安插一个虚指针(GCC安插在头部，其他编译器或有不同)，指向类型对应的虚函数表。运行阶段通过类对象指针调用虚函数时，会先取得对象中的虚指针，进一步找到对象类型对应的虚函数表，然后基于虚函数声明的顺序，以数组下标的方式从表中取得对应函数的地址，这样整个动态派发过程就完成了。

人们经常在父类中只声明一组纯虚函数，也就是不实现函数体，这种只包含一组纯虚函数的类就更符合接口的设计思想了。例如，将父类Type用作接口，声明两个纯虚函数，两个子类A和B分别继承自父类Type，并且实现这两个虚函数，相当于实现了接口，示例代码如下：

```
//第5章/code_5_7.cpp
class Type {
public:
    virtual string Name() = 0;
    virtual size_t Size() = 0;
};

class A : public Type {
public:
    string Name() {
        return "A";
    }
    size_t Size() {
        return sizeof( * this);
    }
};

class B : public Type {
public:
    string Name() {
        return "B";
    }
    size_t Size() {
        return sizeof( * this);
    }
private:
    int somedata;
};
```

可以测试多态的效果，测试代码如下：

```
//第 5 章/code_5_8.cpp
Type * pts[2] = {new A, new B};
for(int i = 0; i < 2; ++i) {
    cout << pts[i] -> Name() << "," << pts[i] -> Size() << endl;
}
```

在笔者的 64 位计算机上，输出结果如下：

```
A,8
B,16
```

图 5-10 以在上述代码中的 pts 为起点，展示出 A、B 的对象实例、各自虚指针及虚函数表在内存中的关联关系，这样就能一目了然地看懂 C++虚函数的动态派发原理了。

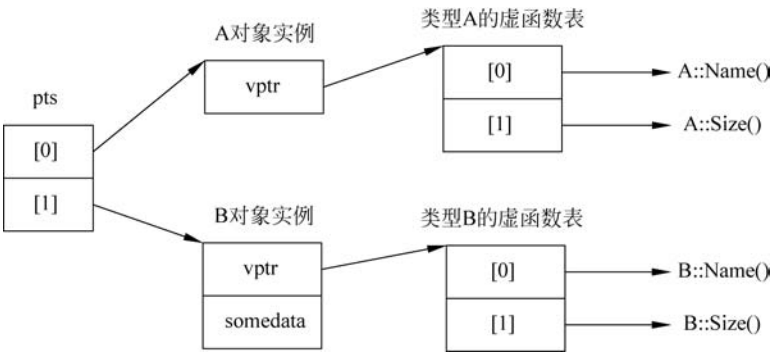

图 5-10 C++虚函数动态派发示例

运行阶段通过父类指针调用虚函数时，并不需要关心指向的是哪个子类。数组 pts 的元素类型是 Type *，运行阶段先通过 pts[0]和 pts[1]找到子类对象中的虚指针 vptr，再通过 vptr 最终定位到子类类型的虚函数表，根据函数声明的顺序按下标取得函数的实际地址。两个指针加一个数组，就完成了整个动态派发的核心逻辑，效率还是非常高的。

参考 C++的虚函数表思想，再回过头来看 Go 语言中接口的设计，如果把这种基于数组的函数地址表应用在接口的实现中，基本就能消除每次查询地址造成的性能开销。显然这里需要对 eface 结构进行扩展，加入函数地址表相关字段，经过扩展的 eface 姑且称作 efacex，代码如下：

```
type efacex struct {
    tab * struct {
        _type * _type
        fun [1]uintptr //方法数
    }
```

```
    data unsafe.Pointer
}
```

把原本的类型元数据指针_type和新添加的方法地址数组fun打包到一个struct中，并用这个struct的地址替换掉eface中原本的_type字段，得到修改后的efacex，如图5-11所示。

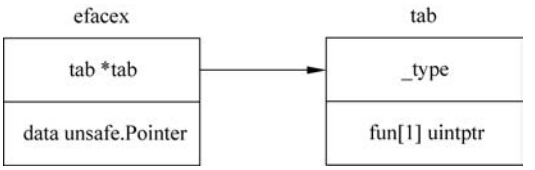

图5-11 参照C++虚函数机制修改后的非空接口数据结构

添加的fun数组相当于C++的虚函数表，这个数组的长度与接口中方法的个数一致，是动态分配的。在struct的最后放置一个动态长度的数组，这是C语言中常用的技巧。什么时候为fun数组赋值呢？当然是在为整个efacex结构赋值的时候最合适，示例代码如下：

```
//第5章/code_5_9.go
f, _ := os.Open("gom.go")
var rw io.ReadWriter
rw = f
```

从f到rw这个看似简单的赋值，至少要展开成如下几步操作：①根据rw接口中方法的个数动态分配tab结构，这里有两个方法，fun数组的长度是2。②从*os.File的方法集中找到Read()方法和Write()方法，把地址写入fun数组对应下标。③把*os.File的元数据地址赋值给tab._type。④把f赋值给data，也就是数据指针。赋值后rw的数据结构如图5-12所示。

这样一来，只需要在为接口变量赋值的时候对方法集进行查找，后续调用接口方法的时候，就可以像C++的虚函数那样直接按数组下标读取地址了。

实际上，fun数组也不用每次都重新分配和初始化，从指定具体类型到指定接口类型变量的赋值，运行阶段无论发生多少次，每次生成的fun数组都是相同的。例如从*os.File到io.ReadWriter的赋值，每次都会生成一个长度为2的fun数组，数组的两个元素分别用于存储(*os.File).Read和(*os.File).Write的地址。也就是说通过一个确定的接口类型和一个确定的具体类型，就能够唯一确定一个fun数组，因此可以通过一个全局的map将fun数组进行缓存，这样就能进一步减少方法集的查询，从而优化性能。

本节结合C++的虚函数机制，简单地推演了一下动态派发的实现原理，跟Go语言的实现已经很接近了，接下来看一下Go语言中的具体实现。

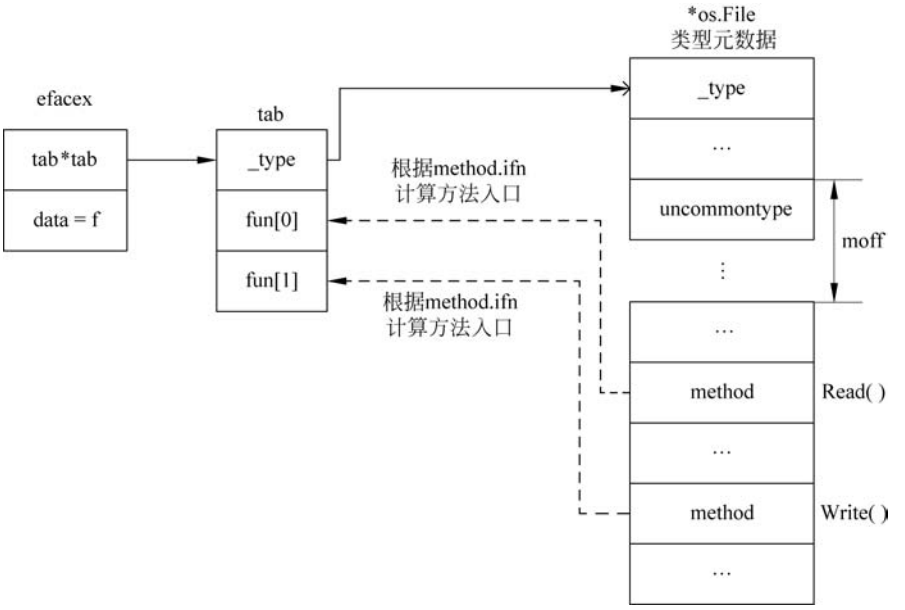

图 5-12 基于 efacex 设计的非空接口变量 rw 赋值后的数据结构

5.2.2 具体实现

5.2.1 节中为了加入地址数组 fun，把原本用于 interface{}的 eface 结构扩展成了 efacex，实际上在 Go 语言的 runtime 中与非空接口对应的结构类型是 iface，代码如下：

```
type iface struct {
    tab * itab
    data unsafe.Pointer
}
```

因为也是通过数据指针 data 来装载数据的，所以也会有逃逸和装箱发生。其中的 itab 结构就包含了具体类型的元数据地址_type，以及等价于虚函数表的方法地址数组 fun，除此之外还包含了接口本身的类型元数据地址 inter，代码如下：

```
type itab struct {
    inter * interfacetype
    _type * _type
    hash  uint32
    _     [4]byte
    fun   [1]uintptr
}
```

根据 5.1 节对类型元数据的简单介绍，从_type 到 uncommontype，再到[mcount]

method,已经找到了自定义类型的方法集。下面再来看一下运行时动态生成 itab 的相关逻辑。

1. 接口类型元数据

首先看一下接口类型的元数据信息对应的数据结构,代码如下:

```
type interfacetype struct {
    typ     _type
    pkgpath name
    mhdr    []imethod
}
```

除去最基本的 typ 字段,pkgpath 表示接口类型被定义在哪个包中,mhdr 是接口声明的方法列表。imethod 结构的代码如下:

```
type imethod struct {
    name nameOff
    ityp typeOff
}
```

比自定义类型的 method 结构少了方法地址,只包含方法名和类型元数据的偏移。这些偏移的实际类型为 int32,与指针的作用一样,但是 64 位平台上比使用指针节省一半空间。以 ityp 为起点,可以找到方法的参数(包括返回值)列表,以及每个参数的类型信息,也就是说这个 ityp 是方法的原型信息。

第 5 章/code_5_9.go 中非空接口类型的变量 rw 的数据结构如图 5-13 所示。

2. 如何获得 itab

运行阶段可通过 runtime.getitab 函数来获得相应的 itab,该函数被定义在 runtime 包中的 iface.go 文件中,函数原型的代码如下:

```
func getitab(inter *interfacetype, typ *_type, canfail bool) *itab
```

前两个参数 inter 和 typ 分别是接口类型和具体类型的元数据,canfail 表示是否允许失败。如果 typ 没有实现 inter 要求的所有方法,则 canfail 为 true 时函数返回 nil,canfail 为 false 时就会造成 panic。对应到具体的语法就是 comma ok 风格的类型断言和普通的类型断言,代码如下:

```
r, ok := a.(io.Reader) //comma ok
r := a.(io.Reader) //有可能造成 panic
```

上述代码第一行就是 comma ok 风格的类型断言,如果 a 没有实现 io.Reader 接口,则 ok 为 false。第二行就不同了,如果 a 没有实现 io.Reader 接口,就会造成 panic。

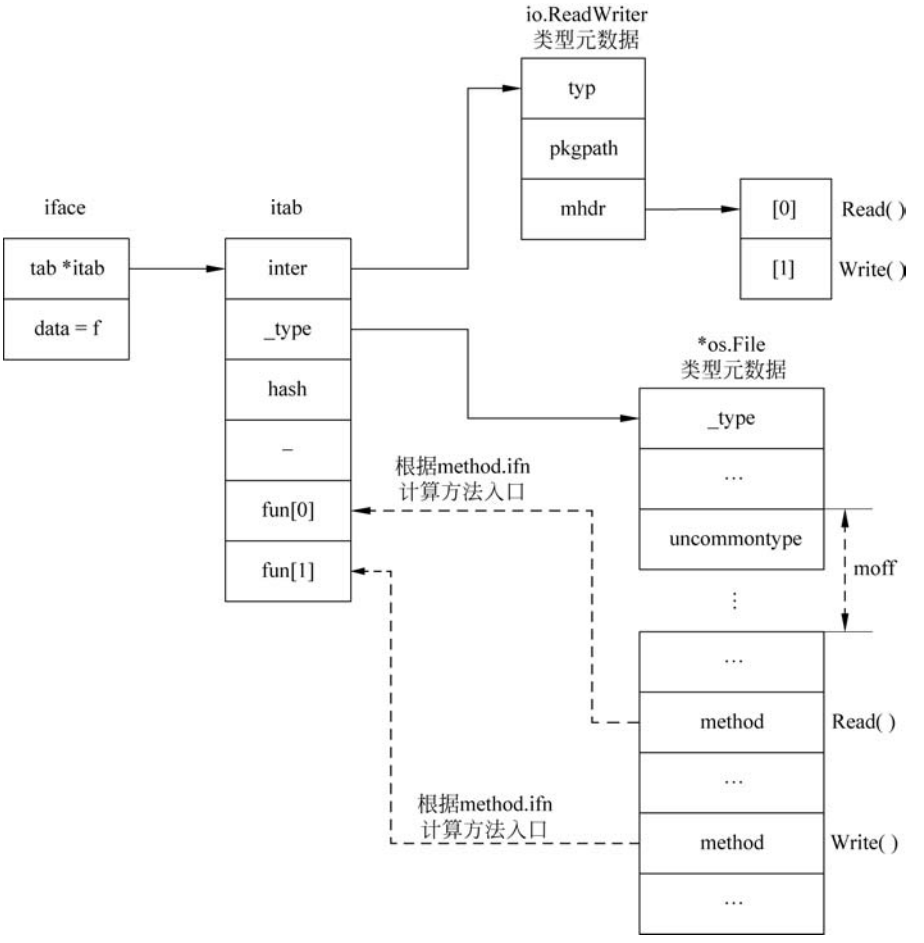

图 5-13　io. ReadWriter 类型的变量 rw 的数据结构

getitab()函数的代码摘抄自 Go 语言 runtime 源码，代码如下：

```
func getitab(inter * interfacetype, typ * _type, canfail bool) * itab {
    if len(inter.mhdr) == 0 {
        throw("internal error - misuse of itab")
    }
    if typ.tflag&tflagUncommon == 0 {
        if canfail {
            return nil
        }
        name := inter.typ.nameOff(inter.mhdr[0].name)
        panic(&TypeAssertionError{nil, typ, &inter.typ, name.name()})
    }
    var m * itab
```

```
    t := (*itabTableType)(atomic.Loadp(unsafe.Pointer(&itabTable)))
    if m = t.find(inter, typ); m != nil {
        goto finish
    }
    lock(&itabLock)
    if m = itabTable.find(inter, typ); m != nil {
        unlock(&itabLock)
        goto finish
    }
    m = (*itab)(persistentalloc(unsafe.Sizeof(itab{}) + uintptr(len(inter.mhdr) - 1) *
sys.PtrSize, 0, &memstats.other_sys))
    m.inter = inter
    m._type = typ
    m.hash = 0
    m.init()
    itabAdd(m)
    unlock(&itabLock)
finish:
    if m.fun[0] != 0 {
        return m
    }
    if canfail {
        return nil
    }
    panic(&TypeAssertionError{concrete: typ, asserted: &inter.typ, missingMethod: m.init
()})
}
```

函数的主要逻辑如下：①校验 inter 的方法列表长度不为 0，为没有方法的接口生成 itab 是没有意义的。②通过 typ.tflag 标志位来校验 typ 为自定义类型，因为只有自定义类型才能有方法集。③在不加锁的前提下，以 inter 和 typ 作为 key 查找 itab 缓存 itabTable，找到后就跳转到⑤。④加锁后再次查找缓存，如果没有就通过 persistentalloc()函数进行持久化分配，然后初始化 itab 并调用 itabAdd 添加到缓存中，最后解锁。⑤通过 itab 的 fun[0]是否为 0 来判断 typ 是否实现了 inter 接口，如果没实现，则根据 canfail 决定是否造成 panic，若实现了，则返回 itab 地址。

