进阶语法——反射与 unsafe

大明

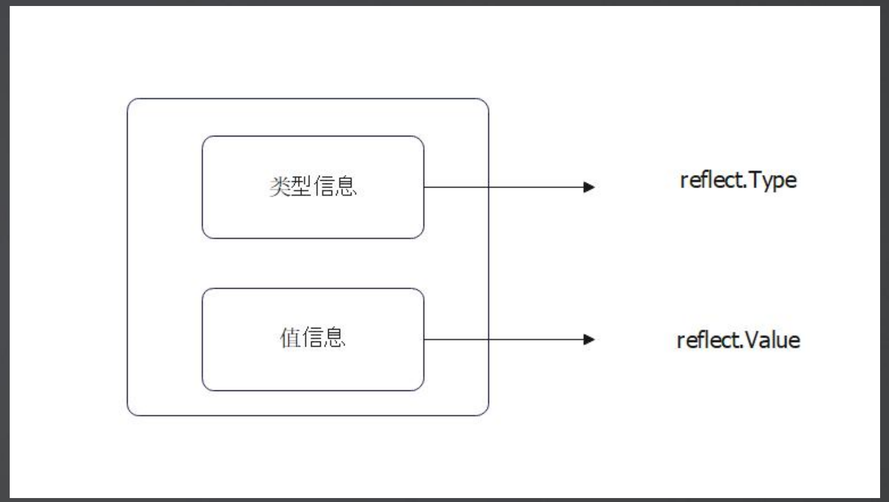
目录

Go 反射 —— 类型系统

绝大多数编程语言的类型系统都是类似的，会有声明类型、实际类型之类的分别。

在 Go 反射里面，一个实例可以看成两部分：

- 值
- 实际类型

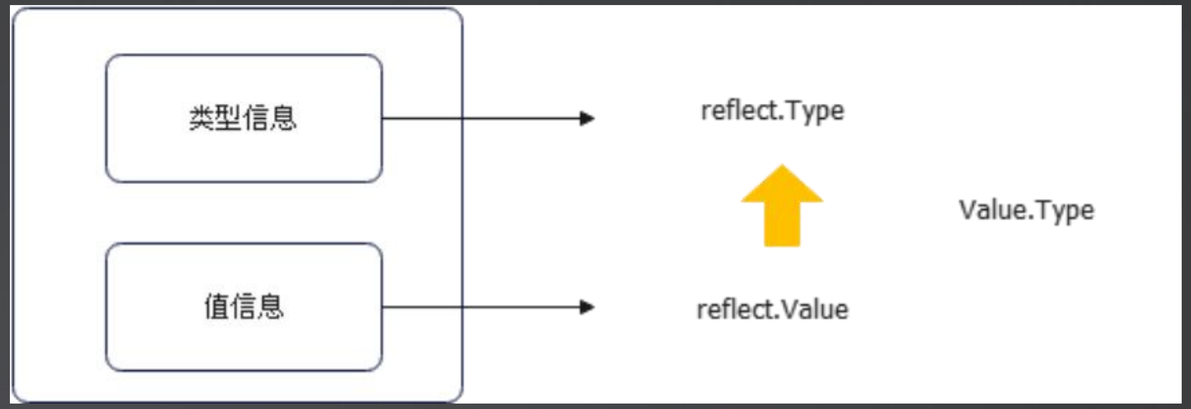


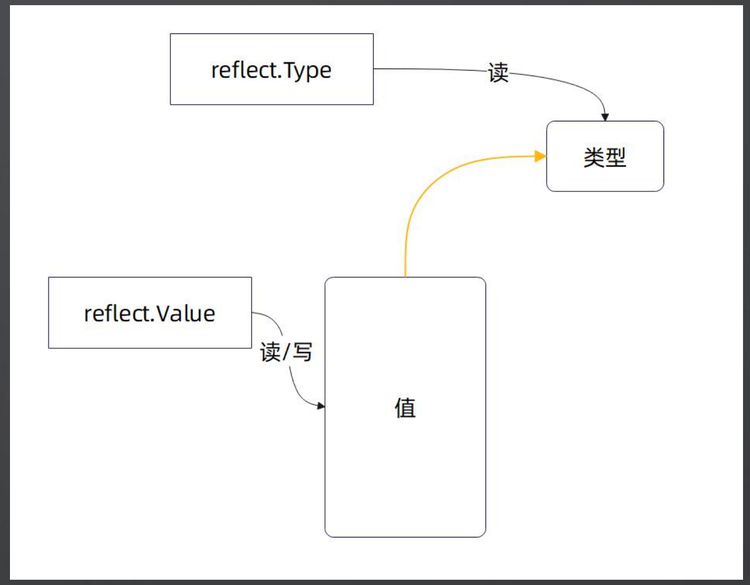
Go 反射 —— reflect.Type 和 reflect.Value

反射的相关 API 都在 reflect 包，最核心的两个：

- reflect.Value：用于操作值，部分值是可以被反射修改的
- reflect.Type：用于操作类信息，类信息是只能读取

reflect.Type 可以通过 reflect.Value 得到，但是反过来则不行。





Go 反射 —— reflect Kind

reflect 包有一个很强的假设：你知道你操作的是什么 Kind。

Kind：Kind 是一个枚举值，用来判断操作的对应类型，例如是否是指针、是否是数组、是否是切片等。

所以 reflect 的方法，如果你调用得不对，它直接就 panic。

在调用 API 之前一定要先读注释，确认什么样的情况下可以调用！！！！

```
// NumField returns a struct type's field count.
// It panics if the type's Kind is not Struct.
NumField() int

// NumIn returns a function type's input parameter count.
// It panics if the type's Kind is not Func.
NumIn() int

// NumOut returns a function type's output parameter count.
// It panics if the type's Kind is not Func.
NumOut() int
```

