

[Discover Packages](#) > [Standard library](#) > [reflect](#) 


reflect

package

standard library

Version: [go1.20.1](#) **Latest** | Published: Feb 14, 2023 | License: [BSD-3-Clause](#) | Imports: [13](#) |Imported by: [365,556](#)

Details

 Valid [go.mod](#) file  Redistributable license  Tagged version  Stable version [Learn more](#)

Repository

[cs.opensource.google/go/go](#)

Links

 [Report a Vulnerability](#) Documentation 

<> Documentation

Overview

Package `reflect` implements run-time reflection, allowing a program to manipulate objects with arbitrary types. The typical use is to take a value with static type `interface{}` and extract its dynamic type information by calling `TypeOf`, which returns a `Type`.

A call to `ValueOf` returns a `Value` representing the run-time data. `Zero` takes a `Type` and returns a `Value` representing a zero value for that type.

See "The Laws of Reflection" for an introduction to reflection in Go:

https://golang.org/doc/articles/laws_of_reflection.html

Index

Constants

```
func Copy(dst, src Value) int
```

```
func DeepEqual(x, y any) bool
```

```
func Swapper(slice any) func(i, j int)
```

```
type ChanDir
```

```
func (d ChanDir) String() string
```

```
type Kind
```

```
func (k Kind) String() string
```

```
type MapIter
```

```
func (iter *MapIter) Key() Value
```

```
func (iter *MapIter) Next() bool
```

```
func (iter *MapIter) Reset(v Value)
```

```
func (iter *MapIter) Value() Value
type Method
func (m Method) IsExported() bool
type SelectCase
type SelectDir
type SliceHeader
type StringHeader
type StructField
func VisibleFields(t Type) []StructField
func (f StructField) IsExported() bool
type StructTag
func (tag StructTag) Get(key string) string
func (tag StructTag) Lookup(key string) (value string, ok bool)
type Type
func ArrayOf(length int, elem Type) Type
func ChanOf(dir ChanDir, t Type) Type
func FuncOf(in, out []Type, variadic bool) Type
func MapOf(key, elem Type) Type
func PointerTo(t Type) Type
func PtrTo(t Type) Type
func SliceOf(t Type) Type
func StructOf(fields []StructField) Type
func TypeOf(i any) Type
type Value
func Append(s Value, x ...Value) Value
func AppendSlice(s, t Value) Value
func Indirect(v Value) Value
func MakeChan(typ Type, buffer int) Value
func MakeFunc(typ Type, fn func(args []Value) (results []Value)) Value
func MakeMap(typ Type) Value
func MakeMapWithSize(typ Type, n int) Value
func MakeSlice(typ Type, len, cap int) Value
func New(typ Type) Value
func NewAt(typ Type, p unsafe.Pointer) Value
func Select(cases []SelectCase) (chosen int, recv Value, recvOK bool)
func ValueOf(i any) Value
func Zero(typ Type) Value
func (v Value) Addr() Value
func (v Value) Bool() bool
func (v Value) Bytes() []byte
func (v Value) Call(in []Value) []Value
func (v Value) CallSlice(in []Value) []Value
func (v Value) CanAddr() bool
func (v Value) CanComplex() bool
func (v Value) CanConvert(t Type) bool
func (v Value) CanFloat() bool
```

func (v Value) CanInt() bool
func (v Value) CanInterface() bool
func (v Value) CanSet() bool
func (v Value) CanUint() bool
func (v Value) Cap() int
func (v Value) Close()
func (v Value) Comparable() bool
func (v Value) Complex() complex128
func (v Value) Convert(t Type) Value
func (v Value) Elem() Value
func (v Value) Equal(u Value) bool
func (v Value) Field(i int) Value
func (v Value) FieldByIndex(index []int) Value
func (v Value) FieldByIndexErr(index []int) (Value, error)
func (v Value) FieldByName(name string) Value
func (v Value) FieldByNameFunc(match func(string) bool) Value
func (v Value) Float() float64
func (v Value) Grow(n int)
func (v Value) Index(i int) Value
func (v Value) Int() int64
func (v Value) Interface() (i any)
func (v Value) InterfaceData() [2]uintptr **DEPRECATED**
func (v Value) IsNil() bool
func (v Value) IsValid() bool
func (v Value) IsZero() bool
func (v Value) Kind() Kind
func (v Value) Len() int
func (v Value) MapIndex(key Value) Value
func (v Value) MapKeys() []Value
func (v Value) MapRange() *MapIter
func (v Value) Method(i int) Value
func (v Value) MethodByName(name string) Value
func (v Value) NumField() int
func (v Value) NumMethod() int
func (v Value) OverflowComplex(x complex128) bool
func (v Value) OverflowFloat(x float64) bool
func (v Value) OverflowInt(x int64) bool
func (v Value) OverflowUint(x uint64) bool
func (v Value) Pointer() uintptr
func (v Value) Recv() (x Value, ok bool)
func (v Value) Send(x Value)
func (v Value) Set(x Value)
func (v Value) SetBool(x bool)
func (v Value) SetBytes(x []byte)
func (v Value) SetCap(n int)
func (v Value) SetComplex(x complex128)

```
func (v Value) SetFloat(x float64)
func (v Value) SetInt(x int64)
func (v Value) SetIterKey(iter *MapIter)
func (v Value) SetIterValue(iter *MapIter)
func (v Value) SetLen(n int)
func (v Value) SetMapIndex(key, elem Value)
func (v Value) SetPointer(x unsafe.Pointer)
func (v Value) SetString(x string)
func (v Value) SetUint(x uint64)
func (v Value) SetZero()
func (v Value) Slice(i, j int) Value
func (v Value) Slice3(i, j, k int) Value
func (v Value) String() string
func (v Value) TryRecv() (x Value, ok bool)
func (v Value) TrySend(x Value) bool
func (v Value) Type() Type
func (v Value) Uint() uint64
func (v Value) UnsafeAddr() uintptr
func (v Value) UnsafePointer() unsafe.Pointer
```