判断 itab.fun[0]是否为零，也就是判断第一个方法的地址是否有效，因为 Go 语言会把无效的 itab 也缓存起来，主要是为了避免缓存穿透。基于一个确定的接口类型和一个确定的具体类型，就能够唯一确定一个 itab，如图 5-14 所示。按照一般的思路，只有具体类型实现了该接口，才能得到一个 itab，进而缓存起来。这样会有个问题，假如具体类型没有实现该接口，但是运行阶段有大量这样的类型断言，缓存中查不到对应的 itab，就会每次都查询元数据的方法列表，从而显著影响性能，所以 Go 语言会把有效、无效的 itab 都缓存起来，通过 fun[0]加以区分。

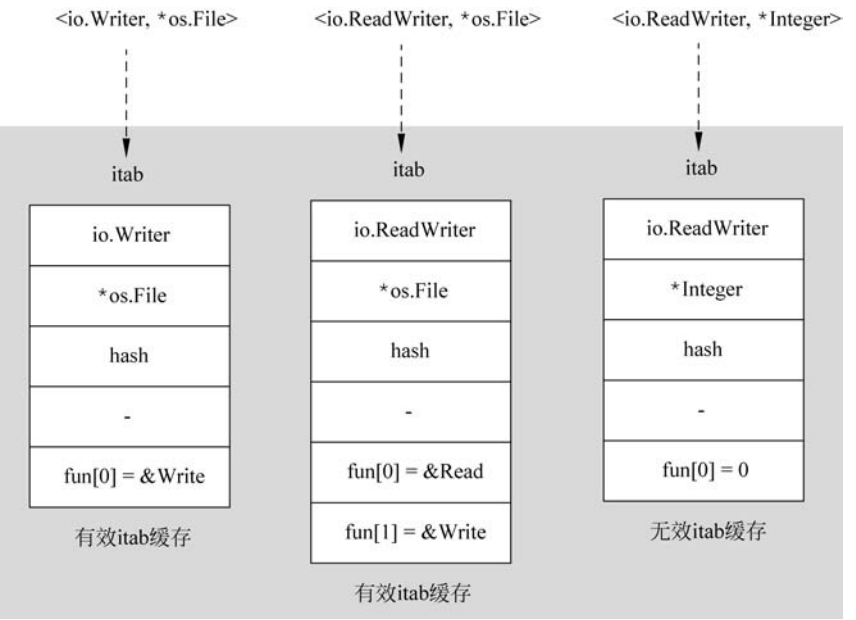

图 5-14　interfacetype 和_type 与 itab 的对应关系

3. itab 缓存

itabTable 就是 runtime 中 itab 的全局缓存，它本身是个 itabTableType 类型的指针，itabTableType 的代码如下：

```
type itabTableType struct {
    size    uintptr
    count   uintptr
    entries [itabInitSize] * itab
}
```

其中 entries 是实际的缓存空间，size 字段表示缓存的容量，也就是 entries 数组的大小，count 表示实际已经缓存了多少个 itab。entries 的初始大小是通过 itabInitSize 指定的，这个常量的值为 512。当缓存存满以后，runtime 会重新分配整个 struct，entries 数组是 itabTableType 的最后一个字段，可以无限增大它的下标来使用超出容量大小的内存，只要在 struct 之后分配足够的空间就够了，这也是 C 语言里常用的手法。

itabTableType 被实现成一个散列表，如图 5-15 所示。查找和插入操作使用的 key 是由接口类型元数据与动态类型元数据组合而成的，哈希值计算方式为接口类型元数据哈希值 inter. typ. hash 与动态类型元数据哈希值 typ. hash 进行异或运算。

方法 find()和 add()分别负责实现 itabTableType 的查找和插入操作，方法 add()操作内部不会扩容存储空间，重新分配操作是在外层实现的，因此对于 find()方法而言，已经插入的内容不会再被修改，所以查找时不需要加锁。方法 add()操作需要在加锁的前提下进

行，getitab()函数是通过调用itabAdd()函数来完成添加缓存的，itabAdd()函数内部会按需对缓存进行扩容，然后调用add()方法。因为缓存扩容需要重新分配itabTableType结构，为了并发安全，使用原子操作更新itabTable指针。加锁后立刻再次查询也是出于并发的考虑，避免其他协程已经将同样的itab添加至缓存。

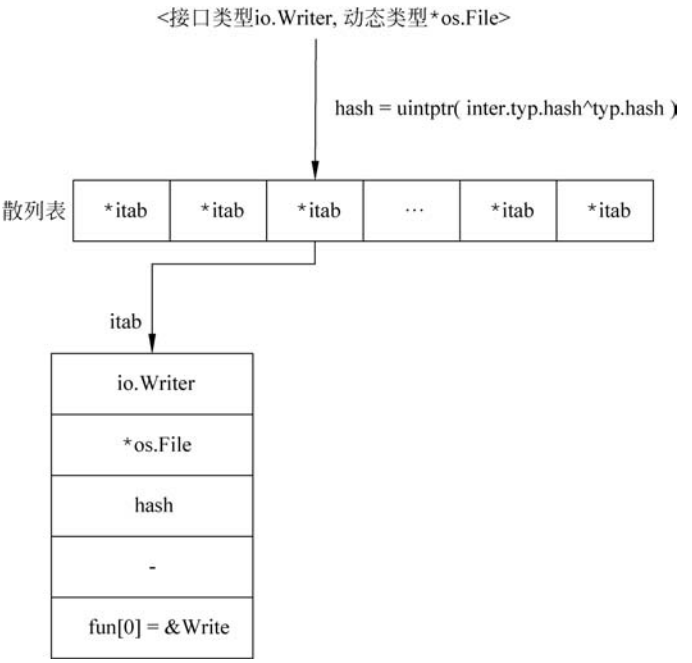

图 5-15 itabTableType哈希表

通过persistentalloc()函数分配的内存不会被回收，分配的大小为itab结构的大小加上接口方法数减一个指针的大小，因为itab中的fun数组声明的长度为1，已经包含了一个指针，分配空间时只需补齐剩下的即可。

还有一个值得一提，就是itab类型的init方法，这里为了节省篇幅，不再摘抄对应的源码。init()函数内部就是遍历接口的方法列表和具体类型的方法集，来寻找匹配的方法的地址。虽然遍历操作使用了两层嵌套循环，但是方法列表和方法集都是有序的，两层循环实际上都只需执行一次。匹配方法时还会考虑方法是否导出，以及接口和具体类型所在的包。如果是导出的方法则直接匹配成功，如果方法未导出，则接口和具体类型需要定义在同一个包中，方可匹配成功。最后需要再次强调的是，对于匹配成功的方法，地址取的是method结构中的ifn字段，具体的细节在5.2.3节中会继续分析，关于方法集的探索就先到这里。

4min

5.2.3 接收者类型

5.2.1节和5.2.2节中都提到了具体类型方法元数据中的ifn字段，该字段存储的是专门供接口使用的方法地址。所谓专门供接口使用的方法，实际上就是个接收者类型为指针

的方法。还记不记得第 4 章中分析 OBJ 文件时，发现编译器总是会为每个值接收者方法包装一个指针接收者方法？这也就说明，接口是不能直接使用值接收者方法的，这是为什么呢？

5.2.2 节已经看过了接口的数据结构 iface，它包含一个 itab 指针和一个 data 指针，data 指针存储的就是数据的地址。对于接口来讲，在调用指针接收者方法时，传递地址是非常方便的，也不用关心数据的具体类型，地址的大小总是一致的。假如通过接口调用值接收者方法，就需要通过接口中的 data 指针把数据的值复制到栈上，由于编译阶段不能确定接口背后的具体类型，所以编译器不能生成相关的指令来完成复制，进而无法调用值接收者方法。

有些读者可能还记得 3.4 节讲到的 runtime.reflectcall()函数，它能够在运行阶段动态地复制参数并完成函数调用。如果基于 reflectcall()函数，能不能实现通过接口调用值接收者方法呢？

肯定是可以实现的，接口的 itab 中有具体类型的元数据，确实能够应用 reflectcall()函数，但是有个明显的问题，那就是性能太差。跟几条用于传参的 MOV 指令加一条普通的 CALL 指令相比，reflectcall()函数的开销太大了，所以 Go 语言选择为值接收者方法生成包装方法。对于代码中的值接收者方法，类型元数据 method 结构中的 ifn 和 tfn 的值是不一样的，指针接收者方法的 ifn 和 tfn 是一样的。

比较有意思的是，从类型元数据来看，T 和 * T 是不同的两种类型。接收者类型为 T 的所有方法，属于 T 的方法集。因为编译器自动包装指针接收者方法的关系，* T 的方法集包含所有方法，也就是所有接收者类型为 T 的方法加上所有接收者类型为 * T 的方法。我们可以用一段代码来实际验证一下二者的关系，代码如下：

```
//第 5 章/code_5_10.go
package main
import (
    "fmt"
    "strconv"
    "unsafe"
)
type Integer int
func (i Integer) Value() float64 {
    return float64(i)
}
func (i Integer) String() string {
    return strconv.Itoa(int(i))
}

type Number interface {
    Value() float64
    String() string
}
```

```
func main() {
    i := Integer(0)
    fmt.Println(Methods(i))
    fmt.Println(Methods(&i))
    var n Number = i
    p := (*[5]unsafe.Pointer)((*face)(unsafe.Pointer(&n)).t)
    fmt.Println((*p)[3], (*p)[4])
}

func Methods(a interface{}) (r []Method) {
    e := (*face)(unsafe.Pointer(&a))
    u := uncommon(e.t)
    if u == nil {
        return nil
    }
    s := methods(u)
    r = make([]Method, len(s))
    for i := range s {
        r[i].Name = name(nameOff(e.t, s[i].name))
        r[i].Type = String(typeOff(e.t, s[i].mtyp))
        r[i].IFn = textOff(e.t, s[i].ifn)
        r[i].TFn = textOff(e.t, s[i].tfn)
    }
    return
}

type Method struct {
    Name string
    Type string
    IFn unsafe.Pointer
    TFn unsafe.Pointer
}

type face struct {
    t unsafe.Pointer
    d unsafe.Pointer
}

//go:linkname uncommon reflect.(*rtype).uncommon
func uncommon(t unsafe.Pointer) unsafe.Pointer

//go:linkname methods reflect.(*uncommonType).methods
func methods(u unsafe.Pointer) []method
```

```
type method struct {
    name int32
    mtyp int32
    ifn int32
    tfn int32
}

//go:linkname nameOff reflect.(*rtype).nameOff
func nameOff(t unsafe.Pointer, off int32) unsafe.Pointer

//go:linkname typeOff reflect.(*rtype).typeOff
func typeOff(t unsafe.Pointer, off int32) unsafe.Pointer

//go:linkname textOff reflect.(*rtype).textOff
func textOff(t unsafe.Pointer, off int32) unsafe.Pointer

//go:linkname String reflect.(*rtype).String
func String(t unsafe.Pointer) string

//go:linkname name reflect.name.name
func name(n unsafe.Pointer) string
```

其中 Number 接口声明了两个方法，即 Value()方法和 String()方法。自定义 Integer 实现了这两种方法，并且接收者类型都是值类型。为了直接解析类型元数据以获得 ifn 和 tfn 的值，示例中使用了 linkname 机制来调用 reflect 包中的私有函数，还用了 unsafe 包访问内存。在 amd64 平台上，用 Go 1.15 可以成功编译上述代码，运行结果如下：

```
$ ./code_5_10.exe
[{String func() string 0xcf5fe0 0xcf59a0} {Value func() float64 0xcf6080 0xcf5980}]
                                          //第 1 行输出
[{String func() string 0xcf5fe0 0xcf5fe0} {Value func() float64 0xcf6080 0xcf6080}]
                                          //第 2 行输出
0xcf5fe0 0xcf6080                         //第 3 行输出
```

第 1 行输出打印出了 Integer 类型的方法集，String()和 Value()这两个方法各自的 IFn 和 TFn 都不相等，这是因为 IFn 指向接收者为指针类型的方法代码，而 TFn 指向接收者为值类型的方法代码。

第 2 行输出打印出了 * Integer 类型的方法集，这两个方法各自的 IFn 和 TFn 是相等的，都与第 1 条指令中同名方法的 IFn 的值相等。

第 3 行输出打印出了 Number 接口 itab 中 fun 数组中的两个方法地址，与第 1 行输出 Integer 方法集中对应方法的 IFn 的值一致。Integer 和 * Integer 类型方法集的关系如图 5-16 所示。

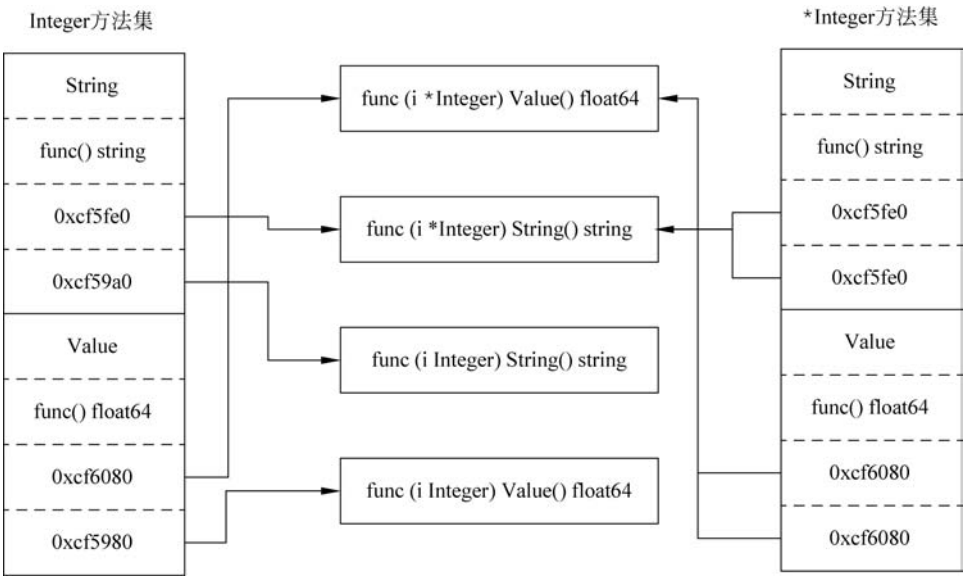

图 5-16 Integer 和 * Integer 类型的方法集

有一点需要格外注意，虽然把 i 赋值给 Number 类型的接口 n 后，n 的 itab 最终使用的是这一对接收者为指针类型的方法，但这是通过查找 Integer 的方法集查到的，语义角度还是 Integer 类型实现了 Number 接口。如果把 &i 赋值给 n，编译器和 runtime 才会从 * Integer 的方法集中查找。因为编译器会为代码中所有的值接收者方法包装生成对应的指针接收者方法，所以 * Integer 的方法集是 Integer 方法集的超集，也就是 Integer 类型实现的所有接口，即 * Integer 类型都实现了。反之不然，从语义角度无法为指针接收者方法包装生成对应的值接收者方法，因为原始的数据地址在值接收者方法中已经丢失。