例如在 reflect.Type 这里，这三个方法都有对应的 Kind 必须是什么，否则 panic。

代码演示 —— 用反射输出字段名字和值

反射输出所有的字段名字，关键点在于，只有 Kind == Struct 的才有字段。

注：指针类型是没有的！

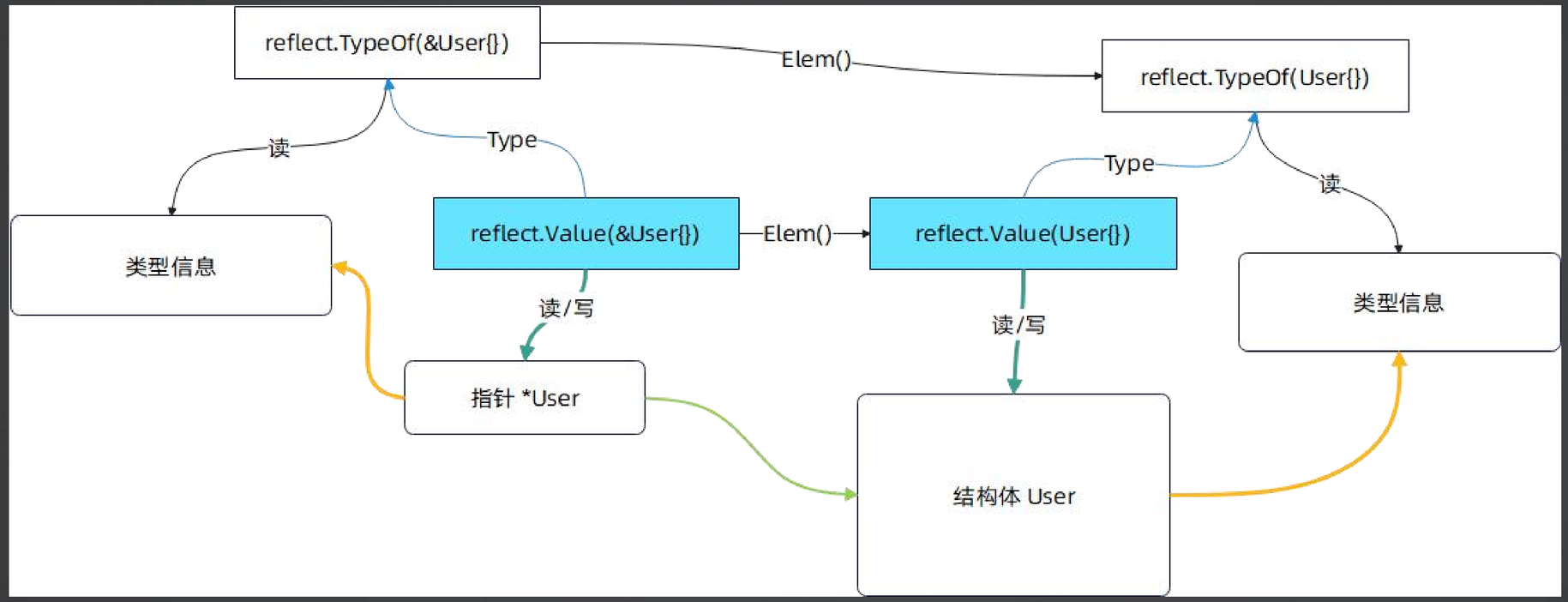
要考虑：

- 输入是不是指针，会不会是多重指针
- 输入会不会是数组或者切片
- 结构体字段会不会也是结构体

```
num := typ.NumField()
res := make(map[string]any, num)
for i := 0; i < num; i++ {
    fd := typ.Field(i)  字段信息
    fdVal := val.Field(i)  字段的值信息
    if fd.IsExported() {
        res[fd.Name] = fdVal.Interface()
    } else {
        // 为了演示效果，不公开字段我们用零值来填充
        res[fd.Name] = reflect.Zero(fd.Type).Interface()
    }
}
```

要注意字段的作用域问题。反射能够拿到私有字段的类型信息，但是拿不到值。

Go 反射 —— 指针和指针指向的结构体



代码演示 —— 用反射设置值

极客时间 训练营

可以用反射来修改一个字段的值。需要注意的是，修改字段的值之前一定要先检查 CanSet。

简单来说，就是必须使用结构体指针，那么结构体的字段才是可以修改的。

当然指针指向的对象也是可以修改的。

```go
func SetField(entity any, field string, newVal any) err
    val := reflect.ValueOf(entity)
    typ := val.Type()
    if typ.Kind() != reflect.Ptr || typ.Elem().Kind() !
        return errors.New(text: "非法类型")
    }
    typ = typ.Elem()
    val = val.Elem()
    fd := val.FieldByName(field)
    if _, found := typ.FieldByName(field); !found {
        return errors.New(text: "字段不存在")
    }
    if !fd.CanSet() {
        return errors.New(text: "不可修改字段")
    }
    fd.Set(reflect.ValueOf(newVal))
    return nil
}
```

代码演示 —— 输出方法信息并且执行调用

输出：

- 方法名
- 方法参数
- 返回值

注意：即便把字段定义为函数类型，它依然被算作字段，而不是方法。
所以严格来说，我们这里应该叫做输出函数类型的字段的信息，并且执行

代码演示 —— 输出方法信息并且执行调用

极客时间 训练营

```go
// IterateFuncs 输出方法信息，并执行调用
func IterateFuncs(val any) (map[string]*FuncInfo, error) {
    typ := reflect.TypeOf(val)
    if typ.Kind() != reflect.Struct && typ.Kind() != reflect.Ptr {
        return nil, errors.New(text:"非法类型")
    }
    num := typ.NumMethod()
    result := make(map[string]*FuncInfo, num)
    for i := 0; i < num; i++ {
        f := typ.Method(i)
        numIn := f.Type.NumIn()
        ps := make([]reflect.Value, 0, f.Type.NumIn())
        // 第一个参数永远都是接收器，类似于 java 的 this 概念
        ps = append(ps, reflect.ValueOf(val))
        in := make([]reflect.Type, 0, f.Type.NumIn())
        for j := 0; j < numIn; j++ {
            p := f.Type.In(j)
            in = append(in, p)
            if j > 0 {
                ps = append(ps, reflect.Zero(p))
            }
        }
        // 调用结果
        ret := f.Func.Call(ps)
        outNum := f.Type.NumOut()
        out := make([]reflect.Type, 0, outNum)
        res := make([]any, 0, outNum)
        for k := 0; k < outNum; k++ {
            out = append(out, f.Type.Out(k))
            res = append(res, ret[k].Interface())
        }
        result[f.Name] = &FuncInfo{
            Name:   f.Name,
            In:     in,
            Out:    out,
            Result: res,
        }
    }
    return result, nil
}
```