type `ValueError`

```
func (e *ValueError) Error() string
```

Bugs

Examples

Kind

MakeFunc

StructOf

StructTag

StructTag.Lookup

TypeOf

Value.FieldByIndex

Value.FieldByName

Constants

[View Source](#)

```
const Ptr = Pointer
```

Ptr is the old name for the Pointer kind.

Variables

This section is empty.

Functions

func **Copy**

```
func Copy(dst, src Value) int
```

Copy copies the contents of src into dst until either dst has been filled or src has been exhausted. It returns the number of elements copied. Dst and src each must have kind Slice or Array, and dst and src must have the same element type.

As a special case, src can have kind String if the element type of dst is kind Uint8.

func DeepEqual

```
func DeepEqual(x, y any) bool
```

DeepEqual reports whether x and y are “deeply equal,” defined as follows. Two values of identical type are deeply equal if one of the following cases applies. Values of distinct types are never deeply equal.

Array values are deeply equal when their corresponding elements are deeply equal.

Struct values are deeply equal if their corresponding fields, both exported and unexported, are deeply equal.

Func values are deeply equal if both are nil; otherwise they are not deeply equal.

Interface values are deeply equal if they hold deeply equal concrete values.

Map values are deeply equal when all of the following are true: they are both nil or both non-nil, they have the same length, and either they are the same map object or their corresponding keys (matched using Go equality) map to deeply equal values.

Pointer values are deeply equal if they are equal using Go's == operator or if they point to deeply equal values.

Slice values are deeply equal when all of the following are true: they are both nil or both non-nil, they have the same length, and either they point to the same initial entry of the same underlying array (that is, &x[0] == &y[0]) or their corresponding elements (up to length) are deeply equal. Note that a non-nil empty slice and a nil slice (for example, []byte{} and []byte(nil)) are not deeply equal.

Other values - numbers, bools, strings, and channels - are deeply equal if they are equal using Go's == operator.

In general DeepEqual is a recursive relaxation of Go's == operator. However, this idea is impossible to implement without some inconsistency. Specifically, it is possible for a value to be unequal to itself, either because it is of func type (uncomparable in general) or because it is a floating-point NaN value (not equal to itself in floating-point comparison), or because it is an array, struct, or interface containing such a value. On the other hand, pointer values are always equal to themselves, even if they point at or contain such problematic values, because they compare equal using Go's == operator, and that is a sufficient condition to be deeply equal, regardless of content. DeepEqual has been defined so that the same short-cut applies to slices and maps: if x and y are the same slice or the same map, they are deeply equal regardless of content.

As DeepEqual traverses the data values it may find a cycle. The second and subsequent times that DeepEqual compares two pointer values that have been compared before, it treats the values as equal rather than examining the values to which they point. This ensures that DeepEqual terminates.

func Swapper

added in go1.8

```
func Swapper(slice any) func(i, j int)
```

Swapper returns a function that swaps the elements in the provided slice.

Swapper panics if the provided interface is not a slice.

Types

type ChanDir

```
type ChanDir int
```

ChanDir represents a channel type's direction.

```
const (
    RecvDir ChanDir      = 1 << iota // <-chan
    SendDir                               // chan<-
    BothDir = RecvDir | SendDir        // chan
)
```

func (ChanDir) String

```
func (d ChanDir) String() string
```

type Kind

```
type Kind uint
```

A Kind represents the specific kind of type that a Type represents. The zero Kind is not a valid kind.