通过以上示例，还能够证明一点，Integer 和 * Integer 的方法集及 Number 接口的 itab 中的方法都是按名称升序排列的，与代码中声明和实现的顺序无关，这和 5.1.2 节讲方法集时从 runtime 源码中看到的逻辑是一致的。

5.2.4 组合式继承

在第 4 章讲解方法的时候，曾经探索过基于嵌入的组合式继承，当时发现编译器会对继承的方法进行包装。因为自定义类型继承来的方法会影响到实现了哪些接口，所以本节再来回顾一下组合式继承，从方法集的角度进行分析，示例代码如下：

```
//第 5 章/code_5_11.go
package inherit

type A int
```

```
func (a A) Value() int {
    return int(a)
}

func (a *A) Set(n int) {
    *a = A(n)
}

type B struct {
    A
    b int
}

type C struct {
    *A
    c int
}
```

类型 A 有一个值接收者方法 Value()和一个指针接收者方法 Set(),将 A 以值嵌入的方式嵌入类型 B 中,以地址嵌入的方式嵌入类型 C 中,然后看一下 B、C、*B 和 *C 会继承哪些方法。先用 go tool compile 命令把上述源码编译为 OBJ 文件,然后就可以通过 go tool nm 工具确认了,命令如下:

```
$ go tool compile -p inherit -trimpath="`pwd`=>" gom.go
$ go tool nm gom.o | grep 'T '
    31ea T inherit.(*A).Set
    3b3f T inherit.(*A).Value
    3b95 T inherit.(*B).Set
    3ba6 T inherit.(*B).Value
    3c7a T inherit.(*C).Set
    3c8c T inherit.(*C).Value
    31df T inherit.A.Value
    3c10 T inherit.B.Value
    3cfd T inherit.C.Set
    3d67 T inherit.C.Value
```

通过这个列表就能知道各个自定义类型的方法集中有哪些方法,将以上结果整理为更加直观的表格形式,如表 5-2 所示,还是要注意 T 和 *T 是不同的类型。

值接收者方法始终能够被继承,但只有在能够获得嵌入对象的地址的情况下才能继承指针接收者方法,所以无论是值嵌入还是地址嵌入,*B 和 *C 都能继承 Set()方法。由于嵌入地址的关系,C 也能够继承 Set()方法。

如果一个接口要求实现 Value()和 Set()这两个方法,则上述几种自定义类型中 *A、*B、*C 和 C 都实现了该接口。A 和 B 没有实现 Set()方法,也就是说 A 和 B 的方法集中

没有 Set()方法。通过接口调用 C 的方法时，虽然实际上调用的是 * C 的方法，但语义层面还是 C 实现了这两个方法，只不过接口机制需要指针接收者。

表 5-2 示例程序中各自定义类型包含的方法的情况

自定义类型	有 Value()方法	有 Set()方法
A	√	
* A	√	√
B	√	
* B	√	√
C	√	√
* C	√	√

所以回过头再来看，Go 语言不允许为 T 和 * T 定义同名方法，实际上并不是因为不支持函数重载，前面已经看到了 A.Value()方法和(*A).Value()方法是可以区分的。其根本原因就是编译器要为值接收者方法生成指针接收者包装方法，要保证两者的逻辑一致，所以不允许用户同时实现，用户可能会实现成不同的逻辑。


7min

5.3 类型断言

所谓类型断言，就是运行阶段根据元数据信息，来判断数据是否属于某种具体类型，或者是否实现了某个接口。既然要用到类型元数据，那么源操作数就必须是 interface{}或某个接口类型的变量，也就是说底层是 runtime.eface 或 runtime.iface 类型。

类型断言在语法上有两种不同的形式，第一种就是直接断言为目标类型，这也是最正常的写法，示例代码如下：

```
dest := source.(dest_type)
```

这种形式存在一定风险，如果断言失败，就会造成 panic。第二种形式比较安全，这种形式的代码常被称为 comma ok 风格，因为有个额外的 bool 变量来表明操作是否成功，人们习惯把这个 bool 变量命名为 ok。这种形式的断言无论成败都不会造成 panic，示例代码如下：

```
dest, ok := source.(dest_type)
```

本节就根据源操作数和目标类型的不同，把类型断言分成 4 种情况，结合反编译和 runtime 源码分析，分别探索几种情况的实现原理。

5.3.1 E To 具体类型

E 指的是 runtime.eface，也就是 interface{}类型，而具体类型是相对于抽象类型来讲

的，抽象类型指的是接口，接口通过方法列表对行为进行抽象，所以具体类型指的是除接口以外的内置类型和自定义类型。E To 具体类型的断言就是从容器中把数据取出来。

先来看一看第一种形式，也就是从 interface{}直接断言为某个具体类型，下面把这部分逻辑放到一个单独的函数中，以便于后续分析，代码如下：

```
func normal(a interface{}) int {
    return a.(int)
}
```

用 go tool compile 命令编译包含上述函数的源码文件 e2t.go，会得到一个 OBJ 文件 e2t.o，再用 go tool objdump 命令反编译该文件中的 normal()函数，得到的汇编代码如下：

```
$ go tool compile -p gom -trimpath="`pwd`=>" e2t.go
$ go tool objdump -S -s '^gom.normal$' e2t.o
TEXT gom.normal(SB) gofile..e2t.go
func normal(a interface{}) int {
  0x7d2       65488b0c2528000000    MOVQ GS:0x28, CX
  0x7db       488b8900000000        MOVQ 0(CX), CX    [3:7]R_TLS_LE
  0x7e2       483b6110              CMPQ 0x10(CX), SP
  0x7e6       7651                  JBE 0x839
  0x7e8       4883ec20              SUBQ $0x20, SP
  0x7ec       48896c2418            MOVQ BP, 0x18(SP)
  0x7f1       488d6c2418            LEAQ 0x18(SP), BP
        return a.(int)
  0x7f6       488d0500000000        LEAQ 0(IP), AX    [3:7]R_PCREL:type.int
  0x7fd       488b4c2428            MOVQ 0x28(SP), CX
  0x802       4839c8                CMPQ CX, AX
  0x805       7517                  JNE 0x81e
  0x807       488b442430            MOVQ 0x30(SP), AX
  0x80c       488b00                MOVQ 0(AX), AX
  0x80f       4889442438            MOVQ AX, 0x38(SP)
  0x814       488b6c2418            MOVQ 0x18(SP), BP
  0x819       4883c420              ADDQ $0x20, SP
  0x81d       c3                    RET
  0x81e       48890c24              MOVQ CX, 0(SP)
  0x822       4889442408            MOVQ AX, 0x8(SP)
  0x827       488d0500000000        LEAQ 0(IP), AX    [3:7]R_PCREL:type.interface{}
  0x82e       4889442410            MOVQ AX, 0x10(SP)
  0x833       e800000000            CALL 0x838        [1:5]R_CALL:runtime.panicdottypeE
  0x838       90                    NOPL
func normal(a interface{}) int {
  0x839       e800000000            CALL 0x83e        [1:5]R_CALL:runtime.morestack_noctxt
  0x83e       eb92                  JMP gom.normal(SB)
```

汇编代码还是不太直观，转换成等价的伪代码如下：

```
func normal(a runtime.eface) int {
entry:
    gp := getg()
    if SP <= gp.stackguard0 {
        goto morestack
    }
    if a._type != &type.int {
        runtime.panicdottypeE(a._type, &type.int, &type.interface{})
    }
    return *(*int)(a.data)
morestack:
    runtime.morestack_noctxt()
    goto entry
}
```

编译器插入与栈增长相关的代码已经屡见不鲜了，真正与类型断言相关的代码只有4行。逻辑也很简单，就是判断a._type与int类型的元数据地址是否相等，如果不相等就调用panicdottypeE()函数，如果相等就把a.data作为*int来提取int数值。

再来看一下comma ok风格的断言，代码如下：

```
func commaOk(a interface{}) (n int, ok bool) {
    n, ok = a.(int)
    return
}
```

用相同的命令进行编译和反编译，得到的汇编代码如下：

```
$ go tool compile -p gom -trimpath="`pwd`=>" e2t2.go
$ go tool objdump -S -s '^gom.commaOk$' e2t2.o
TEXT gom.commaOk(SB) gofile..e2t2.go
        n, ok = a.(int)
  0x810       488d0500000000    LEAQ 0(IP), AX          [3:7]R_PCREL:type.int
  0x817       488b4c2408        MOVQ 0x8(SP), CX
  0x81c       4839c8            CMPQ CX, AX
  0x81f       7515              JNE 0x836
  0x821       488b442410        MOVQ 0x10(SP), AX
  0x826       488b00            MOVQ 0(AX), AX
        return
  0x829       4889442418        MOVQ AX, 0x18(SP)
        n, ok = a.(int)
  0x82e       0f94c0            SETE AL
        return
  0x831       88442420          MOVB AL, 0x20(SP)
  0x835       c3                RET
```

```
  0x836       b800000000              MOVL $0x0, AX
        n, ok = a.(int)
  0x83b       ebec                    JMP 0x829
```

因为函数栈帧足够小，并且没有调用任何外部函数，所以编译器无须插入栈增长代码。转换成等价的伪代码也比较精简，代码如下：

```
func commaOk(a runtime.eface) (n int, ok bool) {
    if a._type != &type.int {
        return 0, false
    }
    return *(*int)(a.data), true
}
```

核心逻辑还是判断 a._type 与 int 类型的元数据地址是否相等，如果不相等就返回 int 类型零值和 false，如果相等就把 a.data 作为 *int 来提取 int 数值，然后和 true 一起返回。

综上所述，从 interface{}到具体类型的断言如图 5-17 所示，基本上就是一个指针比较操作加上一个具体类型相关的复制操作，执行时应该还是很高效的。

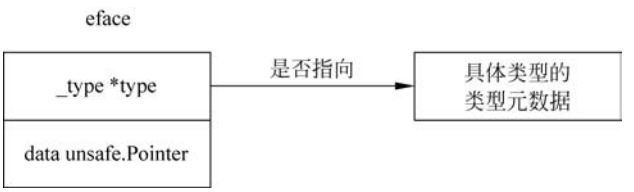

图 5-17 从 interface{}到具体类型的断言

5.3.2 E To I

E 指的还是 runtime.eface，I 指的则是 runtime.iface，E To I 也就是从 interface{}到某个自定义接口类型的断言。断言的目标为接口类型，E 背后的具体类型需要实现接口要求的所有方法，所以涉及具体类型的方法集遍历、动态分配 itab 等操作。5.2.2 节已经分析过 runtime 中用来完成此工作的 getitab()函数，本节继续探索类型断言是如何使用该函数的。

还是先按照一般的代码风格实现一个包含断言逻辑的函数，代码如下：

```
func normal(a interface{}) io.ReadWriter {
    return a.(io.ReadWriter)
}
```

用同样的命令进行编译和反编译，得到的汇编代码如下：

```
$ go tool compile -p gom -trimpath="`pwd`=>" e2i.go
$ go tool objdump -S -s '^gom.normal$' e2i.o
TEXT gom.normal(SB) gofile..e2i.go
func normal(a interface{}) io.ReadWriter {
  0x8b5       65488b0c2528000000    MOVQ GS:0x28, CX
  0x8be       488b8900000000        MOVQ 0(CX), CX  [3:7]R_TLS_LE
  0x8c5       483b6110              CMPQ 0x10(CX), SP
  0x8c9       7650                  JBE 0x91b
  0x8cb       4883ec30              SUBQ $0x30, SP
  0x8cf       48896c2428            MOVQ BP, 0x28(SP)
  0x8d4       488d6c2428            LEAQ 0x28(SP), BP
        return a.(io.ReadWriter)
  0x8d9       488d0500000000        LEAQ 0(IP), AX  [3:7]R_PCREL:type.io.ReadWriter
  0x8e0       48890424              MOVQ AX, 0(SP)
  0x8e4       488b442438            MOVQ 0x38(SP), AX
  0x8e9       4889442408            MOVQ AX, 0x8(SP)
  0x8ee       488b442440            MOVQ 0x40(SP), AX
  0x8f3       4889442410            MOVQ AX, 0x10(SP)
  0x8f8       e800000000            CALL 0x8fd    [1:5]R_CALL:runtime.assertE2I
  0x8fd       488b442418            MOVQ 0x18(SP), AX
  0x902       488b4c2420            MOVQ 0x20(SP), CX
  0x907       4889442448            MOVQ AX, 0x48(SP)
  0x90c       48894c2450            MOVQ CX, 0x50(SP)
  0x911       488b6c2428            MOVQ 0x28(SP), BP
  0x916       4883c430              ADDQ $0x30, SP
  0x91a       c3                    RET
func normal(a interface{}) io.ReadWriter {
  0x91b       e800000000            CALL 0x920    [1:5]R_CALL:runtime.morestack_noctxt
  0x920       eb93                  JMP gom.normal(SB)
```

在保证逻辑不变的前提下，写出等价的 Go 风格伪代码如下：

```
func normal(a runtime.eface) io.ReadWriter {
entry:
    gp := getg()
    if SP <= gp.stackguard0 {
        goto morestack
    }
    return runtime.assertE2I(&type.io.ReadWriter, a)
morestack:
    runtime.morestack_noctxt()
    goto entry
}
```

除去编译器插入的栈增长代码，核心逻辑就是调用了 runtime.assertE2I()函数，摘抄

runtime 包中的函数代码如下：

```
func assertE2I(inter * interfacetype, e eface) (r iface) {
    t := e._type
    if t == nil {
        panic(&TypeAssertionError{nil, nil, &inter.typ, ""})
    }
    r.tab = getitab(inter, t, false)
    r.data = e.data
    return
}
```

函数先校验了 E 的具体类型元数据指针不可为空，没有具体类型的元数据是无法进行断言的，然后通过调用 getitab()函数来得到对应的 itab，data 字段直接复制。注意调用 getitab()函数时最后一个参数为 false，根据之前的源码分析已知这个参数是 canfail。canfail 为 false 时，如果 t 没有实现 inter 要求的所有方法，getitab()函数就会造成 panic。

接下来再看一下 comma ok 风格的断言，代码如下：

```
func commaOk(a interface{}) (i io.ReadWriter, ok bool) {
    i, ok = a.(io.ReadWriter)
    return
}
```