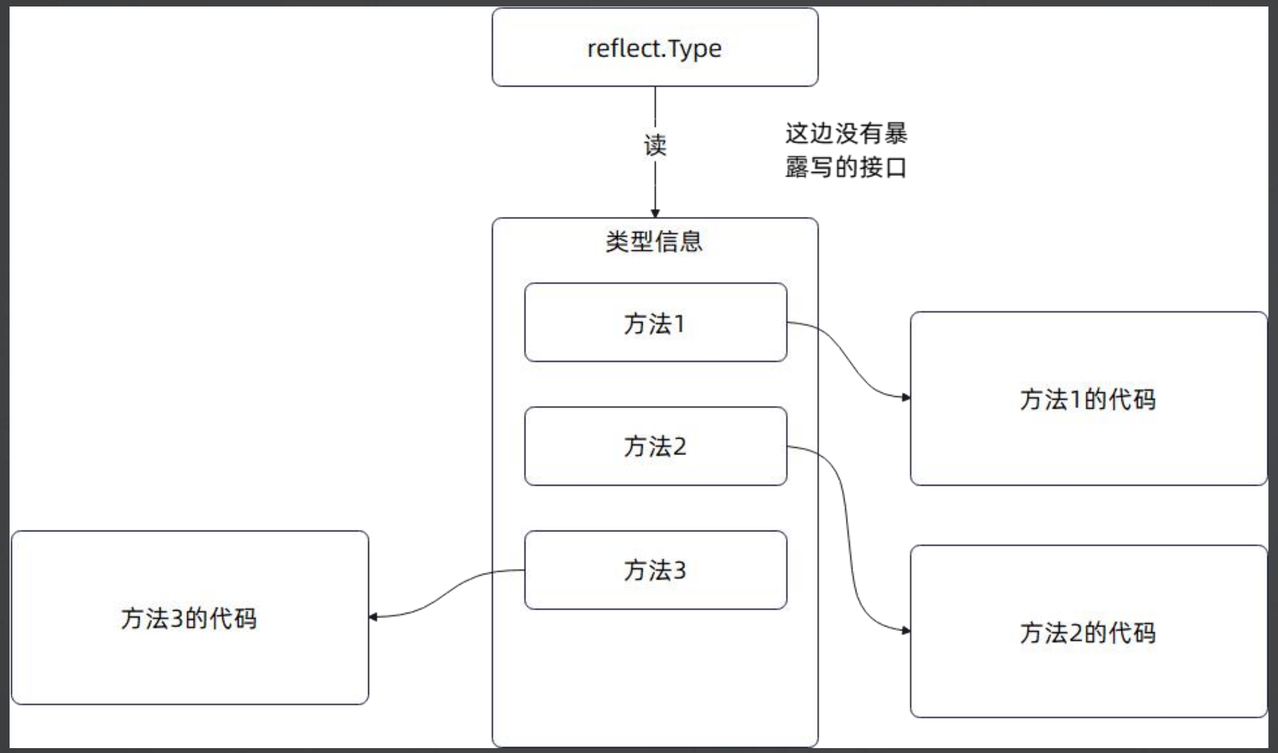
代码演示 —— 输出方法信息并且执行调用

极客时间 训练营

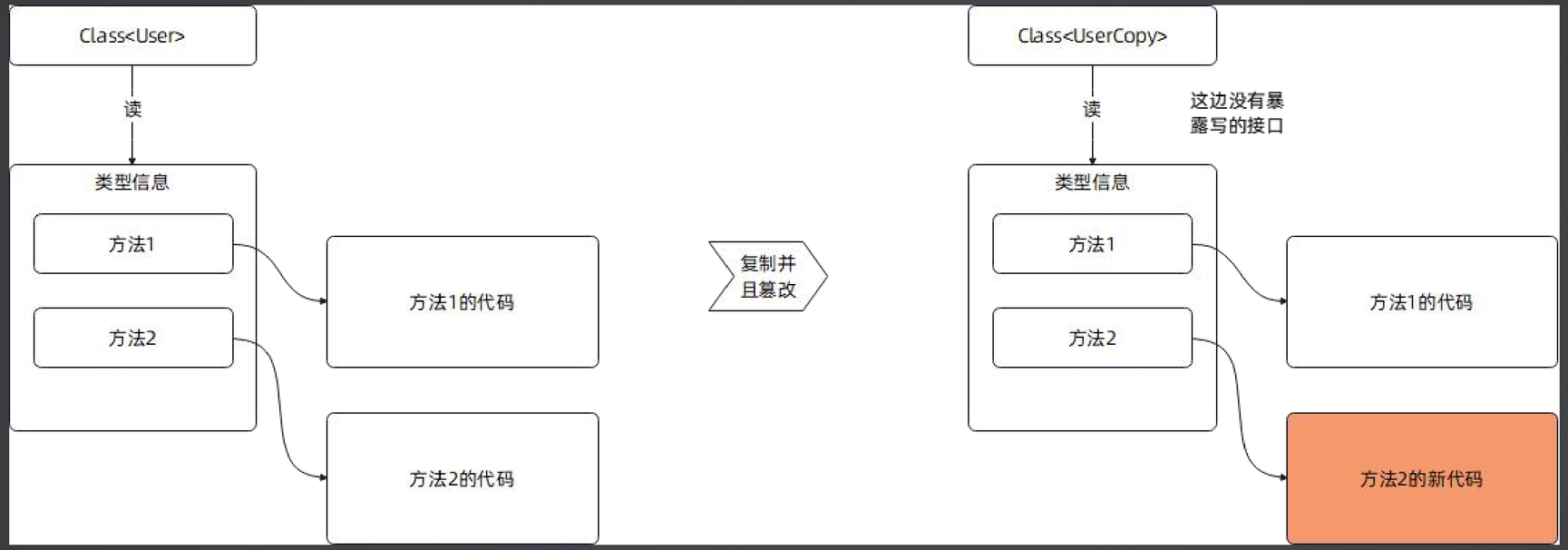
注意事项：

- 方法接收器
  - 以结构体作为输入，那么只能访问到结构体作为接收器的方法
  - 以指针作为输入，那么能访问到任何接收器的方法
- 输入的第一个参数，永远都是接收器本身

代码演示 —— 为什么不能修改方法实现



Java 篡改方法实现



Java 因为有字节码，有类加载器，有虚拟机，所以可以有各种骚操作。例如可以生成新的类文件，也可以修改类加载器，以篡改各种实现。

代码演示 —— 反射遍历

考虑遍历：

- 数组
- 切片
- 字符串
- map

Map 的遍历和其它三个不同。

```
res := make([]any, 0, val.Len())
for i := 0; i < val.Len(); i++ {
    ele := val.Index(i)
    res = append(res, ele.Interface())
}
```

```
l := val.Len()
keys := make([]any, 0, l)
values := make([]any, 0, l)
for _, k := range val.MapKeys() {
    keys = append(keys, k.Interface())
    values = append(values, val.MapIndex(k).Interface())
}
```

```
l := val.Len()
keys := make([]any, 0, l)
values := make([]any, 0, l)
itr := val.MapRange()
for itr.Next() {
    keys = append(keys, itr.Key().Interface())
    values = append(values, itr.Value().Interface())
}
```