► Example

```
const (
    Invalid Kind = iota
    Bool
    Int
    Int8
    Int16
    Int32
    Int64
    Uint
    Uint8
)
```

```
    Uint16
    Uint32
    Uint64
    Uintptr
    Float32
    Float64
    Complex64
    Complex128
    Array
    Chan
    Func
    Interface
    Map
    Pointer
    Slice
    String
    Struct
    UnsafePointer
)
```

func (Kind) String

```
func (k Kind) String() string
```

String returns the name of k.

type MapIter

added in go1.12

```
type MapIter struct {
    // contains filtered or unexported fields
}
```

A MapIter is an iterator for ranging over a map. See Value.MapRange.

func (*MapIter) Key

added in go1.12

```
func (iter *MapIter) Key() Value
```

Key returns the key of iter's current map entry.

func (*MapIter) Next

added in go1.12

```
func (iter *MapIter) Next() bool
```

Next advances the map iterator and reports whether there is another entry. It returns false when iter is exhausted; subsequent calls to Key, Value, or Next will panic.

func (*MapIter) Reset

added in go1.18

```
func (iter *MapIter) Reset(v Value)
```

Reset modifies iter to iterate over v. It panics if v's Kind is not Map and v is not the zero Value.

Reset(Value{}) causes iter to not refer to any map, which may allow the previously iterated-over map to be garbage collected.

func (*MapIter) Value

added in go1.12

```
func (iter *MapIter) Value() Value
```

Value returns the value of iter's current map entry.

type Method

```
type Method struct {
    // Name is the method name.
    Name string

    // PkgPath is the package path that qualifies a lower case (unexported)
    // method name. It is empty for upper case (exported) method names.
    // The combination of PkgPath and Name uniquely identifies a method
    // in a method set.
    // See https://golang.org/ref/spec#Uniqueness\_of\_identifiers
    PkgPath string

    Type  Type // method type
    Func  Value // func with receiver as first argument
    Index int  // index for Type.Method
}
```

Method represents a single method.

func (Method) IsExported

added in go1.17

```
func (m Method) IsExported() bool
```

IsExported reports whether the method is exported.

type SelectCase

added in go1.1

```
type SelectCase struct {
    Dir  SelectDir // direction of case
    Chan Value   // channel to use (for send or receive)
    Send Value   // value to send (for send)
}
```

A SelectCase describes a single case in a select operation. The kind of case depends on Dir, the communication direction.

If Dir is SelectDefault, the case represents a default case. Chan and Send must be zero Values.

If Dir is SelectSend, the case represents a send operation. Normally Chan's underlying value must be a channel, and Send's underlying value must be assignable to the channel's element type. As a special case, if Chan is a zero Value, then the case is ignored, and the field Send will also be ignored and may be either zero or non-zero.

If Dir is SelectRecv, the case represents a receive operation. Normally Chan's underlying value must be a channel and Send must be a zero Value. If Chan is a zero Value, then the case is ignored, but Send must still be a zero Value. When a receive operation is selected, the received Value is returned by Select.

type SelectDir

added in go1.1

```
type SelectDir int
```

A SelectDir describes the communication direction of a select case.

```
const (
    SelectSend    SelectDir // case Chan <- Send
    SelectRecv    // case <-Chan:
    SelectDefault // default
)
```

type SliceHeader

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

SliceHeader is the runtime representation of a slice. It cannot be used safely or portably and its representation may change in a later release. Moreover, the Data field is not sufficient to guarantee the data it references will not be garbage collected, so programs must keep a separate, correctly typed pointer to the underlying data.

In new code, use unsafe.Slice or unsafe.SliceData instead.

type StringHeader

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

StringHeader is the runtime representation of a string. It cannot be used safely or portably and its representation may change in a later release. Moreover, the Data field is not sufficient to guarantee the data it references will not be garbage collected, so programs must keep a separate, correctly typed pointer to the underlying data.

In new code, use `unsafe.String` or `unsafe.StringData` instead.

type `StructField`

```
type StructField struct {
    // Name is the field name.
    Name string

    // PkgPath is the package path that qualifies a lower case (unexported)
    // field name. It is empty for upper case (exported) field names.
    // See https://golang.org/ref/spec#Uniqueness\_of\_identifiers
    PkgPath string

    Type      Type      // field type
    Tag       StructTag // field tag string
    Offset     uintptr   // offset within struct, in bytes
    Index     []int     // index sequence for Type.FieldByIndex
    Anonymous bool      // is an embedded field
}
```

A `StructField` describes a single field in a struct.

func `VisibleFields`

added in go1.17

```
func VisibleFields(t Type) []StructField
```

`VisibleFields` returns all the visible fields in `t`, which must be a struct type. A field is defined as visible if it's accessible directly with a `FieldByName` call. The returned fields include fields inside anonymous struct members and unexported fields. They follow the same order found in the struct, with anonymous fields followed immediately by their promoted fields.