编译再反编译之后，得到的汇编代码如下：

```
$ go tool compile -p gom -trimpath="`pwd`=>" e2i2.go
$ go tool objdump -S -s '^gom.commaOk$' e2i2.o
TEXT gom.commaOk(SB) gofile..e2i2.go
func commaOk(a interface{}) (i io.ReadWriter, ok bool) {
  0x979       65488b0c2528000000    MOVQ GS:0x28, CX
  0x982       488b8900000000        MOVQ 0(CX), CX    [3:7]R_TLS_LE
  0x989       483b6110              CMPQ 0x10(CX), SP
  0x98d       7659                  JBE 0x9e8
  0x98f       4883ec38              SUBQ $0x38, SP
  0x993       48896c2430            MOVQ BP, 0x30(SP)
  0x998       488d6c2430            LEAQ 0x30(SP), BP
        i, ok = a.(io.ReadWriter)
  0x99d       488d0500000000        LEAQ 0(IP), AX    [3:7]R_PCREL:type.io.ReadWriter
  0x9a4       48890424              MOVQ AX, 0(SP)
  0x9a8       488b442440            MOVQ 0x40(SP), AX
  0x9ad       4889442408            MOVQ AX, 0x8(SP)
  0x9b2       488b442448            MOVQ 0x48(SP), AX
  0x9b7       4889442410            MOVQ AX, 0x10(SP)
  0x9bc       e800000000            CALL 0x9c1    [1:5]R_CALL:runtime.assertE2I2
```

```
  0x9c1       488b442418              MOVQ 0x18(SP), AX
  0x9c6       488b4c2420              MOVQ 0x20(SP), CX
  0x9cb       0fb6542428              MOVZX 0x28(SP), DX
        return
  0x9d0       4889442450              MOVQ AX, 0x50(SP)
  0x9d5       48894c2458              MOVQ CX, 0x58(SP)
  0x9da       88542460                MOVB DL, 0x60(SP)
  0x9de       488b6c2430              MOVQ 0x30(SP), BP
  0x9e3       4883c438                ADDQ $0x38, SP
  0x9e7       c3                      RET
func commaOk(a interface{}) (i io.ReadWriter, ok bool) {
  0x9e8       e800000000              CALL 0x9ed    [1:5]R_CALL:runtime.morestack_noctxt
  0x9ed       eb8a                    JMP gom.commaOk(SB)
```

写成等价的 Go 风格伪代码如下：

```
func commaOk(a runtime.eface) (i io.ReadWriter, ok bool) {
entry:
    gp := getg()
    if SP <= gp.stackguard0 {
        goto morestack
    }
    return runtime.assertE2I2(&type.io.ReadWriter, a)
morestack:
    runtime.morestack_noctxt()
    goto entry
}
```

可以看到这次主要通过 runtime.assertE2I2()函数来完成，从 runtime 包中找到该函数的源代码如下：

```
func assertE2I2(inter *interfacetype, e eface) (r iface, b bool) {
    t := e._type
    if t == nil {
        return
    }
    tab := getitab(inter, t, true)
    if tab == nil {
        return
    }
    r.tab = tab
    r.data = e.data
    b = true
    return
}
```

与之前不同的是，可以通过第 2 个返回值来表示操作的成功与否，所以不用再造成 panic。如果 E 的具体类型指针为空，则直接返回 false。调用 getitab()函数时也把 canfail 设置为 true，并且需要检测返回的 tab 是否为 nil，以此来判断是否成功。

综上所述，E To I 形式的类型断言，主要通过 runtime 中的 assertE2I()和 assertE2I2()这两个函数实现，底层的主要任务如图 5-18 所示，都是通过 getitab()函数完成的方法集遍历及 itab 分配和初始化。因为 getitab()函数中用到了全局的 itab 缓存，所以性能方面应该也是很高效的。

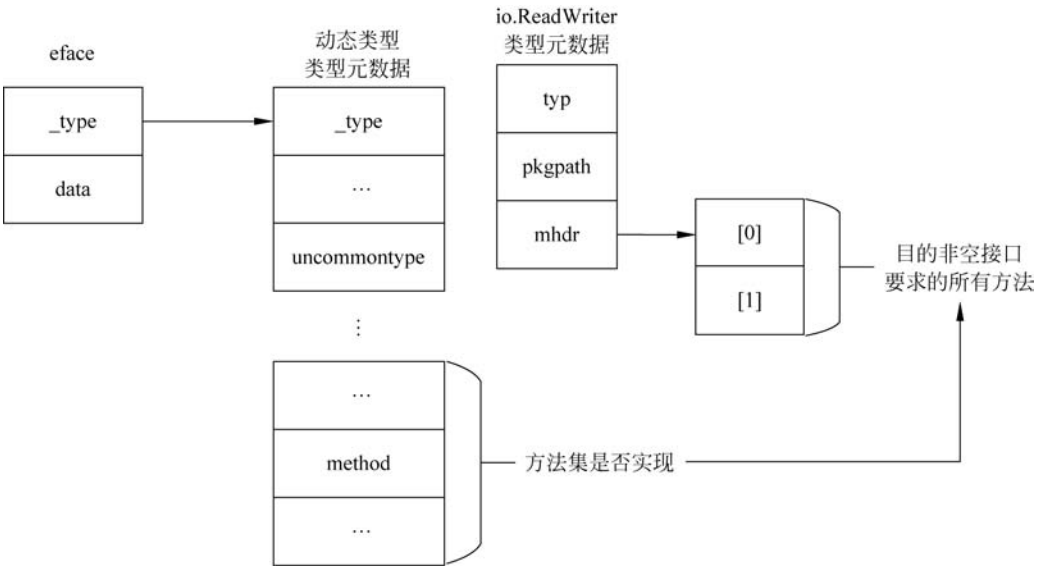

图 5-18 从 interface{}到非空接口的类型断言

5.3.3 I To 具体类型

5.3.1 节和 5.3.2 节主要探索了源类型为 interface{}的类型断言，目标分为具体类型和接口类型两种情况。接下来看一下源类型为接口类型的类型断言，本节首先分析目标为具体类型的断言实现，也就是从 runtime.iface 转换为某种具体类型。

还是先按照一般的写法把类型断言逻辑放到一个单独的函数中，代码如下：

```
func normal(i io.ReadWriter) *os.File {
    return i.(*os.File)
}
```

然后使用 go tool 命令编译和反编译，得到的汇编代码如下：

```
$ go tool compile -p gom -trimpath="`pwd`=>" i2t.go
$ go tool objdump -S -s '^gom.normal$' i2t.o
```

```
TEXT gom.normal(SB) gofile..i2t.go
func normal(i io.ReadWriter) *os.File {
  0x10bd      65488b0c2528000000    MOVQ GS:0x28, CX
  0x10c6      488b8900000000        MOVQ 0(CX), CX    [3:7]R_TLS_LE
  0x10cd      483b6110              CMPQ 0x10(CX), SP
  0x10d1      7655                  JBE 0x1128
  0x10d3      4883ec20              SUBQ $0x20, SP
  0x10d7      48896c2418            MOVQ BP, 0x18(SP)
  0x10dc      488d6c2418            LEAQ 0x18(SP), BP
        return i.(*os.File)
  0x10e1      488d0500000000        LEAQ 0(IP), AX
[3:7]R_PCREL:go.itab.*os.File,io.ReadWriter
  0x10e8      488b4c2428            MOVQ 0x28(SP), CX
  0x10ed      4839c8                CMPQ CX, AX
  0x10f0      7514                  JNE 0x1106
  0x10f2      488b442430            MOVQ 0x30(SP), AX
  0x10f7      4889442438            MOVQ AX, 0x38(SP)
  0x10fc      488b6c2418            MOVQ 0x18(SP), BP
  0x1101      4883c420              ADDQ $0x20, SP
  0x1105      c3                    RET
  0x1106      48890c24              MOVQ CX, 0(SP)
  0x110a      488d0500000000        LEAQ 0(IP), AX  [3:7]R_PCREL:type.*os.File
  0x1111      4889442408            MOVQ AX, 0x8(SP)
  0x1116      488d0500000000        LEAQ 0(IP), AX  [3:7]R_PCREL:type.io.ReadWriter
  0x111d      4889442410            MOVQ AX, 0x10(SP)
  0x1122      e800000000            CALL 0x1127     [1:5]R_CALL:runtime.panicdottypeI
  0x1127      90                    NOPL
func normal(i io.ReadWriter) *os.File {
  0x1128      e800000000            CALL 0x112d     [1:5]R_CALL:runtime.morestack_noctxt
  0x112d      eb8e                  JMP gom.normal(SB)
```

与之前从 interface{}断言有些不同，为了更加直观，写出等价的 Go 风格伪代码如下：

```
func normal(i runtime.iface) *os.File {
entry:
    gp := getg()
    if SP <= gp.stackguard0 {
        goto morestack
    }
    if i.tab != &go.itab.*os.File,io.ReadWriter {
        runtime.panicdottypeI(i.tab, &type.*os.File, &type.io.ReadWriter)
    }
    return (*os.File)(i.data)
morestack:
    runtime.morestack_noctxt()
```

```
    goto entry
}
```

其中的 go.itab.*os.File,io.ReadWriter 指的就是全局 itab 缓存中与 *os.File 和 io.ReadWriter 这一对类型对应的 itab。这个 itab 是在编译阶段就被编译器生成的,所以代码中可以直接链接到它的地址。这个断言的核心逻辑就是比较 iface 中 tab 字段的地址是否与目标 itab 地址相等。如果不相等就调用 panicdottypeI,如果相等就把 iface 的 data 字段返回。注意这里因为 *os.File 是指针类型,所以不涉及自动拆箱,也就没有与具体类型相关的复制操作,如果具体类型为值类型就不然了。

实际反编译之前,笔者曾经以为会比较 i.tab._type 和 &type.*os.File,但是 Go 语言的实际实现更为直接高效,也省去了对 i.tab 的非空校验。

再来看一下 comma ok 风格的断言,代码如下:

```
func commaOk(i io.ReadWriter) (f *os.File, ok bool) {
    f, ok = i.(*os.File)
    return
}
```

先编译成 OBJ,再反编译,得到的汇编代码如下:

```
$ go tool compile -p gom -trimpath="`pwd`=>" i2t2.go
$ go tool objdump -S -s '^gom.commaOk$' i2t2.o
TEXT gom.commaOk(SB) gofile..i2t2.go
        f, ok = i.(*os.File)
  0x10a8        488d0500000000      LEAQ 0(IP), AX      [3:7]R_PCREL:go.itab.*os.File,
io.ReadWriter
  0x10af        488b4c2408          MOVQ 0x8(SP), CX
  0x10b4        4839c8              CMPQ CX, AX
  0x10b7        7512                JNE 0x10cb
  0x10b9        488b442410          MOVQ 0x10(SP), AX
        return
  0x10be        4889442418          MOVQ AX, 0x18(SP)
        f, ok = i.(*os.File)
  0x10c3        0f94c0              SETE AL
        return
  0x10c6        88442420            MOVB AL, 0x20(SP)
  0x10ca        c3                  RET
  0x10cb        b800000000          MOVL $0x0, AX
        f, ok = i.(*os.File)
  0x10d0        ebec                JMP 0x10be
```

因为不需要调用 panicdottypeI()函数的关系,所以编译器可以省略掉与栈增长相关的代码。核心逻辑还是比较 itab 的地址,写出等价的 Go 风格伪代码如下:

```
func commaOk(i runtime.iface) (f *os.File, ok bool) {
    if i.tab != &go.itab.*os.File,io.ReadWriter {
        return nil, false
    }
    return (*os.File)(i.data), true
}
```

与一般风格的类型断言也没有太大的不同，不同点就是通过返回值为 false 表示断言失败，代替了调用 panicdottypeI()函数。

综上所述，I To 具体类型的断言与 E To 具体类型的断言在实现上极其相似，核心逻辑如图 5-19 所示，都是一个指针的相等判断。

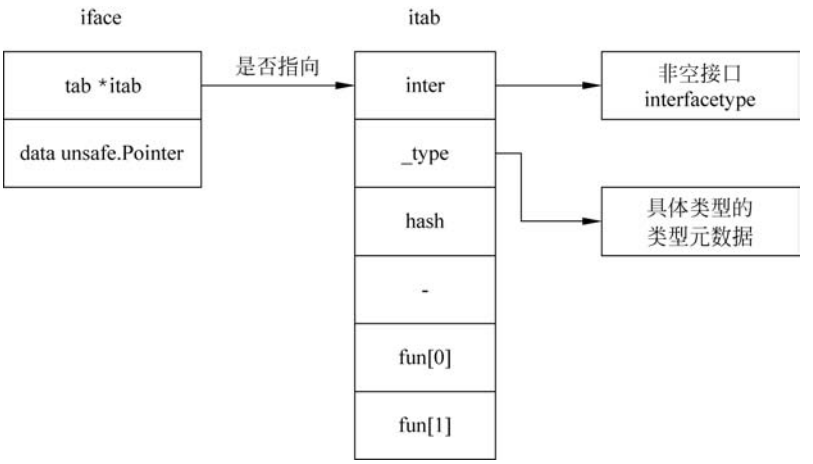

图 5-19 从非空接口到具体类型的类型断言

是否涉及自动拆箱，要视具体类型为值类型还是指针类型而定。值类型要进行拆箱操作，也就是从 data 地址处把值复制出来，指针类型则无须拆箱，直接返回 data 即可，无论源类型为 E 或 I，其实都是一样的。

5.3.4 I To I

本节探索类型断言的最后一种场景，从一种接口类型到另一种接口类型，因为接口类型对应着 runtime.iface，所以简称为 I To I。断言的源接口和目标接口应该有着不同的类型，而实际影响断言的就是目标接口有着怎样的方法列表，底层应该还是基于 getitab()函数。

按照一般的类型断言风格，准备一个示例函数，代码如下：

```
func normal(rw io.ReadWriter) io.Reader {
    return rw.(io.Reader)
}
```

还是经过编译和反编译，得到的汇编代码如下：

```
$ go tool compile -p gom -trimpath="`pwd`=>" i2i.go
$ go tool objdump -S -s '^gom.normal$' i2i.o
TEXT gom.normal(SB) gofile..i2i.go
func normal(rw io.ReadWriter) io.Reader {
  0x64c       65488b0c2528000000      MOVQ GS:0x28, CX
  0x655       488b8900000000          MOVQ 0(CX), CX    [3:7]R_TLS_LE
  0x65c       483b6110                CMPQ 0x10(CX), SP
  0x660       7650                    JBE 0x6b2
  0x662       4883ec30                SUBQ $0x30, SP
  0x666       48896c2428              MOVQ BP, 0x28(SP)
  0x66b       488d6c2428              LEAQ 0x28(SP), BP
        return rw.(io.Reader)
  0x670       488d0500000000          LEAQ 0(IP), AX    [3:7]R_PCREL:type.io.Reader
  0x677       48890424                MOVQ AX, 0(SP)
  0x67b       488b442438              MOVQ 0x38(SP), AX
  0x680       4889442408              MOVQ AX, 0x8(SP)
  0x685       488b442440              MOVQ 0x40(SP), AX
  0x68a       4889442410              MOVQ AX, 0x10(SP)
  0x68f       e800000000              CALL 0x694    [1:5]R_CALL:runtime.assertI2I
  0x694       488b442418              MOVQ 0x18(SP), AX
  0x699       488b4c2420              MOVQ 0x20(SP), CX
  0x69e       4889442448              MOVQ AX, 0x48(SP)
  0x6a3       48894c2450              MOVQ CX, 0x50(SP)
  0x6a8       488b6c2428              MOVQ 0x28(SP), BP
  0x6ad       4883c430                ADDQ $0x30, SP
  0x6b1       c3                      RET
func normal(rw io.ReadWriter) io.Reader {
  0x6b2       e800000000              CALL 0x6b7    [1:5]R_CALL:runtime.morestack_noctxt
  0x6b7       eb93                    JMP gom.normal(SB)
```

写出逻辑等价的 Go 风格伪代码如下：

```
func normal(i runtime.iface) io.Reader {
entry:
    gp := getg()
    if SP <= gp.stackguard0 {
        goto morestack
    }
    return runtime.assertI2I(&type.io.Reader, i)
morestack:
    runtime.morestack_noctxt()
    goto entry
}
```