只需要调用这些 value 上的 Set 方法就可以修改这些值，注意检查 CanSet。

开源实例 —— Dubbo-go 反射生成代理

在 Go 里面，生成代理的机制稍微有点奇怪。因为 Go 并没有提供篡改方法实现的 API，所以实际上我们是声明一个方法类型的字段。

这一个生成代理机制，是设计 RPC 框架的核心环节。

```
type UserService struct {
    GetByIdV1 func()
}

func (u *UserService) GetByIdV2() {
    fmt.Println(a...: "aa")
}
```

可以赋予新的值

没办法篡改这个方法

开源实例 —— Dubbo-go 反射生成代理

极客时间 训练营

```go
// DefaultProxyImplementFunc the default function for proxy impl
func DefaultProxyImplementFunc(p *Proxy, v common.RPCService) {
    // check parameters, incoming interface must be a elem's pointer.
    valueOf := reflect.ValueOf(v)

    valueOfElem := valueOf.Elem()
    typeOf := valueOfElem.Type()

    // check incoming interface, incoming interface's elem must be a str
    if typeOf.Kind() != reflect.Struct {
        logger.Errorf(fmt: "The type of RPCService(=\"%T\") must be a po
        return
    }

    makeDubboCallProxy := func(methodName string, outs []reflect.Type) 
        return func(in []reflect.Value) []reflect.Value {...}
    }
```

makeDubboCallProxy 就是我们的新值，用来取代旧的。

```go
numField := valueOfElem.NumField()
for i := 0; i < numField; i++ {
    t := typeOf.Field(i)
    methodName := t.Tag.Get(key: "dubbo")
    if methodName == "" {...}
    f := valueOfElem.Field(i)
    if f.Kind() == reflect.Func && f.IsValid() && f.CanSet()
        outNum := t.Type.NumOut()

        if outNum != 1 && outNum != 2 {...}

        // The latest return type of the method must be error
        if returnType := t.Type.Out(outNum - 1); returnType !

        funcOuts := make([]reflect.Type, outNum)
        for i := 0; i < outNum; i++ {...}

        // do method proxy here:
        f.Set(reflect.MakeFunc(f.Type(), makeDubboCallProxy(m
        logger.Debugf(fmt: "set method [%s]", methodName)
    }
```

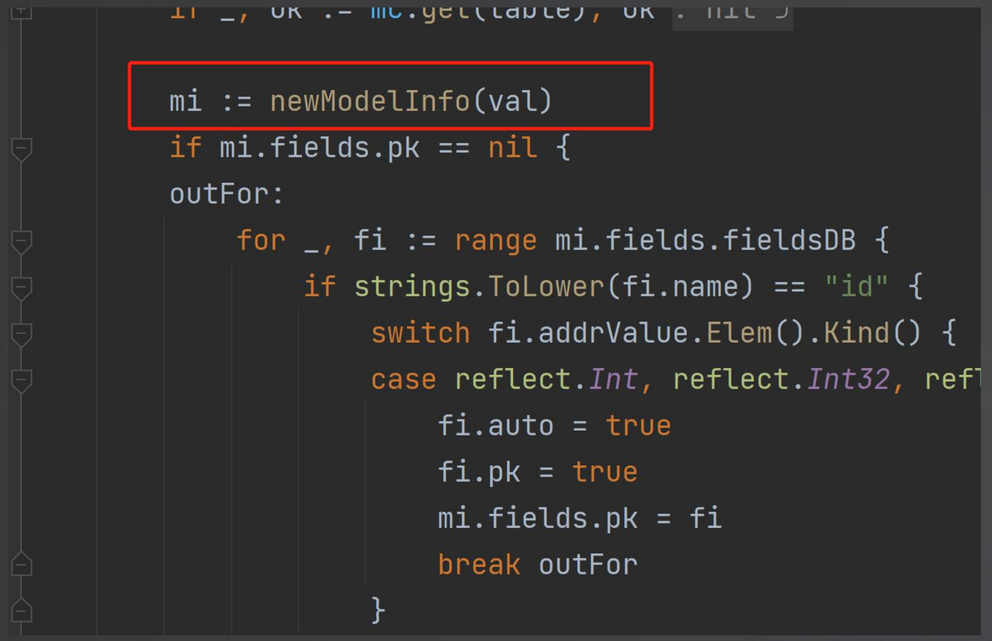
开源实例 —— Beego 反射解析模型数据

在 ORM 框架里面，一个很重要的环境是解析模型定义，比如说用户定义一个 User，要从里面解析出来表名、列名、主键、外键、索引、关联关系等。

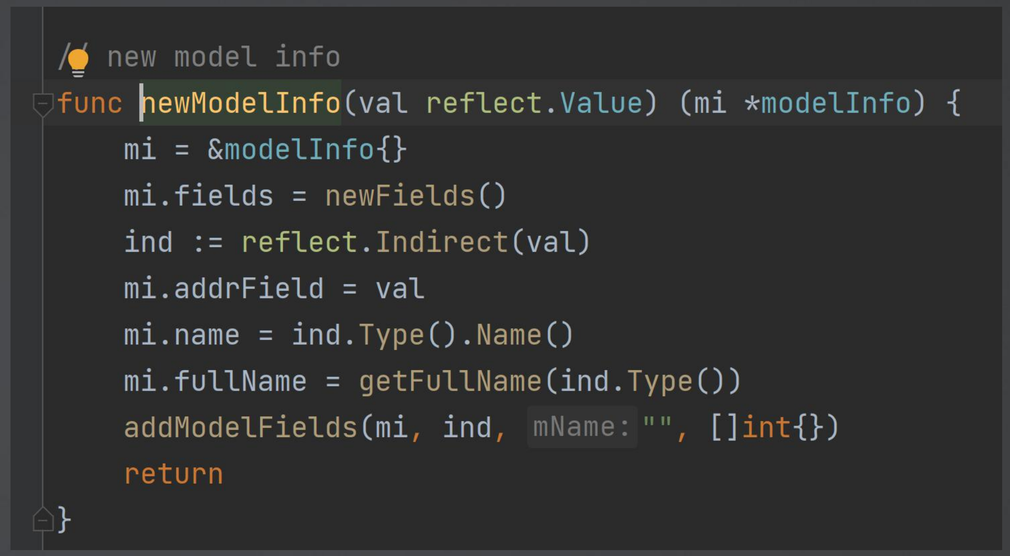
Beego 的这部分工作在 orm 包下面的 `modelCache` 的 register 方法里面完成。

```
// register register models to model cache
func (mc *modelCache) register(prefixOrSuffixStr string, pr
    for _, model := range models {
        val := reflect.ValueOf(model)
        typ := reflect.Indirect(val).Type()

        if val.Kind() != reflect.Ptr {...}
        // For this case:
        // u := &User{}
        // registerModel(&u)
        if typ.Kind() == reflect.Ptr {...}
        if val.Elem().Kind() == reflect.Slice {
            val = reflect.New(val.Elem().Type().Elem())
        }
        table := getTableName(val)
```