For each element `e` of the returned slice, the corresponding field can be retrieved from a value `v` of type `t` by calling `v.FieldByIndex(e.Index)`.

func (StructField) `IsExported`

added in go1.17

```
func (f StructField) IsExported() bool
```

`IsExported` reports whether the field is exported.

type `StructTag`

```
type StructTag string
```

A `StructTag` is the tag string in a struct field.

By convention, tag strings are a concatenation of optionally space-separated `key:"value"` pairs. Each key is a non-empty string consisting of non-control characters other than space (U+0020 ' '), quote (U+0022 ""), and colon (U+003A ':'). Each value is quoted using U+0022 "" characters and Go string literal syntax.

► Example

func (StructTag) Get

```
func (tag StructTag) Get(key string) string
```

Get returns the value associated with key in the tag string. If there is no such key in the tag, Get returns the empty string. If the tag does not have the conventional format, the value returned by Get is unspecified. To determine whether a tag is explicitly set to the empty string, use Lookup.

func (StructTag) Lookup

added in go1.7

```
func (tag StructTag) Lookup(key string) (value string, ok bool)
```

Lookup returns the value associated with key in the tag string. If the key is present in the tag the value (which may be empty) is returned. Otherwise the returned value will be the empty string. The ok return value reports whether the value was explicitly set in the tag string. If the tag does not have the conventional format, the value returned by Lookup is unspecified.

► Example

type Type

```
type Type interface {

    // Align returns the alignment in bytes of a value of
    // this type when allocated in memory.
    Align() int

    // FieldAlign returns the alignment in bytes of a value of
    // this type when used as a field in a struct.
    FieldAlign() int

    // Method returns the i'th method in the type's method set.
    // It panics if i is not in the range [0, NumMethod()).
    //
    // For a non-interface type T or *T, the returned Method's Type and Func
    // fields describe a function whose first argument is the receiver,
    // and only exported methods are accessible.
    //
    // For an interface type, the returned Method's Type field gives the
    // method signature, without a receiver, and the Func field is nil.
    //
    // Methods are sorted in lexicographic order.
    Method(int) Method

    // MethodByName returns the method with that name in the type's
    // method set and a boolean indicating if the method was found.
```

```
//  
// For a non-interface type T or *T, the returned Method's Type and Func  
// fields describe a function whose first argument is the receiver.  
//  
// For an interface type, the returned Method's Type field gives the  
// method signature, without a receiver, and the Func field is nil.  
MethodByName(string) (Method, bool)  
  
// NumMethod returns the number of methods accessible using Method.  
//  
// For a non-interface type, it returns the number of exported methods.  
//  
// For an interface type, it returns the number of exported and unexported methods.  
NumMethod() int  
  
// Name returns the type's name within its package for a defined type.  
// For other (non-defined) types it returns the empty string.  
Name() string  
  
// PkgPath returns a defined type's package path, that is, the import path  
// that uniquely identifies the package, such as "encoding/base64".  
// If the type was predeclared (string, error) or not defined (*T, struct{},  
// []int, or A where A is an alias for a non-defined type), the package path  
// will be the empty string.  
PkgPath() string  
  
// Size returns the number of bytes needed to store  
// a value of the given type; it is analogous to unsafe.Sizeof.  
Size() uintptr  
  
// String returns a string representation of the type.  
// The string representation may use shortened package names  
// (e.g., base64 instead of "encoding/base64") and is not  
// guaranteed to be unique among types. To test for type identity,  
// compare the Types directly.  
String() string  
  
// Kind returns the specific kind of this type.  
Kind() Kind  
  
// Implements reports whether the type implements the interface type u.  
Implements(u Type) bool  
  
// AssignableTo reports whether a value of the type is assignable to type u.  
AssignableTo(u Type) bool  
  
// ConvertibleTo reports whether a value of the type is convertible to type u.  
// Even if ConvertibleTo returns true, the conversion may still panic.  
// For example, a slice of type []T is convertible to *[N]T,  
// but the conversion will panic if its length is less than N.  
ConvertibleTo(u Type) bool  
  
// Comparable reports whether values of this type are comparable.
```

```

// Even if Comparable returns true, the comparison may still panic.
// For example, values of interface type are comparable,
// but the comparison will panic if their dynamic type is not comparable.
Comparable() bool

// Bits returns the size of the type in bits.
// It panics if the type's Kind is not one of the
// sized or unsized Int, Uint, Float, or Complex kinds.
Bits() int

// ChanDir returns a channel type's direction.
// It panics if the type's Kind is not Chan.
ChanDir() ChanDir

// IsVariadic reports whether a function type's final input parameter
// is a "... " parameter. If so, t.In(t.NumIn() - 1) returns the parameter's
// implicit actual type []T.
//
// For concreteness, if t represents func(x int, y ... float64), then
//
//   t.NumIn() == 2
//   t.In(0) is the reflect.Type for "int"
//   t.In(1) is the reflect.Type for "[]float64"
//   t.IsVariadic() == true
//
// IsVariadic panics if the type's Kind is not Func.
IsVariadic() bool

// Elem returns a type's element type.
// It panics if the type's Kind is not Array, Chan, Map, Pointer, or Slice.
Elem() Type

// Field returns a struct type's i'th field.
// It panics if the type's Kind is not Struct.
// It panics if i is not in the range [0, NumField()).
Field(i int) StructField

// FieldByIndex returns the nested field corresponding
// to the index sequence. It is equivalent to calling Field
// successively for each index i.
// It panics if the type's Kind is not Struct.
FieldByIndex(index []int) StructField

// FieldByName returns the struct field with the given name
// and a boolean indicating if the field was found.
FieldByName(name string) (StructField, bool)

// FieldByNameFunc returns the struct field with a name
// that satisfies the match function and a boolean indicating if
// the field was found.
//
// FieldByNameFunc considers the fields in the struct itself
// and then the fields in any embedded structs, in breadth first order,