实际上就是调用了 runtime.assertI2I()函数,该函数的源代码如下:

```
func assertI2I(inter * interfacetype, i iface) (r iface) {
    tab := i.tab
    if tab == nil {
        panic(&TypeAssertionError{nil, nil, &inter.typ, ""})
    }
    if tab.inter == inter {
        r.tab = tab
        r.data = i.data
        return
    }
    r.tab = getitab(inter, tab._type, false)
    r.data = i.data
    return
}
```

先校验 i.tab 不为 nil,否则就意味着没有类型元数据,类型断言也就无从谈起,然后检测 i.tab.inter 是否等于 inter,相等就意味着源接口和目标接口类型相同,直接复制就可以了。最后才调用 getitab()函数,根据 inter 和 i.tab._type 获取对应的 itab。canfail 参数为 false,所以如果 getitab()函数失败就会造成 panic。

再来看一下 comma ok 风格的断言,准备的函数代码如下:

```
func commaOk(rw io.ReadWriter) (r io.Reader, ok bool) {
    r, ok = rw.(io.Reader)
    return
}
```

将上述代码先编译为 OBJ,再进行反编译,得到的汇编代码如下:

```
$ go tool compile -p gom -trimpath="`pwd`=>" i2i2.go
$ go tool objdump -S -s '^gom.commaOk$' i2i2.o
TEXT gom.commaOk(SB) gofile..i2i2.go
func commaOk(rw io.ReadWriter) (r io.Reader, ok bool) {
  0x710       65488b0c2528000000    MOVQ GS:0x28, CX
  0x719       488b8900000000        MOVQ 0(CX), CX    [3:7]R_TLS_LE
  0x720       483b6110              CMPQ 0x10(CX), SP
  0x724       7659                  JBE 0x77f
  0x726       4883ec38              SUBQ $0x38, SP
  0x72a       48896c2430            MOVQ BP, 0x30(SP)
  0x72f       488d6c2430            LEAQ 0x30(SP), BP
        r, ok = rw.(io.Reader)
  0x734       488d0500000000        LEAQ 0(IP), AX    [3:7]R_PCREL:type.io.Reader
  0x73b       48890424              MOVQ AX, 0(SP)
```

```
  0x73f       488b442440              MOVQ 0x40(SP), AX
  0x744       4889442408              MOVQ AX, 0x8(SP)
  0x749       488b442448              MOVQ 0x48(SP), AX
  0x74e       4889442410              MOVQ AX, 0x10(SP)
  0x753       e800000000              CALL 0x758    [1:5]R_CALL:runtime.assertI2I2
  0x758       488b442418              MOVQ 0x18(SP), AX
  0x75d       488b4c2420              MOVQ 0x20(SP), CX
  0x762       0fb6542428              MOVZX 0x28(SP), DX
        return
  0x767       4889442450              MOVQ AX, 0x50(SP)
  0x76c       48894c2458              MOVQ CX, 0x58(SP)
  0x771       88542460                MOVB DL, 0x60(SP)
  0x775       488b6c2430              MOVQ 0x30(SP), BP
  0x77a       4883c438                ADDQ $0x38, SP
  0x77e       c3                      RET
func commaOk(rw io.ReadWriter) (r io.Reader, ok bool) {
  0x77f       e800000000              CALL 0x784    [1:5]R_CALL:runtime.morestack_noctxt
  0x784       eb8a                    JMP gom.commaOk(SB)
```

等价的 Go 风格伪代码如下：

```
func commaOk(rw io.ReadWriter) (r io.Reader, ok bool) {
entry:
    gp := getg()
    if SP <= gp.stackguard0 {
        goto morestack
    }
    return runtime.assertI2I2(&type.io.Reader, i)
morestack:
    runtime.morestack_noctxt()
    goto entry
}
```

这次是通过 runtime.assertI2I2()函数实现的，该函数的代码如下：

```
func assertI2I2(inter *interfacetype, i iface) (r iface, b bool) {
    tab := i.tab
    if tab == nil {
        return
    }
    if tab.inter != inter {
        tab = getitab(inter, tab._type, true)
        if tab == nil {
            return
        }
```

```
        }
        r.tab = tab
        r.data = i.data
        b = true
        return
    }
```

如果 i. tab 为 nil，则直接返回 false。只有在 i. tab. inter 与 inter 不相等时才调用 getitab()函数，而且 canfail 为 true，如果 getitab()函数失败，则不会造成 panic，而是返回 nil。

综上所述，I To I 的类型断言，如图 5-20 所示，实际上是通过 runtime. assertI2I()函数和 runtime. assertI2I2()函数实现的，底层也都是基于 getitab()函数实现的。

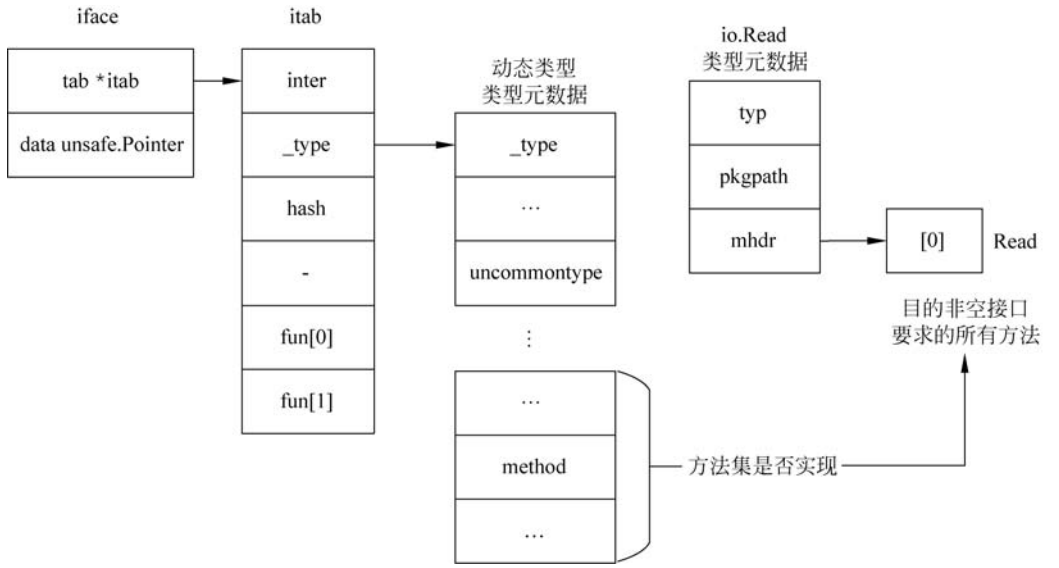

图 5-20 从非空接口到非空接口的类型断言

5.4 反射


5min

所谓反射，实际上就是围绕类型元数据展开的编程。程序的源码中包含最全面的类型信息，在 C/C++一类的编程语言中，源码中的类型信息主要供编译阶段使用，这些类型信息定义了数据对象的内存布局、所支持的操作等，编译器依赖这些信息来生成相应的机器指令。经过编译之后，上层语言中那些直观、抽象的代码都被转换成了具体的机器指令，指令中操作的都是不同宽度的整型、浮点数这类很底层的数据类型，那些上层语言中的抽象数据类型也不复存在了。

而对于 Go、Java 这类支持反射的编程语言，经过编译阶段以后，代码中定义的各种类型信息会被保留下来。编译器会使用特定的数据结构来装载类型信息，并把它们写入生成的 OBJ 文件中，这些信息最终会被链接器存放到可执行文件相应的节区，供运行阶段检索使用。在 Go 语言中用来装载类型信息的数据结构就是 5.1.2 节介绍过的 runtime._type，也就是我们俗称的类型元数据。在介绍动态派发和类型断言时，已经见识过类型元数据的重要性，本节就更系统地研究 Go 语言的类型系统，以及在此之上建立的强大的反射机制。

4min

5.4.1 类型系统

Go 语言一共提供了 26 种类型种类：一个布尔型，包含 uintptr 在内一共 11 种整型，两种浮点类型，两种复数类型，一个字符串类型，指针、数组、切片、map 和 struct 共 5 种常用复合类型，以及 chan、func、interface 和 unsafe.Pointer 这 4 种特殊类型。这 26 种类型是 Go 语言整个类型系统的基础，任何更复杂的类型都由这些类型组合而来，即使用户自定义的类型有着各种各样的名称，它们的种类也不会超出这 26 种的范畴。

至此，我们已经知道类型元数据是用 runtime._type 结构表示的，那么这些数据是如何组织起来的，以及运行阶段又是如何解析的呢？带着这个问题，下面就深入 runtime 的源码中去找答案。

1. 类型信息的萃取

提到反射和类型，很自然地就会想起 reflect 包中用于获取类型信息的 TypeOf()函数，该函数有一个 interface{}类型的参数，可以接受传入任意类型。函数的返回值类型是 reflect.Type，这是个接口类型，提供了一系列方法来从类型元数据中提取信息。TypeOf()函数所做的事情如图 5-21 所示，就是找到传入参数的类型元数据，并以 reflect.Type 形式返回。

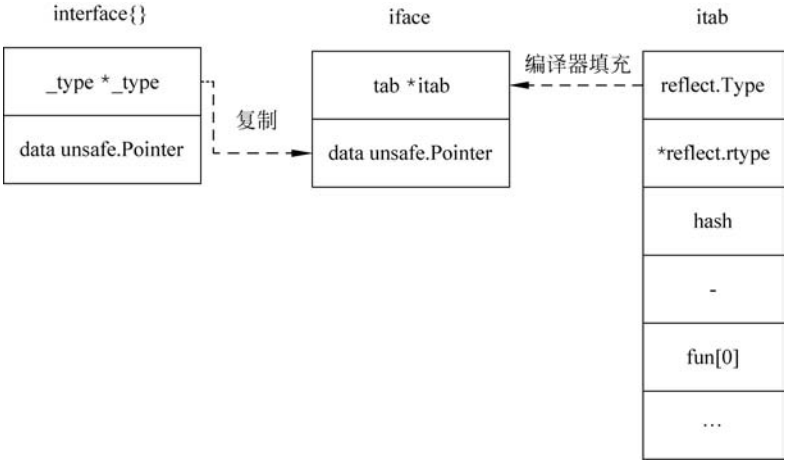

图 5-21 由一个 * _type 和一个 * itab 组建一个 iface

TypeOf()函数的代码如下：

```
func TypeOf(i interface{}) Type {
    eface := *(*emptyInterface)(unsafe.Pointer(&i))
    return toType(eface.typ)
}
```

第2行代码相当于把传入的参数i强制转换成了emptyInterface类型，emptyInterface类型和5.1节介绍过的eface类型在内存布局上等价，emptyInterface类型定义的代码如下：

```
type emptyInterface struct {
    typ *rtype
    word unsafe.Pointer
}
```

其中的rtype类型与runtime._type类型在内存布局方面也是等价的，只不过因为无法使用其他包中未导出的类型定义，所以需要在reflect包中重新定义一下。代码中的eface.typ实际上就是从interface{}变量中提取出的类型元数据地址，再来看一下toType()函数，代码如下：

```
func toType(t *rtype) Type {
    if t == nil {
        return nil
    }
    return t
}
```

先判断了一下传入的rtype指针是否为nil，如果不为nil就把它作为Type类型返回，否则返回nil。从这里可以知道*rtype类型肯定实现了Type接口，之所以要加上这个nil判断，需要考虑到Go的接口类型是个双指针结构，一个指向itab，另一个指向实际的数据对象。如图5-22所示，只有在两个指针都为nil的时候，接口变量才等于nil。

用一段更直观的代码加以说明，代码如下：

```
//第5章/code_5_12.go
var rw io.ReadWriter
if rw == nil {
    println(1)
}
var f *os.File
rw = f
if rw == nil {
    println(2)
}
```

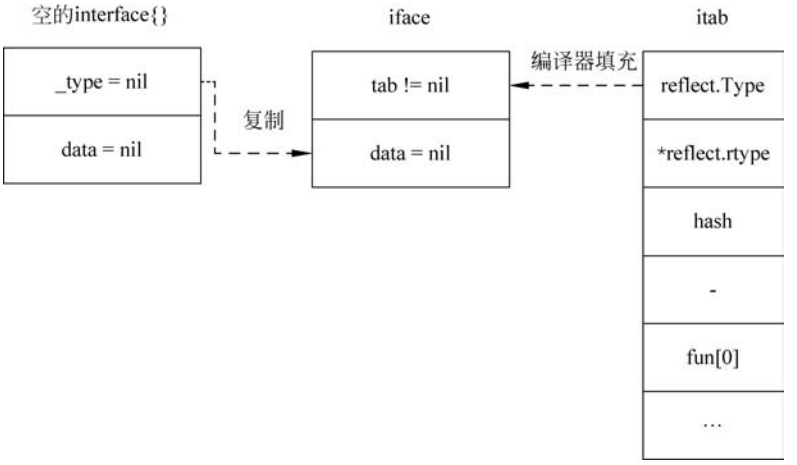

图 5-22 萃取前判断非空

在上述代码中第 1 个 if 处判断结果为真，所以会打印出 1。第 2 个 if 处 rw 不再为 nil，所以不会打印 2。这里需要注意一下，f 本身为 nil，赋值给 rw 之后却不再为 nil，这是因为接口的双指针结构，其中数据指针为 nil，itab 指针不为空。也就是说 nil 指针也是有类型的，所以在赋值给 interface{}和一般的非空接口变量时要格外注意。toType()函数中前置的 nil 检测就是为了避免返回一个 itab 指针不为 nil，而数据指针为 nil 的 Type 变量，使上层代码无法通过 nil 检测区分返回值是否有效，由此带来诸多不便和隐患。

综上所述，TypeOf()函数所做的事情就是从 interface{}中提取出类型元数据地址，然后在地址不为 nil 的时候将其作为 Type 类型返回。并没有太神奇的逻辑，而 interface{}中的类型元数据地址是从哪里来的呢？当然是在编译阶段由编译器赋值的，实际的地址可能是由链接器填写的，也就是说源头还是要追溯到最初的源码中。

2. 类型系统的初始化

迄今为止，见过的所有基于类型元数据的特性都少不了 interface 的影子，通过反射实现类型信息的萃取也要依赖于 interface 参数，然而对于 interface{}和非空接口，其中用到的类型元数据，论及源头都是在编译阶段由编译器赋值的。这样一来，整个类型系统给人的感觉就像是一个 KV 存储，只能在获得某个 key 的前提下去查询对应的 value，有没有一个地方能够遍历所有的 key 呢？下面就带着这个问题去研究 runtime 的源码。

通过 buildmode=plugin 可以把 Go 项目构建成一个动态链接库，后续以插件的形式被程序的主模块按需加载，这样一来运行阶段就需要加载多个二进制模块。由于每个模块中都有自己的一组类型元数据，所以就会出现类型信息不一致的问题，像类型断言这样的特性，底层通过比较元数据地址实现，也就无法正常工作了。保证类型系统中的类型唯一性至关重要，因此 Go 语言的 runtime 会在类型系统的初始化阶段进行去重操作，如图 5-23 所示。