开源实例 —— Beego 反射解析模型数据



```
if _, ok := mc.get(table); ok { nil }

mi := newModelInfo(val)
if mi.fields.pk == nil {
outFor:
    for _, fi := range mi.fields.fieldsDB {
        if strings.ToLower(fi.name) == "id" {
            switch fi.addrValue.Elem().Kind() {
            case reflect.Int, reflect.Int32, ref
                fi.auto = true
                fi.pk = true
                mi.fields.pk = fi
                break outFor
            }
```



```
// new model info
func newModelInfo(val reflect.Value) (mi *modelInfo) {
    mi = &modelInfo{}
    mi.fields = newFields()
    ind := reflect.Indirect(val)
    mi.addrField = val
    mi.name = ind.Type().Name()
    mi.fullName = getFullName(ind.Type())
    addModelFields(mi, ind, mName: "", []int{})
    return
}
```

开源实例 —— Beego 反射解析模型数据

解析模型元数据这一步虽然很重要，但是不需要纠缠细节。

因为不同的人设计 ORM 框架，模型定义规范都是不同的。但是这些 ORM 框架需要的元数据总是类似的，总结出来这一点就可以。

```go
// index: FieldByIndex returns the nested field corresponding to index
func addModelFields(mi *modelInfo, ind reflect.Value, mName string, index []int) {
    var (
        err error
        fi  *fieldInfo
        sf  reflect.StructField
    )

    for i := 0; i < ind.NumField(); i++ {
        field := ind.Field(i)
        sf = ind.Type().Field(i)
        // if the field is unexported skip
        if sf.PkgPath != "" {
            continue
        }
        // add anonymous struct fields
        if sf.Anonymous {
            addModelFields(mi, field, mName+"."+sf.Name, append(index, i))
            continue
        }
```

一般来说考虑组合情况的时候，就难免要使用递归

开源实例 —— GORM 反射解析模型数据

GORM 也是类似，它将模型元数据称为 Schema。

核心代码在`ParseWithSpecialTableName`中。

右边是入口，一大堆的校验揭示了 GORM 支持什么样的模型定义。

```go
// ParseWithSpecialTableName get data type from d
func ParseWithSpecialTableName(dest interface{},
    if dest == nil : nil, fmt.Errorf("%w: %+v", E

    value := reflect.ValueOf(dest)
    if value.Kind() == reflect.Ptr && value.IsNil
        value = reflect.New(value.Type().Elem())
    }
    modelType := reflect.Indirect(value).Type()

    if modelType.Kind() == reflect.Interface {
        modelType = reflect.Indirect(reflect.Valu
    }

    for modelType.Kind() == reflect.Slice || mode
        modelType = modelType.Elem()
    }

    if modelType.Kind() != reflect.Struct {
```

开源实例 —— GORM 反射解析模型数据

极客时间 训练营

```go
for i := 0; i < modelType.NumField(); i++ {
    if fieldStruct := modelType.Field(i); ast.IsExported(fieldStruct.Name) {
        if field := schema.ParseField(fieldStruct); field.EmbeddedSchema != nil {
            schema.Fields = append(schema.Fields, field.EmbeddedSchema.Fields...)
        } else {
            schema.Fields = append(schema.Fields, field)
        }
    }
}
```

逐个字段解析，可以清晰看到，它只解析公开字段。

```go
// ParseField parses reflect.StructField to Field
func (schema *Schema) ParseField(fieldStruct reflect.StructField) *Field {
    var (
        err         error
        tagSetting = ParseTagSetting(fieldStruct.Tag.Get(key: "gorm"), sep: ";")
    )

    field := &Field{
        Name:                   fieldStruct.Name,
        DBName:                 tagSetting["COLUMN"],
        BindNames:              []string{fieldStruct.Name},
        FieldType:              fieldStruct.Type,
        IndirectFieldType:      fieldStruct.Type,
```

解析 tag

GORM 利用 tag 来允许用户设置一些对字段的描述，例如是否是主键、是否允许自增。

开源实例 —— GORM 反射解析模型数据

官网定义的模型，使用了 tag （标签）。

```go
// gorm.Model 的定义
type Model struct {
  ID        uint           `gorm:"primaryKey"`
  CreatedAt time.Time
  UpdatedAt time.Time
  DeletedAt gorm.DeletedAt `gorm:"index"`
}
```

ParseField 方法剩余部分及其复杂，不用看。因为它都是根据 GORM 的设计来的。如果我们设计的 ORM 不一样，那么这些代码就不具备参考价值。

```go
// ParseField parses reflect.StructField to Field
func (schema *Schema) ParseField(fieldStruct reflect.StructField) *Field {
    var (
        err        error
        tagSetting = ParseTagSetting(fieldStruct.Tag.Get(key: "gorm"), sep: ";")
    )

    field := &Field{
        Name:                   fieldStruct.Name,
        DBName:                 tagSetting["COLUMN"],
        BindNames:              []string{fieldStruct.Name},
        FieldType:              fieldStruct.Type,
        IndirectFieldType:      fieldStruct.Type,
```

解析 tag

开源实例 —— Beego 与 GORM 模型元数据的对比

极客时间 训练营

```go
// single model info
type modelInfo struct {
    manual    bool
    isThrough bool
    pkg       string
    name      string
    fullName  string
    table     string
    model     interface{}
    fields    *fields
    addrField reflect.Value // sto
    uniques   []string
}
```

```go
// field info collection
type fields struct {
    pk            *fieldInfo
    columns       map[string]*fieldInfo
    fields        map[string]*fieldInfo
    fieldsLow     map[string]*fieldInfo
    fieldsByType  map[int][]*fieldInfo
    fieldsRel     []*fieldInfo
    fieldsReverse []*fieldInfo
    fieldsDB      []*fieldInfo
    rels          []*fieldInfo
    orders        []string
    dbcols        []string
}
```

```go
type Schema struct {
    Name                          string
    ModelType                     reflect.Type
    Table                         string
    PrioritizedPrimaryField       *Field
    DBNames                       []string
    PrimaryFields                 []*Field
    PrimaryFieldDBNames           []string
    Fields                        []*Field
    FieldsByName                  map[string]*Field
    FieldsByDBName                map[string]*Field
    FieldsWithDefaultDBValue      []*Field // fields with defa
    Relationships                 Relationships
    CreateClauses                 []clause.Interface
    QueryClauses                  []clause.Interface
    UpdateClauses                 []clause.Interface
    DeleteClauses                 []clause.Interface
    BeforeCreate, AfterCreate     bool
    BeforeUpdate, AfterUpdate     bool
    BeforeDelete, AfterDelete     bool
    BeforeSave, AfterSave         bool
    AfterFind                     bool
```