```

```

// stopping at the shallowest nesting depth containing one or more
// fields satisfying the match function. If multiple fields at that depth
// satisfy the match function, they cancel each other
// and FieldByNameFunc returns no match.
// This behavior mirrors Go's handling of name lookup in
// structs containing embedded fields.
FieldByNameFunc(match func(string) bool) (StructField, bool)

// In returns the type of a function type's i'th input parameter.
// It panics if the type's Kind is not Func.
// It panics if i is not in the range [0, NumIn()).
In(i int) Type

// Key returns a map type's key type.
// It panics if the type's Kind is not Map.
Key() Type

// Len returns an array type's length.
// It panics if the type's Kind is not Array.
Len() int

// 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

// Out returns the type of a function type's i'th output parameter.
// It panics if the type's Kind is not Func.
// It panics if i is not in the range [0, NumOut()).
Out(i int) Type
// contains filtered or unexported methods
}

```

Type is the representation of a Go type.

Not all methods apply to all kinds of types. Restrictions, if any, are noted in the documentation for each method. Use the Kind method to find out the kind of type before calling kind-specific methods. Calling a method inappropriate to the kind of type causes a run-time panic.

Type values are comparable, such as with the == operator, so they can be used as map keys. Two Type values are equal if they represent identical types.

func ArrayOf

added in go1.5

```
func ArrayOf(length int, elem Type) Type
```

ArrayOf returns the array type with the given length and element type. For example, if t represents int, ArrayOf(5, t) represents [5]int.

If the resulting type would be larger than the available address space, ArrayOf panics.

func ChanOf

added in go1.1

```
func ChanOf(dir ChanDir, t Type) Type
```

ChanOf returns the channel type with the given direction and element type. For example, if t represents int, ChanOf(RecvDir, t) represents <-chan int.

The gc runtime imposes a limit of 64 kB on channel element types. If t's size is equal to or exceeds this limit, ChanOf panics.

func FuncOf

added in go1.5

```
func FuncOf(in, out []Type, variadic bool) Type
```

FuncOf returns the function type with the given argument and result types. For example if k represents int and e represents string, FuncOf([]Type{k}, []Type{e}, false) represents func(int) string.

The variadic argument controls whether the function is variadic. FuncOf panics if the in[len(in)-1] does not represent a slice and variadic is true.

func MapOf

added in go1.1

```
func MapOf(key, elem Type) Type
```

MapOf returns the map type with the given key and element types. For example, if k represents int and e represents string, MapOf(k, e) represents map[int]string.

If the key type is not a valid map key type (that is, if it does not implement Go's == operator), MapOf panics.

func PointerTo

added in go1.18

```
func PointerTo(t Type) Type
```

PointerTo returns the pointer type with element t. For example, if t represents type Foo, PointerTo(t) represents *Foo.

func PtrTo

```
func PtrTo(t Type) Type
```

PtrTo returns the pointer type with element t. For example, if t represents type Foo, PtrTo(t) represents *Foo.