下面从源码层面看一下具体的实现，用来初始化类型系统的就是 runtime.typelinksinit()函数，代码如下：

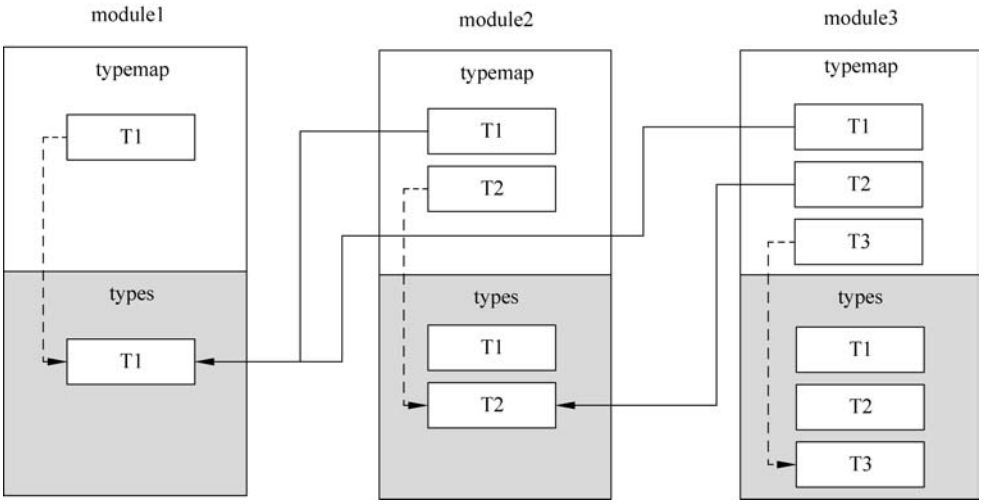

图 5-23 类型系统初始化利用 typemap 去重

```
func typelinksinit() {
    if firstmoduledata.next == nil {
        return
    }
    typehash := make(map[uint32][]*_type, len(firstmoduledata.typelinks))

    modules := activeModules()
    prev := modules[0]
    for _, md := range modules[1:] {
        //把前一个模块中的各种类型收集到 typehash 中
    collect:
        for _, tl := range prev.typelinks {
            var t *_type
            if prev.typemap == nil {
                t = (*_type)(unsafe.Pointer(prev.types + uintptr(tl)))
            } else {
                t = prev.typemap[typeOff(tl)]
            }
            //已经有的就不重复添加了
            tlist := typehash[t.hash]
            for _, tcur := range tlist {
                if tcur == t {
                    continue collect
                }
            }
            typehash[t.hash] = append(tlist, t)
        }
```

```
            if md.typemap == nil {
                //如果当前模块 typelinks 中的某种类型与某个前驱模块中的某类型一致
                //通过当前模块的 typemap 将其映射到前驱模块中的对应类型
                tm := make(map[typeOff] * _type, len(md.typelinks))
                pinnedTypemaps = append(pinnedTypemaps, tm)
                md.typemap = tm
                for _, tl := range md.typelinks {
                    t := ( * _type)(unsafe.Pointer(md.types + uintptr(tl)))
                    for _, candidate := range typehash[t.hash] {
                        seen := map[_typePair]struct{}{}
                        if typesEqual(t, candidate, seen) {
                            t = candidate
                            break
                        }
                    }
                    md.typemap[typeOff(tl)] = t
                }
            }

            prev = md
        }
    }
```

在类型系统内部，元数据间通过 typeOff 互相引用，typeOff 实际上就是个 int32。类型元数据在二进制文件中是存放在一起的，单独占据了一段空间，moduledata 结构的 types 字段和 etypes 字段就是这段空间的起始地址和结束地址。typeOff 表示的就是目标类型的元数据距离起始地址 types 的偏移。梳理一下这个函数的大致逻辑：

(1) 分配了一个 map[uint32][] * _type 类型的变量 typehash，用来收集所有模块中的类型信息，用类型的 hash 作为 map 的 key，收集的是类型元数据_type 结构的地址，把 hash 相同的类型的地址放到同一个 slice 中。

(2) 通过 activeModules()函数得到当前活动模块的列表，也就是所有能够正常使用的 Go 二进制模块，然后从第 2 个模块开始向后遍历。

(3) 每次循环中通过前一个模块的 typelinks 字段，收集模块内的类型信息，将 typehash 中尚未包含的类型添加进去，注意是收集前一个模块的类型信息。这样一来，typehash 中包含的类型信息都是该类型在整个模块列表中首次出现时的那个地址。假如按照 A、B、C 的顺序遍历模块列表，而类型 T 在 B 和 C 中都出现过，typehash 中只会包含 B 模块中 T 的地址。

(4) 如果当前模块的 typemap 为 nil，就分配一个新的 map 并填充数据。遍历当前模块的 typelinks，对于其中所有的类型，先去 typehash 中查找，优先使用 typehash 中的类型地址，typehash 中没有的类型才使用当前模块自身包含的地址，把地址添加到 typemap 中。

pinnedTypemaps 主要是避免 GC 回收掉 typemap，因为模块列表对于 GC 不可见。

这样当整个循环执行完成后，所有模块中的 typemap 中的任何一种类型都是该类型在整个模块列表中第一次出现时的地址，也就实现了类型信息的唯一化，而每个模块的 typelinks 字段就相当于遍历该模块所有类型的入口，虽然并不能从这里找到所有类型信息（有些闭包的类型信息就不会包含）。后续通过 typeOff 引用类型元数据时，会先从 typemap 中查找，如果找不到才会把当前模块的 types 加上 typeOff 作为结果返回，5.4.2 节会更详细地分析讲解。

经过 typelinksinit 之后，用于反射的类型元数据实现了唯一化，跨多个模块的 reflect 就不会出现不一致现象了，但是回过头来继续看一看 5.3.1 节的类型断言的实现原理，底层直接比较类型元数据的地址，不会用到模块的 typemap 字段，所以上述唯一化操作应该无法解决这类问题。

类型断言所用到的元数据地址是由编译器直接编码在指令中的，下面先来研究一下编译器是如何确定类型元数据地址的，代码如下：

```
func IsBool(a interface{}) bool {
    _, ok := a.(bool)
    return ok
}
```

用 compile 命令将上述代码编译成 OBJ 文件，然后进行反编译，得到的汇编代码如下：

```
$ go tool compile -p gom -o assert.o assert.go
$ go tool objdump -S -s 'IsBool' assert.o
TEXT gom.IsBool(SB) gofile../home/kylin/go/src/fengyoulin.com/gom/assert.go
        _, ok := a.(bool)
  0x422       488b442408          MOVQ 0x8(SP), AX          //第 1 条指令
  0x427       488d0d00000000      LEAQ 0(IP), CX
                                  [3:7]R_PCREL:type.bool    //第 2 条指令
  0x42e       4839c8              CMPQ CX, AX
        return ok
  0x431       0f94442418          SETE 0x18(SP)
  0x436       c3                  RET
```

第 2 条汇编指令 LEAQ 用于获取 bool 类型元数据的地址，第 1 个操作数 0(IP)中的 0 是个偏移量，编译阶段只预留了 4 字节的空间，所以在 OBJ 文件中是 0，等到链接器填写了实际的偏移量后可执行文件中就会有值了。LEAQ offset(IP)，CX 的含义就是把当前指令指针 IP 的值加上 offset，把结果存入 CX 寄存器中。这种计算方式是以当前指令位置为基址，然后加上 32 位的偏移来得到目标地址。32 位偏移能够覆盖－2GB～2GB 的偏移范围，多用于单个二进制文件内部的寻址，因为单个二进制文件的大小一般不会超过 2GB。

对于模块间的地址引用，这种相对地址的计算方式就不能很好地支持了。因为64位地址空间中两个模块间的距离可能会超过2GB，所以需要直接使用64位宽度的地址。还是使用IsBool()函数，这次编译的时候加上一个dynlink参数，实际上在以plugin方式构建项目时工具链会自动添加这个编译参数。再反编译得到的OBJ文件，汇编代码如下：

```
$ go tool compile -dynlink -p gom -o assert.o assert.go
$ go tool objdump -S -s 'IsBool' assert.o
TEXT gom.IsBool(SB) gofile../home/kylin/go/src/fengyoulin.com/gom/assert.go
        _, ok := a.(bool)
  0x44d       488b442408          MOVQ 0x8(SP), AX
  0x452       488b0d00000000      MOVQ 0(IP), CX    [3:7]R_GOTPCREL:type.bool
  0x459       4839c8              CMPQ CX, AX
        return ok
  0x45c       0f94442418          SETE 0x18(SP)
  0x461       c3                  RET
```

唯一的不同就是原来的LEAQ变成了MOVQ，含义也发生了很大变化，LEAQ与MOVQ的区别如图5-24所示。LEAQ直接把当前指令地址加上偏移用作元数据地址，而MOVQ从当前指令地址加上偏移处取出一个64位整型，用作类型元数据的地址。也就是MOVQ不直接计算元数据地址，而是又多了一层中转，也就是又多了一层灵活性。

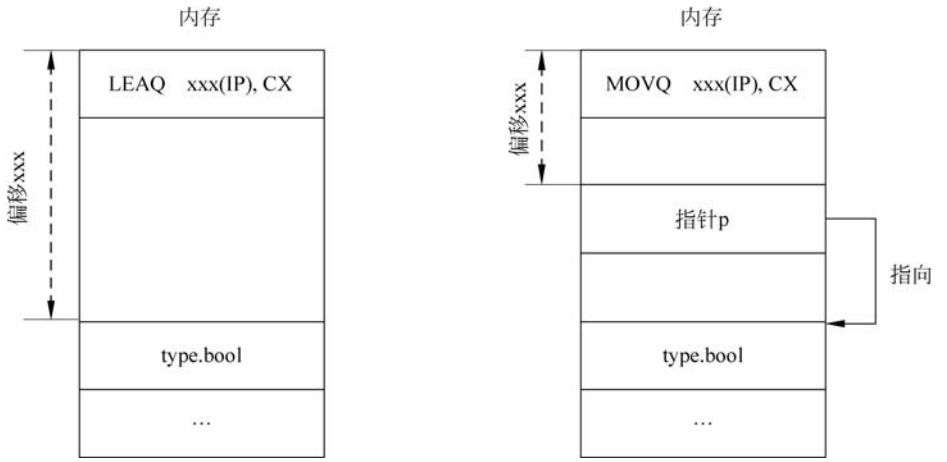

图5-24 LEAQ与MOVQ的区别

进一步分析会发现，MOVQ读取地址的地方是ELF文件中一个叫.got的节区，.got节中有一个全局偏移表(Global Offset Table)，表中的一系列重定位项会在ELF文件被加载的时候由操作系统的动态链接器完成赋值。像类型断言这种，代码中直接使用元数据地

址的场景，其中的类型唯一性问题在二进制模块加载的时候就被动态链接器处理掉了，如图 5-25 所示。

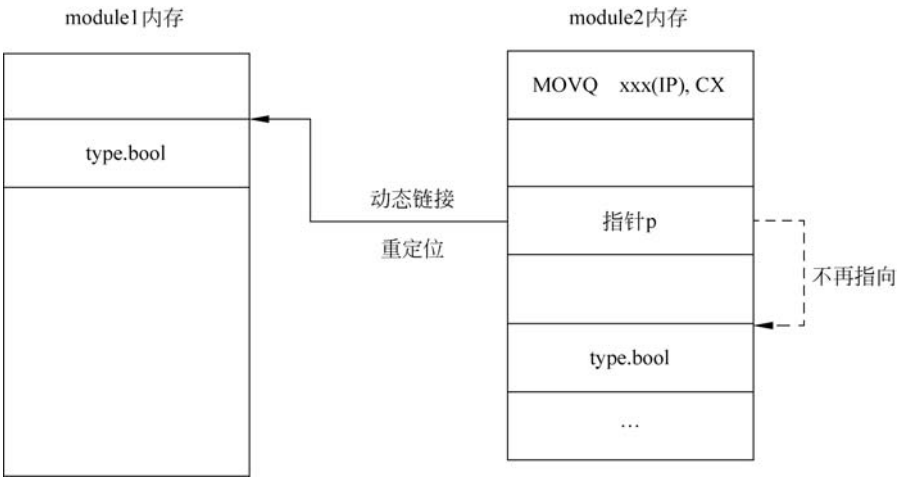

图 5-25 地址被动态链接重定位直接使用类型元数据

讲解了这么多，都是通过读源码和反编译的方式来分析的，还是需要有个实例来运行验证一下。下面就基于 Go 的 plugin 机制来实践一下，实验环境是运行在 amd64 架构上的 Linux 系统。

首先创建第 1 个 mod，这个模块只定义了一个 User 类型，下面来看各个文件的源码。

（1）go.mod 文件的代码如下：

```
//第 5 章/mod1/go.mod
module fengyoulin.com/mod1

go 1.14
```

（2）user.go 文件的代码如下：

```
//第 5 章/mod1/user.go
package mod1

type User struct {
    ID int
    Nick string
}
```

然后是第 2 个 mod，这个模块按照 plugin 的形式，实现了一个 UserFactory。

（1）go.mod 文件的代码如下：

```
//第 5 章/mod2/go.mod
module fengyoulin.com/mod2

go 1.14

require fengyoulin.com/mod1 v0.0.0

replace fengyoulin.com/mod1 => /home/kylin/go/src/fengyoulin.com/mod1
```

（2）factory.go 文件的代码如下：

```
//第 5 章/mod2/factory.go
package main

import "fengyoulin.com/mod1"

type uf struct{}

func (*uf) NewUser(id int, nick string) interface{} {
    return &mod1.User {
        ID: id,
        Nick: nick,
    }
}

var UserFactory uf
```

接下来是第 3 个 mod，这个模块也是一个 plugin，实现了一个 UserChecker。

（1）go.mod 文件的代码如下：

```
//第 5 章/mod3/go.mod
module fengyoulin.com/mod3

go 1.14

require fengyoulin.com/mod1 v0.0.0

replace fengyoulin.com/mod1 => /home/kylin/go/src/fengyoulin.com/mod1
```

（2）checker.go 文件的代码如下：

```
//第 5 章/mod3/checker.go
package main
```

```
import "fengyoulin.com/mod1"

type uc struct{}

func ( * uc) IsUser(a interface{}) bool {
    _, ok : = a.( * mod1.User)
    return ok
}

var UserChecker uc
```

第 4 个模块，也是最后一个模块，此模块是用来加载并调用前面两个 plugin 的主程序。

(1) go.mod 文件的代码如下：

```
//第 5 章/mod4/go.mod
module fengyoulin.com/mod4

go 1.14
```

(2) main.go 文件的代码如下：

```
//第 5 章/mod4/main.go
package main

import (
    "log"
    "plugin"
    "reflect"
)

type UserFactory interface {
    NewUser(id int, nick string) interface{}
}

type UserChecker interface {
    IsUser(a interface{}) bool
}

func factory() UserFactory {
    p, err : = plugin.Open("./mod2.so")
    if err != nil {
        log.Fatalln(err)
    }
    a, err : = p.Lookup("UserFactory")
```

```
    if err != nil {
        log.Fatalln(err)
    }
    uf, ok := a.(UserFactory)
    if !ok {
        log.Fatalln("not a UserFactory")
    }
    return uf
}

func checker() UserChecker {
    p, err := plugin.Open("./mod3.so")
    if err != nil {
        log.Fatalln(err)
    }
    a, err := p.Lookup("UserChecker")
    if err != nil {
        log.Fatalln(err)
    }
    uc, ok := a.(UserChecker)
    if !ok {
        log.Fatalln("not a UserChecker")
    }
    return uc
}

func main() {
    uf := factory()
    uc := checker()
    u := uf.NewUser(1, "Jack")
    if !uc.IsUser(u) {
        log.Println("not a User")
    }
    t := reflect.TypeOf(u)
    println(u, t.String())
    select{}
}
```