表名、列名、主键、外键、索引、关联关系。我们核心是学习这两个框架是怎么表达这些信息，也就是 modelInfo 和 Schema 两个结构体的定义。解析过程就是个累活，但是没啥技术含量。

Go 反射编程小技巧

极客时间 训练营

- 读写值，使用 reflect.Value
- 读取类型信息，使用 reflect.Type
- 时刻注意你现在操作的类型是不是指针。指针和指针指向的对象在反射层面上是两个东西
- 大多数情况，指针类型对应的 reflect.Type 毫无用处。我们操作的都是指针指向的那个类型
- 没有足够的测试就不要用反射，因为反射 API 充斥着 panic
- 切片和数组在反射上也是两个东西
- 方法类型的字段和方法，在反射上也是两个不同的东西

Go 反射面试要点

Go 反射面得很少，因为 Go 反射本身是一个写代码用的，理论上的东西不太多。

- 什么是反射？反射可以看做是对对象和对类型的描述，而我们可以通过反射来间接操作对象。
- 反射的使用场景？一大堆，基本上任何高级框架都会用反射，ORM 是一个典型例子。Beego 的 controller 模式的 Web 框架也利用了反射。
- 能不能通过反射修改方法？不能。为什么不能？go runtime 没暴露接口。
- 什么样的字段可以被反射修改？有一个方法 CanSet 可以判断，简单来说就是 addressable

目录

Go unsafe —— 对象内存布局

要理解 unsafe，核心就是要理解 Go 中一个对象在内存中究竟是怎么布局的。

需要掌握：

- 计算地址
- 计算偏移量
- 直接操作内存

```go
// PrintFieldOffset 用来打印字段偏移量
// 用于研究内存布局
// 只接受结构体作为输入
func PrintFieldOffset(entity any) {
    typ := reflect.TypeOf(entity)
    for i := 0; i < typ.NumField(); i++ {
        fd := typ.Field(i)
        fmt.Println(fd.Offset)
    }
}
```

感兴趣的同学可以跑一下这个代码，看看不同写法各种偏移量怎么算。

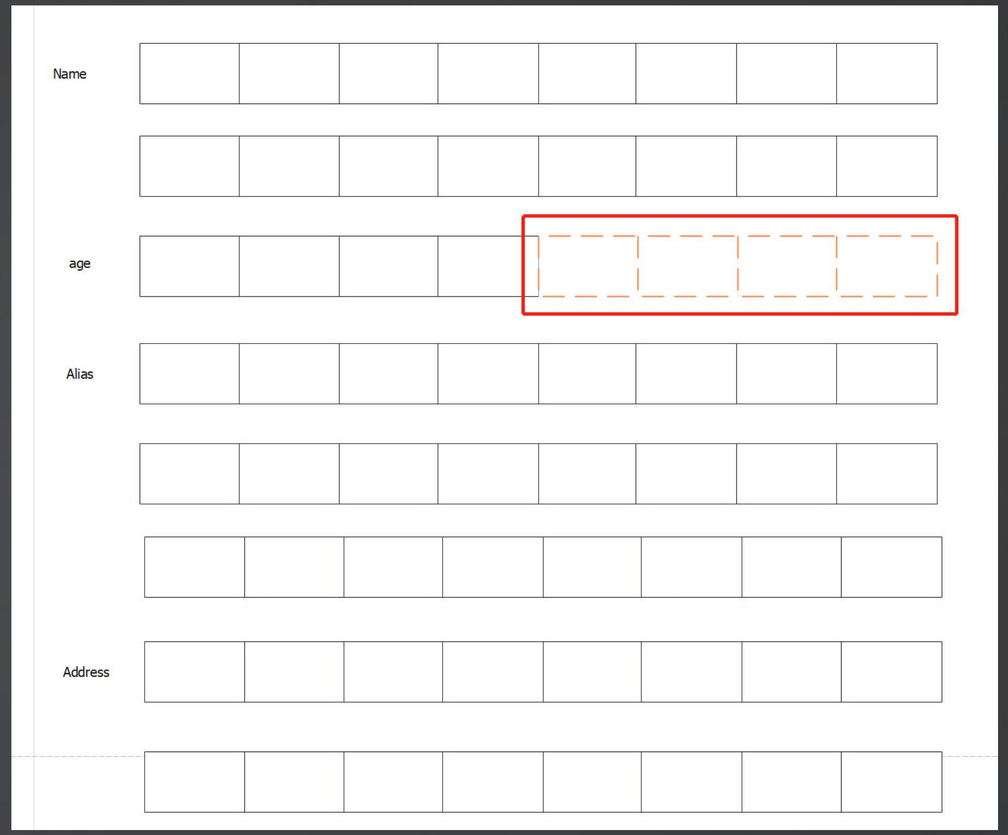
Go unsafe —— 对象内存布局例子

```go
type User struct {
    Name    string
    age     int32
    Alias   []byte
    Address string
}
```

```
» ✔ Tests passed: 1 of 1 test – 2 sec 257 ms
7 ms  === RUN   TestPrintFieldOffset
      64
      Name: 0
      age: 16
      Alias: 24
      Address: 48
      --- PASS: TestPrintFieldOffset (0.00s)
```