PtrTo is the old spelling of PointerTo. The two functions behave identically.

func SliceOf

added in go1.1

```
func SliceOf(t Type) Type
```

SliceOf returns the slice type with element type t. For example, if t represents int, SliceOf(t) represents []int.

func StructOf

added in go1.7

```
func StructOf(fields []StructField) Type
```

StructOf returns the struct type containing fields. The Offset and Index fields are ignored and computed as they would be by the compiler.

StructOf currently does not generate wrapper methods for embedded fields and panics if passed unexported StructFields. These limitations may be lifted in a future version.

► [Example](#)

func TypeOf

```
func TypeOf(i any) Type
```

TypeOf returns the reflection Type that represents the dynamic type of i. If i is a nil interface value, TypeOf returns nil.

► [Example](#)

type Value

```
type Value struct {  
    // contains filtered or unexported fields  
}
```

Value is the reflection interface to a Go value.

Not all methods apply to all kinds of values. Restrictions, if any, are noted in the documentation for each method. Use the Kind method to find out the kind of value before calling kind-specific methods. Calling a method inappropriate to the kind of type causes a run time panic.

The zero Value represents no value. Its IsValid method returns false, its Kind method returns Invalid, its String method returns "<invalid Value>", and all other methods panic. Most functions and methods never return an invalid value. If one does, its documentation states the conditions explicitly.

A Value can be used concurrently by multiple goroutines provided that the underlying Go value can be used concurrently for the equivalent direct operations.

To compare two Values, compare the results of the Interface method. Using == on two Values does not compare the underlying values they represent.

func Append

```
func Append(s Value, x ...Value) Value
```

Append appends the values x to a slice s and returns the resulting slice. As in Go, each x's value must be assignable to the slice's element type.

func AppendSlice

```
func AppendSlice(s, t Value) Value
```

AppendSlice appends a slice t to a slice s and returns the resulting slice. The slices s and t must have the same element type.

func Indirect

```
func Indirect(v Value) Value
```

Indirect returns the value that v points to. If v is a nil pointer, Indirect returns a zero Value. If v is not a pointer, Indirect returns v.

func MakeChan

```
func MakeChan(typ Type, buffer int) Value
```

MakeChan creates a new channel with the specified type and buffer size.

func MakeFunc

added in go1.1

```
func MakeFunc(typ Type, fn func(args []Value) (results []Value)) Value
```

MakeFunc returns a new function of the given Type that wraps the function fn. When called, that new function does the following:

- converts its arguments to a slice of Values.
- runs results := fn(args).
- returns the results as a slice of Values, one per formal result.

The implementation fn can assume that the argument Value slice has the number and type of arguments given by typ. If typ describes a variadic function, the final Value is itself a slice representing the variadic arguments, as in the body of a variadic function. The result Value slice returned by fn must have the number and type of results given by typ.

The `Value.Call` method allows the caller to invoke a typed function in terms of `Values`; in contrast, `MakeFunc` allows the caller to implement a typed function in terms of `Values`.

The Examples section of the documentation includes an illustration of how to use `MakeFunc` to build a swap function for different types.

► [Example](#)

func `MakeMap`

```
func MakeMap(typ Type) Value
```

`MakeMap` creates a new map with the specified type.

func `MakeMapWithSize`

added in go1.9

```
func MakeMapWithSize(typ Type, n int) Value
```

`MakeMapWithSize` creates a new map with the specified type and initial space for approximately `n` elements.

func `MakeSlice`

```
func MakeSlice(typ Type, len, cap int) Value
```

`MakeSlice` creates a new zero-initialized slice value for the specified slice type, length, and capacity.

func `New`

```
func New(typ Type) Value
```

`New` returns a `Value` representing a pointer to a new zero value for the specified type. That is, the returned `Value`'s `Type` is `PtrTo(typ)`.

func `NewAt`

```
func NewAt(typ Type, p unsafe.Pointer) Value
```

`NewAt` returns a `Value` representing a pointer to a value of the specified type, using `p` as that pointer.

func `Select`

added in go1.1

```
func Select(cases []SelectCase) (chosen int, recv Value, recvOK bool)
```

`Select` executes a select operation described by the list of cases. Like the Go select statement, it blocks until at least one of the cases can proceed, makes a uniform pseudo-random choice, and then executes that case. It returns the index of the chosen case and, if that case was a receive operation, the value

received and a boolean indicating whether the value corresponds to a send on the channel (as opposed to a zero value received because the channel is closed). Select supports a maximum of 65536 cases.

func ValueOf

```
func ValueOf(i any) Value
```

ValueOf returns a new Value initialized to the concrete value stored in the interface i. ValueOf(nil) returns the zero Value.