最后，以 plugin 模式构建 mod2 和 mod3，会得到两个 so 库，命令如下：

```
$ go build -buildmode=plugin
```

主程序 mod4 直接使用 go build 命令以默认方式构建就可以了。构建完成后，将 mod2.so 及 mod3.so 复制到 mod4 所在目录下，然后运行 mod4，命令如下：

```
$ ./mod4
(0x7f3039507ba0,0xc0000a0100) * mod1.User
```

其中第 1 个地址 0x7f3039507ba0 就是 * mod1.User 的类型元数据的地址，可以通过查看当前进程地址空间中的模块布局，来确定该地址位于哪个模块中。打开另一个终端，执行命令如下：

```
$ ps aux | grep mod4
kylin 16805 0.0 0.2 751788 5880 pts/0 Sl+ 21:18 0:00 ./mod4
...
$ cat /proc/16805/maps
...
7f3038efa000-7f3038fb9000 r-xp 00000000 fd:02 1057567 ./mod3.so
7f3038fb9000-7f30391b9000 ---p 000bf000 fd:02 1057567 ./mod3.so
7f30391b9000-7f3039212000 r--p 000bf000 fd:02 1057567 ./mod3.so
7f3039212000-7f3039216000 rw-p 00118000 fd:02 1057567 ./mod3.so
7f3039216000-7f3039241000 rw-p 00000000 00:00 0
7f3039241000-7f3039300000 r-xp 00000000 fd:02 1057436 ./mod2.so
7f3039300000-7f3039500000 ---p 000bf000 fd:02 1057436 ./mod2.so
7f3039500000-7f3039559000 r--p 000bf000 fd:02 1057436 ./mod2.so
7f3039559000-7f303955d000 rw-p 00118000 fd:02 1057436 ./mod2.so
7f303955d000-7f3039588000 rw-p 00000000 00:00 0
...
```

可以看到类型元数据的地址落在了 mod2.so 的第 3 个区间内，也就是说 mod3.so 的 got 中的地址项被动态链接器修改了。假如对换一下 mod4 的 main()函数的前两行代码的顺序，也就是先加载 mod3.so，后加载 mod2.so，就会发现程序使用的 * mod1.User 的元数据位于 mod3.so 中，也就是以先加载的模块为准，感兴趣的读者可以自己尝试，这里不再赘述。

综上所述，代码中 typelinksinit 构造了各模块的 typemap(首个模块除外)，这样就实现了类型元数据间引用关系的唯一化，而在二进制模块加载时动态链接器能够使代码中引用的类型元数据地址唯一化，前者作用于类型系统内部，后者作用于类型系统的入口，从而整体上解决了多个二进制模块的类型信息不一致问题。

5.4.2 类型元数据详细讲解

在 5.1.2 节已经介绍过用来表示类型元数据的 runtime._type 类型，以及其中各个字段的含义，reflect 包中的 rtype 类型与 runtime._type 类型是等价的。本节深入研究各种类型的元数据细节，重点分析 array、slice、map、struct 及指针等几种复合数据类型的元数据结构。再结合反射提供的方法，探索类型系统是如何解析元数据的。

下面先看一下布尔、整型、浮点、复数、字符串和 unsafe. Pointer 这些基本类型，元数据中关键字段的取值如表 5-3 所示。

表 5-3 基本类型元数据中关键字段的取值

type	kind	size	ptrdata	tflag	align	fieldAlign	equal
bool	1	1	0	15	1	1	runtime. memequal8
int	2	8	0	15	8	8	runtime. memequal64
int8	3	1	0	15	1	1	runtime. memequal8
int16	4	2	0	15	2	2	runtime. memequal16
int32	5	4	0	15	4	4	runtime. memequal32
int64	6	8	0	15	8	8	runtime. memequal64
uint	7	8	0	15	8	8	runtime. memequal64
uint8	8	1	0	15	1	1	runtime. memequal8
uint16	9	2	0	15	2	2	runtime. memequal16
uint32	10	4	0	15	4	4	runtime. memequal32
uint64	11	8	0	15	8	8	runtime. memequal64
uintptr	12	8	0	15	8	8	runtime. memequal64
float32	13	4	0	7	4	4	runtime. f32equal
float64	14	8	0	7	8	8	runtime. f64equal
complex32	15	8	0	7	4	4	runtime. c64equal
complex64	16	16	0	7	8	8	runtime. c128equal
string	24	16	8	7	8	8	runtime. strequal
unsafe. Pointer	58	8	8	15	8	8	runtime. memequal64

其中有几个地方需要解释一下：

(1) unsafe. Pointer 类型的 kind 值是 58，实际上 kind 字段只有低 5 位用来表示数据类型所属的种类，第 6 位在源码中定义为 kindDirectIface，其含义是该类数据可以直接存储在 interface 中。通过 5.1 节和 5.2 节已经知道 interface 的结构实际上是个双指针，所以能够直接存储在其中的类型，本质上来讲应该都是个地址。除了地址之外，其他的值类型需要经过装箱操作。unsafe. Pointer 类型可以直接存储在 interface 中，所以其 kind 值就是原本的类型编号 26|32＝58。

(2) ptrdata 一列表示数据类型的前多少字节内包含地址，string 类型本质上是一个指针和一个整型组成的结构，在 amd64 平台上指针大小为 8 字节。unsafe. Pointer 本身是一个指针。

(3) 对于浮点、复数和 string 类型，tflag 中的 tflagRegularMemory 位没有被设置。浮点数不能直接像整型那样直接比较内存，string 包含指针，实际上数据存储在别的地方。这一点通过最后一列的 equal 函数也可以看出来。

对于复合类型而言，单个 rtype 结构就不够用了，所以会在此基础之上进行扩展，利用 struct 嵌入可以很方便地实现。用来描述 array 类型的 arrayType 定义的代码如下：

```
type arrayType struct {
    rtype
    elem * rtype //数组元素类型
    slice * rtype //切片类型
    len uintptr
}
```

其中的 rtype 嵌入 arrayType 结构中，相当于 arrayType 继承自 rtype。elem 指向数组元素的类型元数据，len 表示数组的长度，通过元素类型和长度就确定了数组的类型。slice 字段指向相同元素类型的切片对应的元数据，因为反射提供的与切片相关的函数在操作数组时需要根据 array 的元数据找到 slice 的元数据，这样直接持有一个地址更加高效。

切片类型元数据的结构比数组要简单一些，除了 rtype 和元素类型外，没有了长度字段，也不用指向其他类型，因为切片运算的结果还是切片类型，代码如下：

```
type sliceType struct {
    rtype
    elem * rtype //切片元素类型
}
```

指针类型的元数据结构和切片类型一样，除了嵌入的 rtype 之外，还包含了一个元素类型，也就是指针所指向的数据的类型，代码如下：

```
type ptrType struct {
    rtype
    elem * rtype //指向的元素类型
}
```

struct 类型的元数据结构就稍微复杂一些了，有一个 pkgPath 字段记录着该 struct 被定义在哪个包里，还有一个切片记录着一组 structField，也就是 struct 的所有字段，代码如下：

```
type structType struct {
    rtype
    pkgPath name
    fields []structField //按照在 struct 内的 offset 排列
}
```

每个 structField 用于描述 struct 的一个字段，字段必须有名字，所以 name 字段不能为空。typ 指向字段类型对应的元数据，offsetEmbed 字段是由两个值组合而成的，先把字段的偏移量的值左移一位，然后最低位用来表示是否为嵌入字段，代码如下：

```
type structField struct {
    name         name     //始终非空
    typ          * rtype  //字段的类型
    offsetEmbed uintptr   //字段偏移量、是否为嵌入字段
}
```

map 的元数据结构就更复杂了，需要记录 key、elem 及 bucket 对应的类型元数据地址，还有用来对 key 进行哈希运算的 hasher()函数，还要记录 key slot、value slot 及 bucket 的大小，flags 字段用来记录一些标志位，代码如下：

```
type mapType struct {
    rtype
    key          * rtype   //key 类型
    elem         * rtype   //元素类型
    bucket       * rtype   //内部 bucket 的类型
    hasher      func(unsafe.Pointer, uintptr) uintptr
    keysize     uint8      //key slot 大小
    valuesize   uint8      //value slot 大小
    bucketsize  uint16     //bucket 大小
    flags       uint32
}
```

其中 flags 字段的几个标志位的含义如表 5-4 所示。

表 5-4　flags 字段的几个标志位的含义

标志位	含　义
最低位	表示 key 是以间接方式存储的，因为当 key 的数据类型大小超过 128 后，就会存储地址而不是直接存储值
第二位	表示 value 是以间接方式存储的，与 key 一样，value 类型大小超过 128 后就会存储地址
第三位	表示 key 的数据类型是 reflexive 的，也就是可以使用==运算符来比较相等性
第四位	表示 map 在覆盖时 key 是否需要被再复制一次(覆盖)，否则在 key 已经存在的情况下不会对 key 进行赋值
第五位	表示 hash 函数可能会触发 panic

下面再来看一下 channel 的类型元数据结构，elem 字段指向元素类型，dir 字段存储了通道的方向，也就是 send、recv，或者既 send 又 recv，代码如下：

```
type chanType struct {
    rtype
    elem * rtype //channel 元素类型
    dir uintptr //channel 方向(send、recv)
}
```

关于 dir 字段，虽然在结构体中的类型是 uintptr，但是 reflect 包在操作该字段的时候会把它转换为 reflect.ChanDir 类型。ChanDir 类型本质上是个 int，表示的是 channel 的方向，定义了 3 个常量值：RecvDir 的值是 1，表示可以 recv；SendDir 的值是 2，表示可以 send；BothDir 是前两者的组合，值是 3，表示既能 recv 又能 send。

接下来是函数类型的元数据结构，inCount 表示输入参数的个数，outCount 表示返回值的个数。这两个 count 都是 uint16 类型，所以理论上可以有 65535 个入参，由于 outCount 的最高位被用来表示最后一个入参是否为变参(...)，所以理论上的返回值最多有 32767 个，代码如下：

```
type funcType struct {
    rtype
    inCount uint16
    outCount uint16 //最高位表示是否为变参函数
}
```

最后就是接口类型的元数据结构，与 runtime.interfacetype 是等价的，在 5.2 节已经分析过了，此处不再赘述。在 reflect 包中的定义代码如下：

```
type interfaceType struct {
    rtype
    pkgPath name
    methods []imethod
}
```

至此，总共 26 种类型都介绍完了，Go 语言中所有的内置类型、标准库类型，以及用户自定义类型都不会超出这 26 种类型。

下面来看一下，运行阶段如何根据 typeOff 定位元数据的地址，以及在存在多个模块时是如何利用各模块的 typemap 实现唯一化的，主要逻辑在 runtime.resolveTypeOff()函数中，代码如下：

```
func resolveTypeOff(ptrInModule unsafe.Pointer, off typeOff) *_type {
    if off == 0 {
        return nil
    }
    base := uintptr(ptrInModule)
    var md *moduledata
    for next := &firstmoduledata; next != nil; next = next.next {
        if base >= next.types && base < next.etypes {
            md = next
            break
        }
    }
```

```
        if md == nil {
            reflectOffsLock()
            res := reflectOffs.m[int32(off)]
            reflectOffsUnlock()
            if res == nil {
                //省略少量代码
                throw("runtime: type offset base pointer out of range")
            }
            return (*_type)(res)
        }
        if t := md.typemap[off]; t != nil {
            return t
        }
        res := md.types + uintptr(off)
        if res > md.etypes {
            //省略少量代码
            throw("runtime: type offset out of range")
        }
        return (*_type)(unsafe.Pointer(res))
    }
```

因为 typeOff 这个偏移量是相对于模块的 types 起始地址而言的，所以要通过 ptrInModule 来确定是在哪个模块中查找。该函数的逻辑大致分为以下几步：

(1) 遍历所有模块，查找 ptrInModule 这个地址落在哪个模块的 types 区间内，后续就在这个模块中查找。

(2) 如果上一步没能找到对应的模块，就到 reflectOffs 中去查找，这里面都是运行阶段通过反射机制动态创建的类型，如果找到，则直接返回。

(3) 尝试在模块的 typemap 中通过 off 查找对应的类型，如果找到，则直接返回。因为 typemap 中已经是 typelinksinit 处理好的数据，这一步实现了类型信息的唯一化。

(4) 最后才会尝试用 types 直接加上 off 作为元数据地址，只要该地址没有超出当前模块的类型数据区间就行。因为首个模块没有 typemap，所以这一步是必要的。

最后来看一下反射是如何在运行阶段创建类型的。构造对应的类型元数据并没有什么难点，关键是如何与编译阶段生成的大量类型信息整合起来。因为是运行阶段创建的类型，所以不会有重定位之类的问题，只需考虑如何根据 typeOff 来检索就好了。reflect 包中 addReflectOff()函数用来为动态生成的类型分配 typeOff，具体逻辑是在 runtime. reflect_addReflectOff()函数中实现的，reflect. addReflectOff()函数又是通过 linkname 机制链接过去的，函数的代码如下：

```
    func reflect_addReflectOff(ptr unsafe.Pointer) int32 {
        reflectOffsLock()
        if reflectOffs.m == nil {
```

```
            reflectOffs.m = make(map[int32]unsafe.Pointer)
            reflectOffs.minv = make(map[unsafe.Pointer]int32)
            reflectOffs.next = -1
        }
        id, found := reflectOffs.minv[ptr]
        if !found {
            id = reflectOffs.next
            reflectOffs.next--
            reflectOffs.m[id] = ptr
            reflectOffs.minv[ptr] = id
        }
        reflectOffsUnlock()
        return id
    }
```

在梳理该函数的逻辑之前，有必要先弄清楚 reflectOffs 的类型，代码如下：

```
    var reflectOffs struct {
        lock mutex
        next int32
        m map[int32]unsafe.Pointer
        minv map[unsafe.Pointer]int32
    }
```

其中，lock 用来保护整个 struct 中的其他字段，next 表示下一个可分配的 typeOff 值，m 是从 typeOff 值到类型元数据地址的映射，minv 是 m 的逆映射，也就是从类型元数据地址到 typeOff 的映射。理清这些之后，再来梳理上面函数的逻辑：

(1) 先加锁。

(2) 通过检查 m 是否为 nil 来判断是否已经初始化了，注意 next 的初始值是－1。

(3) 先通过元数据的地址 ptr 在 minv 里面查找，如果已经有了就不再添加了。

(4) 把 next 的值作为 typeOff 分配给 ptr，分别添加到 m 和 minv 中，然后递减 next。

(5) 解锁，返回查找到的或新分配的 typeOff。

所以运行阶段动态分配的 typeOff 都是负值，只是用作唯一 ID，并不是真正地偏移了，而编译阶段生成的 typeOff 是真正的偏移，是与本模块 types 区间起始地址的差，都是正值。返回去再看前面的 resolveTypeOff()函数，只有在通过 ptrInModule 找不到对应的二进制模块时才会查找 reflectOffs，因为编译时期生成的那些类型元数据是不可能依赖动态生成的类型元数据的，只有动态生成的类型元数据才有可能依赖动态生成的类型元数据，而动态分配的内存是不会匹配上任何一个模块的 types 区间的。