64 是结构体的总大小。

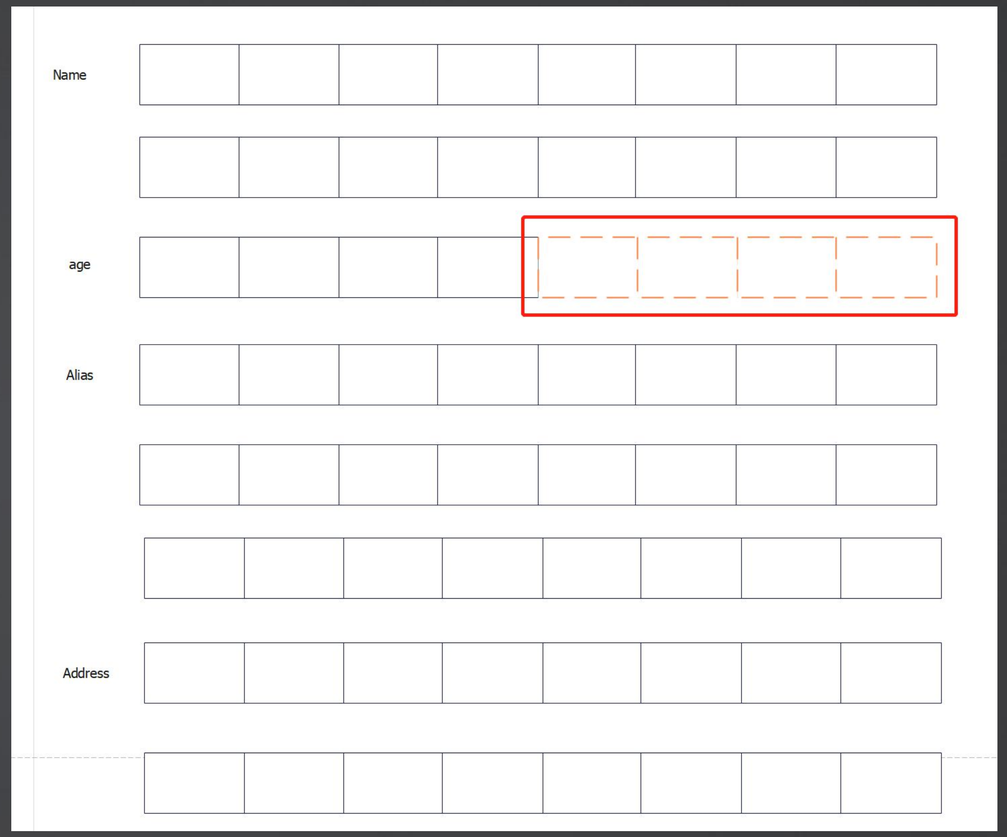
问题在于，为什么 Alias 的偏移量是 24，按照道理来说应该是20?



Go unsafe —— Go 对齐规则

按照字长对齐。因为 Go 本身每一次访问内存都是按照字长的倍数来访问的。

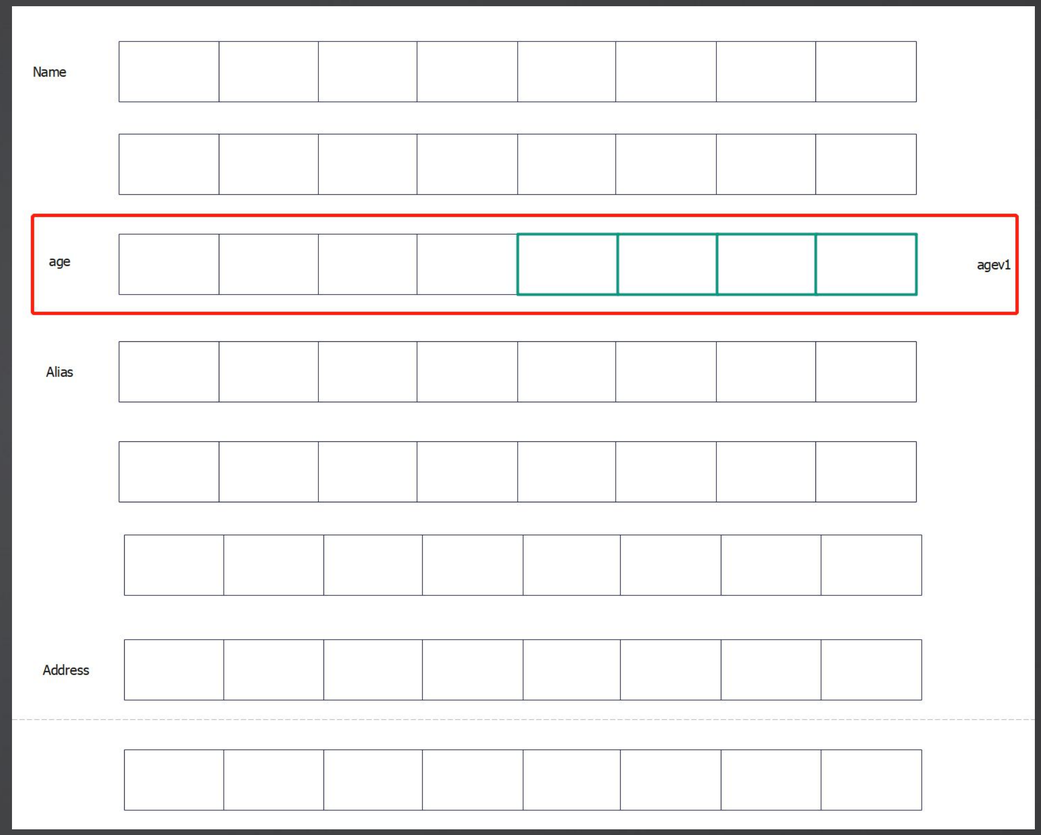
- 在 32 位字长机器上，就是按照 4 个字节对齐
- 在 64 位字长机器上，就是按照 8 个字节对齐



Go unsafe —— 对象内存布局例子

```
type User struct {
    Name    string
    age     int32
    agev1   int32
    Alias   []byte
    Address string
}
```

```
64
Name: 0
age: 16
agev1: 20
Alias: 24
Address: 48
```



64 是结构体的总大小。

因为 Go 是按照字长来对齐的，所以在64位机器上，age + agev1 恰好一个字长，所以 Alias 的偏移量其实没有变。

代码演示 —— 使用 unsafe 来读写字段

注意点：unsafe 操作的是内存，本质上是对象的起始地址。

读：*(*T)(ptr)，T 是目标类型，如果类型不知道，只能拿到反射的 Type，那么可以用reflect.NewAt(typ, ptr).Elem()。

写：*(*T)(ptr) = T，T 是目标类型。

ptr 是字段偏移量：

ptr = 结构体起始地址 + 字段偏移量

```go
type UnsafeAccessor struct {
    fields     map[string]FieldMeta
    entityAddr unsafe.Pointer
}
```

主要是字段偏移量

结构体起始地址

```go
func NewUnsafeAccessor(entity interface{}) (*UnsafeAccessor, error) {
    if entity == nil : nil, errors.New("invalid entity")
    val := reflect.ValueOf(entity)
    typ := reflect.TypeOf(entity)
    if typ.Kind() != reflect.Pointer || typ.Elem().Kind() != reflect.Struct : nil,
    fields := make(map[string]FieldMeta, typ.Elem().NumField())
    elemType := typ.Elem()
    for i := 0; i < elemType.NumField(); i++ {
        fd := elemType.Field(i)
        fields[fd.Name] = FieldMeta{offset: fd.Offset}
    }
    return &UnsafeAccessor{entityAddr: val.UnsafePointer(), fields: fields}, nil
}
```