func Zero

```
func Zero(typ Type) Value
```

Zero returns a Value representing the zero value for the specified type. The result is different from the zero value of the Value struct, which represents no value at all. For example, Zero(ZeroOf(42)) returns a Value with Kind Int and value 0. The returned value is neither addressable nor settable.

func (Value) Addr

```
func (v Value) Addr() Value
```

Addr returns a pointer value representing the address of v. It panics if CanAddr() returns false. Addr is typically used to obtain a pointer to a struct field or slice element in order to call a method that requires a pointer receiver.

func (Value) Bool

```
func (v Value) Bool() bool
```

Bool returns v's underlying value. It panics if v's kind is not Bool.

func (Value) Bytes

```
func (v Value) Bytes() []byte
```

Bytes returns v's underlying value. It panics if v's underlying value is not a slice of bytes or an addressable array of bytes.

func (Value) Call

```
func (v Value) Call(in []Value) []Value
```

Call calls the function v with the input arguments in. For example, if len(in) == 3, v.Call(in) represents the Go call v(in[0], in[1], in[2]). Call panics if v's Kind is not Func. It returns the output results as Values. As in Go, each input argument must be assignable to the type of the function's corresponding input parameter. If v is a variadic function, Call creates the variadic slice parameter itself, copying in the corresponding values.

func (Value) CallSlice

```
func (v Value) CallSlice(in []Value) []Value
```

CallSlice calls the variadic function `v` with the input arguments `in`, assigning the slice `in[len(in)-1]` to `v`'s final variadic argument. For example, if `len(in) == 3`, `v.CallSlice(in)` represents the Go call `v(in[0], in[1], in[2]...)`. CallSlice panics if `v`'s Kind is not Func or if `v` is not variadic. It returns the output results as Values. As in Go, each input argument must be assignable to the type of the function's corresponding input parameter.

func (Value) CanAddr

```
func (v Value) CanAddr() bool
```

CanAddr reports whether the value's address can be obtained with Addr. Such values are called addressable. A value is addressable if it is an element of a slice, an element of an addressable array, a field of an addressable struct, or the result of dereferencing a pointer. If CanAddr returns false, calling Addr will panic.

func (Value) CanComplex

added in go1.18

```
func (v Value) CanComplex() bool
```

CanComplex reports whether Complex can be used without panicking.

func (Value) CanConvert

added in go1.17

```
func (v Value) CanConvert(t Type) bool
```

CanConvert reports whether the value `v` can be converted to type `t`. If `v.CanConvert(t)` returns true then `v.Convert(t)` will not panic.

func (Value) CanFloat

added in go1.18

```
func (v Value) CanFloat() bool
```

CanFloat reports whether Float can be used without panicking.

func (Value) CanInt

added in go1.18

```
func (v Value) CanInt() bool
```

CanInt reports whether Int can be used without panicking.

func (Value) CanInterface

```
func (v Value) CanInterface() bool
```

CanInterface reports whether Interface can be used without panicking.

func (Value) CanSet

```
func (v Value) CanSet() bool
```

CanSet reports whether the value of v can be changed. A Value can be changed only if it is addressable and was not obtained by the use of unexported struct fields. If CanSet returns false, calling Set or any type-specific setter (e.g., SetBool, SetInt) will panic.

func (Value) CanUint

added in go1.18

```
func (v Value) CanUint() bool
```

CanUint reports whether Uint can be used without panicking.

func (Value) Cap

```
func (v Value) Cap() int
```

Cap returns v's capacity. It panics if v's Kind is not Array, Chan, Slice or pointer to Array.

func (Value) Close

```
func (v Value) Close()
```

Close closes the channel v. It panics if v's Kind is not Chan.

func (Value) Comparable

added in go1.20

```
func (v Value) Comparable() bool
```

Comparable reports whether the value v is comparable. If the type of v is an interface, this checks the dynamic type. If this reports true then v.Interface() == x will not panic for any x, nor will v.Equal(u) for any Value u.

func (Value) Complex

```
func (v Value) Complex() complex128
```

Complex returns v's underlying value, as a complex128. It panics if v's Kind is not Complex64 or Complex128

func (Value) Convert

added in go1.1

```
func (v Value) Convert(t Type) Value
```

Convert returns the value `v` converted to type `t`. If the usual Go conversion rules do not allow conversion of the value `v` to type `t`, or if converting `v` to type `t` panics, Convert panics.

func (Value) Elem

```
func (v Value) Elem() Value
```

Elem returns the value that the interface `v` contains or that the pointer `v` points to. It panics if `v`'s Kind is not Interface or Pointer. It returns the zero Value if `v` is nil.