关于类型元数据的分析就到这里，笔者只是选了自己认为还算重要的几部分内容着重分析了一下，感兴趣的读者可以从 reflect 的源码中发现更多有趣的细节，这里就不占用更多篇幅了。

8min

5.4.3 对数据的操作

至此，对于反射如何解析类型元数据已经有了大致的了解，而大多数场景下使用反射的最终目的是操作数据。为了便于对数据进行操作，reflect 包提供了 Value 类型，通过该类型的一系列方法来动态操作各种数据类型。Value 类型本身是个 struct，代码如下：

```
type Value struct {
    typ * rtype
    ptr unsafe.Pointer
    flag
}
```

Value 的作用就像它的名字那样，用来装载一个值，其中的 typ 字段指向值的类型对应的元数据。ptr 字段可能是值本身(对于本质上是个地址的值，即 kindDirectIface)，也可能是一段内存的起始地址，实际的值存放在那里。flag 字段存储了一系列标志位，各个标志位的含义如表 5-5 所示。

表 5-5 flag 字段各个标志位的含义

标 志 位	含 义
flagStickyRO：1 << 5	未导出且非嵌入的字段，是只读的
flagEmbedRO：1 << 6	未导出且嵌入的字段，是只读的
flagIndir：1 << 7	ptr 字段中存储的是值的地址，而非值本身
flagAddr：1 << 8	值是可寻址的(addressable)
flagMethod：1 << 9	值是个 Method Value

其中前两个只读标志位主要是针对 struct 的字段而言的，如果目标字段也是个 struct，这些只读标志会被更内层的字段继承。flag 本质上是个 uintptr，所以至少有 32 位，最低 5 位一般与 typ.kind 的低 5 位一致，只有在值是个 Method 时例外，此时 flag 的低 5 位为 reflect.Func，高 22 位存储了 Method 在方法集中的序号，方法的接收者是通过 typ 和 ptr 来描述的，如图 5-26 所示。

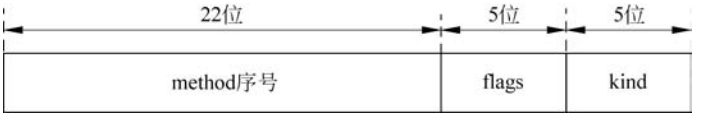

图 5-26 flag 字段的结构

再来看一下 reflect.ValueOf()函数，该函数会返回一个 Value 对象。类似于 reflect.TypeOf()函数，可以认为是反射操作数据的起点，代码如下：

```
func ValueOf(i interface{}) Value {
    if i == nil {
        return Value{}
    }
    escapes(i)
    return unpackEface(i)
}
```

一个入参，类型也是 interface{}，如果为 nil，就会返回一个零值的 Value，零值的 Value 是 Invalid 的。escapes 的作用是确保 i.data 指向的数据会逃逸，因为反射相关的代码涉及较多 unsafe 操作，编译器的逃逸分析极有可能无法追踪某些实质上逃逸了的变量，而误把它们分配到栈上，从而造成问题。后续的版本可能会允许 Value 指向栈上的值，现阶段先忽略此问题。最后的 unpackEface()函数才是关键，代码如下：

```
func unpackEface(i interface{}) Value {
    e := (*emptyInterface)(unsafe.Pointer(&i))
    t := e.typ
    if t == nil {
        return Value{}
    }
    f := flag(t.Kind())
    if ifaceIndir(t) {
        f |= flagIndir
    }
    return Value{t, e.word, f}
}
```

如果 e.typ 为 nil，也就是得不到类型元数据，就返回一个无效的 Value 对象。用 t.Kind()的返回值对 flag 进行初始化，也就是复制了 t.kind 的低 5 位。如果值本身不是个地址，还要设置 flagIndir 标志位。ptr 字段也是直接复制自 e.word，也就是 interface{}中的数据指针。

用一段实际的代码看一下 typ 和 flag 的取值，代码如下：

```
//第 5 章/code_5_13.go
type Value struct {
    typ unsafe.Pointer
    ptr unsafe.Pointer
    flag uintptr
}

func toType(p unsafe.Pointer) (t reflect.Type) {
    t = reflect.TypeOf(0)
```

```
    (*[2]unsafe.Pointer)(unsafe.Pointer(&t))[1] = p
    return
}

func main() {
    n := 6789
    s := []interface{}{
        n,
        &n,
    }
    for i, v := range s {
        r := reflect.ValueOf(v)
        p := (*Value)(unsafe.Pointer(&r))
        println(i, p.typ, p.ptr, p.flag, toType(p.typ).String())
    }
}
```

这段代码的作用是分别基于 int 和 *int 两种类型的输入，用 reflect.ValueOf()函数得到两个 Value，然后打印出 Value 的各个字段。在笔者的计算机上得到的输出如下：

```
$ ./code_5_13.exe
0 0x24c060 0xc00000c078 130 int
1 0x2487c0 0xc00000c070 22 *int
```

其中 int 类型对应的 flag 是 130＝128＋2，也就是 flagIndir 加上 kindInt。*int 类型对应的 flag 是 22，等于 kindPtr。事实上 unpackEface()函数只是简单地从 interface{}中复制了类型指针和数据指针，在把 int 类型赋值给 interface{}时发生了装箱操作，所以设置了 flagIndir。

由此看来，Value 和 interface{}非常相似，都有一个类型指针和一个数据指针，不同的是 Value 多了一个 flag 字段，基于 flag 中提供的信息可以实现很多很灵活的操作，比较典型的有如 Elem()方法和 Addr()方法。先来看一下 Elem()方法，代码如下：

```
func (v Value) Elem() Value {
    k := v.kind()
    switch k {
    case Interface:
        var eface interface{}
        if v.typ.NumMethod() == 0 {
            eface = *(*interface{})(v.ptr)
        } else {
            eface = (interface{})(*(*interface {
                M()
            })(v.ptr))
```

```
            }
            x := unpackEface(eface)
            if x.flag != 0 {
                x.flag |= v.flag.ro()
            }
            return x
        case Ptr:
            ptr := v.ptr
            if v.flag&flagIndir != 0 {
                ptr = *(*unsafe.Pointer)(ptr)
            }
            if ptr == nil {
                return Value{}
            }
            tt := (*ptrType)(unsafe.Pointer(v.typ))
            typ := tt.elem
            fl := v.flag&flagRO | flagIndir | flagAddr
            fl |= flag(typ.Kind())
            return Value{typ, ptr, fl}
        }
        panic(&ValueError{"reflect.Value.Elem", v.kind()})
    }
```

Elem()方法的功能是根据地址返回地址处存储的对象，要求 v 的 kind 必须是 Interface 或 Ptr，否则就会造成 panic。已经分析过 interface 的双指针结构，可以把它等价于一个带有类型的指针。下面先来梳理一下处理 Interface 的逻辑：

(1) 通过接口方法数判断是否为 eface，如果方法数为 0 就可以直接把 ptr 强制转换为 *interface{}类型，然后通过指针解引用操作得到 eface 的值。

(2) 对于方法数不为 0 的接口类型就是 iface，先把 ptr 强制转换为 *interface{M()}类型，然后通过指针解引用操作得到 iface 的值，再强制转换为 interface{}类型，也就是 eface。

(3) 调用 unpackEface()函数，从 eface 中提取类型指针和数据指针的值，并设置 flag 字段，返回一个新的 Value。这一步几乎等价于 ValueOf()函数。

(4) 通过设置 flag 来继承 v 的只读相关标志位。

前两步是从 ptr 地址处提取出 interface{}类型的值，第二步需要解释一下，有关不同接口类型间的强制类型转换。假如有 A、B 两个接口类型，其中 A 的方法列表是 B 方法列表的子集，那么编译器允许通过强制类型转换把 B 类型的实例转换成 A 类型。例如从 io.ReadWriter 到 io.Reader，也可以从 io.Writer 到 interface{}，因为空集是任意集合的子集，所以第二步接口中的 M 方法没有实际意义，只是告诉编译器这是个有方法的接口，双指针是 itab 指针和数据指针。

对于不同 iface 之间的强制类型转换，编译器会调用 runtime.convI2I()函数。从 iface 到 eface 的强制类型转换，编译器直接生成代码复制类型元数据指针和数据指针。

再来梳理一下处理 Ptr 时的逻辑：

(1) 检查 flag 中的 flagIndir 标志，如果是间接存储的，就进行一次指针解引用操作。

(2) 如果 ptr 为 nil，就返回一个无效的 Value。

(3) 将 typ 修改为指针元素类型对应的元数据地址。

(4) 根据 typ.Kind()函数设置新的 flag，设置 flagIndir 和 flagAddr 标志，并继承只读标志。

(5) 基于新的 typ、ptr 和 flag 构造 Value 并返回结果。

其中值得注意的是 flagAddr 标志，通俗来讲该标志位表示能够获得原始变量的地址，而不只是值的副本，Set 系列方法会检查该标志位，只有在设置了该标志位的情况下才允许修改，否则是没有意义的，会触发 panic。

Addr()方法可以认为是 Elem()方法的逆操作，功能上等价于取地址操作，要求目标必须是可定址的，也就是有 flagAddr 标志，代码如下：

```
func (v Value) Addr() Value {
    if v.flag&flagAddr == 0 {
        panic("reflect.Value.Addr of unaddressable value")
    }
    fl := v.flag & flagRO
    return Value{v.typ.ptrTo(), v.ptr, fl | flag(Ptr)}
}
```

typ.ptrTo()根据当前类型 T 得到了 *T 的元数据地址，新的 flag 就是 kindPtr 加上继承的只读标志位。ptr 的值没有改变，这一点很重要，Value 的相关方法会根据 typ 和 flag 来确定如何解释 ptr。修改一下本节最开始的示例，看一下 Elem()方法和 Addr()方法逆操作的效果，代码如下：

```
//第 5 章/code_5_14.go
func main() {
    n := 6789
    v := reflect.ValueOf(&n)
    p := (*Value)(unsafe.Pointer(&v))
    println(p.typ, p.ptr, p.flag, toType(p.typ).String())
    e := v.Elem()
    p = (*Value)(unsafe.Pointer(&e))
    println(p.typ, p.ptr, p.flag, toType(p.typ).String())
    f := e.Addr()
    p = (*Value)(unsafe.Pointer(&f))
    println(p.typ, p.ptr, p.flag, toType(p.typ).String())
}
```

在笔者的计算机上得到的输出如下：

```
$ ./code_5_14.exe
0xc187c0 0xc00000c070 22 * int
0xc1c060 0xc00000c070 386 int
0xc187c0 0xc00000c070 22 * int
```

第 2 行输出的 flag 值是 386，也就是 kindInt、flagIndir、flagAddr 组合的结果，再加上 typ 为 int，与 * int 是等价的，可以互相转换，所以在调用 json.Unmarshal()之类的函数时，需要把 struct 实例的地址传进去，这样 struct 才是可定址的，函数内部才能为 struct 的字段赋值。

通过反射来操作数据，实际上也是围绕着类型元数据展开的，本节主要分析了 Value 各个字段的作用，以及比较重要的 flagIndir 和 flagAddr 这两个标志位。以此为起点，各位有兴趣的读者可以自行阅读 reflect 源码，以此来了解更多底层实现细节，本节就讲解到这里。

5.4.4 对链接器裁剪的影响

第 4 章在讲解方法的时候，我们发现了编译器会为接收者为值类型的方法生成接收者为指针类型的包装方法，经过本章的探索，我们知道这些包装方法主要是为了支持接口，但是如果反编译或者用 nm 命令来分析可执行文件，就会发现不只是这些包装方法，就连代码中的原始方法也不一定会存在于可执行文件中。这是怎么回事呢？

道理其实很简单，链接器在生成可执行文件的时候，会对所有 OBJ 文件中的函数、方法及类型元数据等进行统计分析，对于那些确定没有用到的数据，链接器会直接将其裁剪掉，以优化最终可执行文件的大小。看起来一切顺理成章，但是又有一个问题，反射是在运行阶段工作的，通过反射还可以调用方法，那么链接器是如何保证不把反射要用的方法给裁剪掉呢？

于是笔者就做了一个小小的实验，编译一个示例，代码如下：

```
//第 5 章/code_5_15.go
type Number float64

func (n Number) IntValue() int {
    return int(n)
}

func main() {
    n := Number(9)
    v := reflect.ValueOf(n)
    _ = v
}
```

然后用 nm 命令分析得到的可执行文件，命令如下：

```
$ go tool nm code_5_15.exe | grep Number
```

结果发现 IntValue()方法被裁剪掉了,对 main()函数稍做修改,代码如下:

```
//第 5 章/code_5_16.go
func main() {
    n := Number(9)
    v := reflect.ValueOf(n)
    v.MethodByName("")
    _ = v
}
```

再次编译并用 nm 命令检查,命令如下:

```
$ go tool nm code_5_16.exe | grep Number
  48f0c0 T main.(*Number).IntValue
  48efa0 T main.Number.IntValue
```

这次 IntValue 的两个方法都被保留了下来,如果换成 v.Method(0)也能达到同样的效果。也就是说链接器裁剪的时候会检查用户代码是否会通过反射来调用方法,如果会就把该类型的方法保留下来,只有在明确确认这些方法在运行阶段不会被用到时,才可以安全地裁剪。

再次修改 main()函数的代码来进一步尝试,代码如下:

```
//第 5 章/code_5_17.go
func main() {
    n := Number(9)
    var a interface{} = n
    println(a)
    v := reflect.ValueOf("")
    v.MethodByName("")
}
```

发现这种情况下 Number 的两个方法依旧被保留了下来,从代码逻辑来看,运行阶段是不可能用到 Number 的方法的。再把 main()函数修改一下,代码如下:

```
//第 5 章/code_5_18.go
func main() {
    n := Number(9)
    println(n)
    v := reflect.ValueOf("")
    v.MethodByName("")
}
```

这次有所不同，Number 的两个方法被裁剪掉了。由此可以总结出反射影响方法裁剪的两个必要条件：一是代码中存在从目标类型到接口类型的赋值操作，因为运行阶段类型信息萃取始于接口。二是代码中调用了 MethodByName()方法或 Method()方法。因为代码中有太多灵活的逻辑，编译阶段的分析无法做到尽如人意。

5.5 本章小结

本章以空接口 interface{}为起点，初步介绍了 Go 语言的类型元数据，并且分析了数据指针带来的逃逸和装箱问题。非空接口部分，深入分析了实现方法动态派发的底层原理，还找到了编译器生成指针接收者包装方法的原因，即为了让接口方法调用更简单高效。还分析了组合式继承对方法集的影响，也是对非空接口的支持。类型断言分为 4 种场景共 8 种情况，分别通过反编译确认了汇编代码层面的实现原理。最后的反射部分，对类型系统进行了更深入的分析，并对反射如何操作数据进行了简单的探索。

接口，尤其是其背后的类型系统，有很多细节，本章无法全面地进行介绍。笔者只是把自己认为比较典型的问题拿出来分析一下，鼓励各位读者去源码中发现更多乐趣。