字段偏移量

起始地址

代码演示 —— 使用 unsafe 来读写字段

极客时间 训练营

```go
func (u *UnsafeAccessor) Field(field string) (int, error) {
    fdMeta, ok := u.fields[field]
    if !ok : 0, fmt.Errorf("invalid field %s", field)
    ptr := unsafe.Pointer(uintptr(u.entityAddr) + fdMeta.offset)
    if ptr == nil {
        return 0, fmt.Errorf("invalid address of the field: #{field}")
    }
    res := *(*int)(ptr)
    return res, nil
}
```

关键操作

```go
func (u *UnsafeAccessor) SetField(field string, val int) error {
    fdMeta, ok := u.fields[field]
    if !ok {
        return fmt.Errorf("invalid field #{field}")
    }
    ptr := unsafe.Pointer(uintptr(u.entityAddr) + fdMeta.offset)
    if ptr == nil {
        return fmt.Errorf("invalid address of the field: #{field}")
    }
    *(*int)(ptr) = val
    return nil
}
```

res := *(*T)(ptr)

*(*T)(ptr)=newVal

Go unsafe —— unsafe.Pointer 和 uintptr

前面我们使用了 unsafe.Pointer 和 uintptr，这两者都代表指针，那么有什么区别?

- **unsafe.Pointer**：是 Go 层面的指针，GC 会维护 unsafe.Pointer 的值
- **uintptr**：直接就是一个数字，代表的是一个内存地址

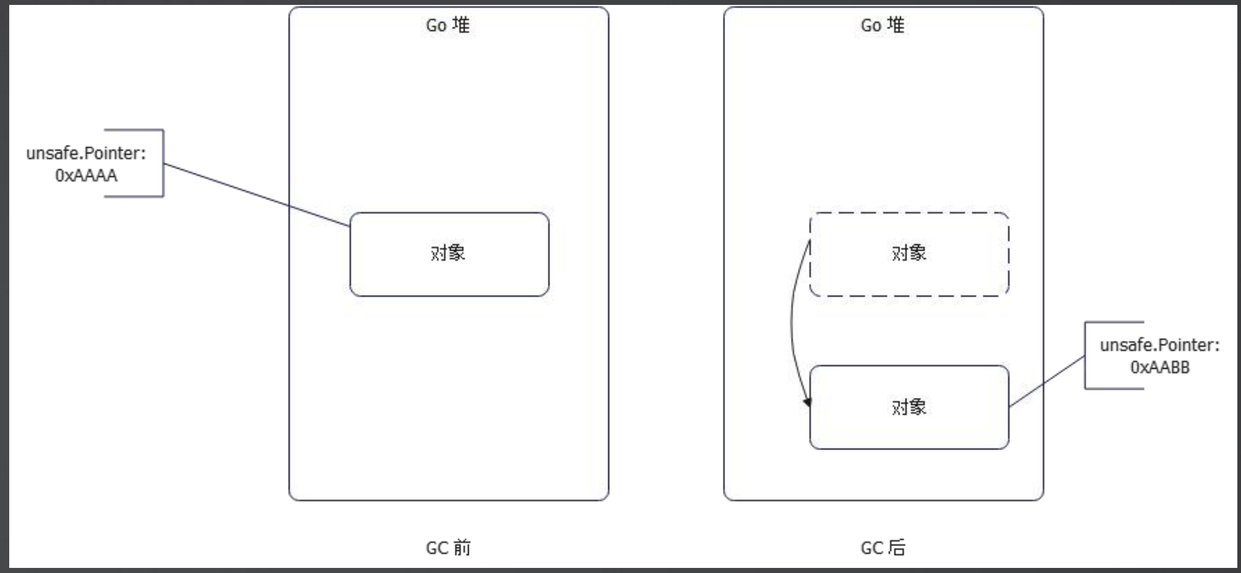
```go
type UnsafeAccessor struct {
    fields     map[string]FieldMeta
    entityAddr unsafe.Pointer
}
```

```go
type FieldMeta struct {
    // offset 后期在我们考虑组
    offset uintptr
}
```

Go unsafe —— unsafe.Pointer 和 GC

假设说 GC 前一个 unsafe.Pointer 代表对象的指针，它此时指向的地址是 0xAAAA。

如果发生了 GC， GC 之后这个对象依旧存活，但是此时这个对象被复制过去了另外一个位置（Go GC 算法是标记-复制）。那么此时代表对象的 unsafe.Pointer 会被 GC 修正，指向新的地址 0xAABB。



Go unsafe —— uintptr 使用误区

如果使用 uintptr 来保存对象的起始地址，那么如果发生 GC 了，原本的代码会直接崩溃。

例如在 GC 前，计算到的 entityAddr = 0xAAAA，那么 GC 后因为复制的原因，实际上的地址变成了 0xAABB。

因为 GC 不会维护 uintptr 变量，所以 entityAddr 还是0xAAAA，这个时候再用 0xAAAA 作为起始地址去访问字段，就不知道访问到什么东西了。

```
type UnsafeAccessor struct {
    fields     map[string]FieldMeta
    entityAddr unsafe.Pointer
}
```

```
type UnsafeAccessorV1 struct {
    fields     map[string]FieldMeta
    entityAddr uintptr
}
```

Go unsafe —— uintptr 使用误区

极客时间 训练营

但是 uintptr 可以用于表达相对的量。

例如字段偏移量。这个字段的偏移量是不管怎么 GC 都不会变的。

如果怕出错，那么就只在进行地址运算的时候使用 uintptr，其它时候都用 unsafe.Pointer。

```
type FieldMeta struct {
    // offset 后期在我们考虑组合，
    offset uintptr
}
```

unsafe 面试要点

极客时间 训练营

- uintptr 和 unsafe.Pointer 的区别：前者代表的是一个具体的地址，后者代表的是一个逻辑上的指针。后者在 GC 等情况下，go runtime 会帮你调整，使其永远指向真实存放对象的地址。
- Go 对象是怎么对齐的？按照字长。有些比较恶心的面试官可能要你手动演示如何对齐，或者写一个对象问你怎么计算对象的大小。
- 怎么计算对象地址？对象的起始地址是通过反射来获取，对象内部字段的地址是通过起始地址 + 字段偏移量来计算。
- unsafe 为什么比反射高效？可以简单认为反射帮我们封装了很多 unsafe 的操作，所以我们直接使用 unsafe 绕开了这种封装的开销。有点像是我们不用 ORM 框架，而是直接自己写 SQL 执行查询。

作业

极客时间 训练营

- 利用反射为结构体构造 INSERT 查询。
- 使用反射和 unsafe 读写字段，并且写基准测试，比较两者的性能差异。这个性能差异受到组合的影响，有余力的同学可以比较不同组合深度的性能差异。

极客时间 训练营

Q & A

THANKS