func (Value) Equal

added in go1.20

```
func (v Value) Equal(u Value) bool
```

Equal reports true if `v` is equal to `u`. For two invalid values, Equal will report true. For an interface value, Equal will compare the value within the interface. Otherwise, If the values have different types, Equal will report false. Otherwise, for arrays and structs Equal will compare each element in order, and report false if it finds non-equal elements. During all comparisons, if values of the same type are compared, and the type is not comparable, Equal will panic.

func (Value) Field

```
func (v Value) Field(i int) Value
```

Field returns the `i`'th field of the struct `v`. It panics if `v`'s Kind is not Struct or `i` is out of range.

func (Value) FieldByIndex

```
func (v Value) FieldByIndex(index []int) Value
```

FieldByIndex returns the nested field corresponding to `index`. It panics if evaluation requires stepping through a nil pointer or a field that is not a struct.

► [Example](#)

func (Value) FieldByIndexErr

added in go1.18

```
func (v Value) FieldByIndexErr(index []int) (Value, error)
```

FieldByIndexErr returns the nested field corresponding to `index`. It returns an error if evaluation requires stepping through a nil pointer, but panics if it must step through a field that is not a struct.

func (Value) FieldByName

```
func (v Value) FieldByName(name string) Value
```

FieldByName returns the struct field with the given name. It returns the zero Value if no field was found. It panics if v's Kind is not struct.

► Example

func (Value) FieldByNameFunc

```
func (v Value) FieldByNameFunc(match func(string) bool) Value
```

FieldByNameFunc returns the struct field with a name that satisfies the match function. It panics if v's Kind is not struct. It returns the zero Value if no field was found.

func (Value) Float

```
func (v Value) Float() float64
```

Float returns v's underlying value, as a float64. It panics if v's Kind is not Float32 or Float64

func (Value) Grow

added in go1.20

```
func (v Value) Grow(n int)
```

Grow increases the slice's capacity, if necessary, to guarantee space for another n elements. After Grow(n), at least n elements can be appended to the slice without another allocation.

It panics if v's Kind is not a Slice or if n is negative or too large to allocate the memory.

func (Value) Index

```
func (v Value) Index(i int) Value
```

Index returns v's i'th element. It panics if v's Kind is not Array, Slice, or String or i is out of range.

func (Value) Int

```
func (v Value) Int() int64
```

Int returns v's underlying value, as an int64. It panics if v's Kind is not Int, Int8, Int16, Int32, or Int64.

func (Value) Interface

```
func (v Value) Interface() (i any)
```

Interface returns v's current value as an interface{}. It is equivalent to:

```
var i interface{} = (v's underlying value)
```

It panics if the Value was obtained by accessing unexported struct fields.

func (Value) InterfaceData DEPRECATED Show

func (Value) IsNil

```
func (v Value) IsNil() bool
```

IsNil reports whether its argument *v* is nil. The argument must be a chan, func, interface, map, pointer, or slice value; if it is not, IsNil panics. Note that IsNil is not always equivalent to a regular comparison with nil in Go. For example, if *v* was created by calling ValueOf with an uninitialized interface variable *i*, *i*==nil will be true but *v*.IsNil will panic as *v* will be the zero Value.

func (Value) IsValid

```
func (v Value) IsValid() bool
```

IsValid reports whether *v* represents a value. It returns false if *v* is the zero Value. If IsValid returns false, all other methods except String panic. Most functions and methods never return an invalid Value. If one does, its documentation states the conditions explicitly.

func (Value) IsZero

added in go1.13

```
func (v Value) IsZero() bool
```

IsZero reports whether *v* is the zero value for its type. It panics if the argument is invalid.

func (Value) Kind

```
func (v Value) Kind() Kind
```

Kind returns *v*'s Kind. If *v* is the zero Value (IsValid returns false), Kind returns Invalid.

func (Value) Len

```
func (v Value) Len() int
```

Len returns *v*'s length. It panics if *v*'s Kind is not Array, Chan, Map, Slice, String, or pointer to Array.

func (Value) MapIndex

```
func (v Value) MapIndex(key Value) Value
```

MapIndex returns the value associated with *key* in the map *v*. It panics if *v*'s Kind is not Map. It returns the zero Value if *key* is not found in the map or if *v* represents a nil map. As in Go, the *key*'s value must be assignable to the map's key type.

func (Value) MapKeys


```
func (v Value) MapKeys() []Value
```

MapKeys returns a slice containing all the keys present in the map, in unspecified order. It panics if v's Kind is not Map. It returns an empty slice if v represents a nil map.

func (Value) MapRange

added in go1.12

```
func (v Value) MapRange() *MapIter
```

MapRange returns a range iterator for a map. It panics if v's Kind is not Map.

Call Next to advance the iterator, and Key/Value to access each entry. Next returns false when the iterator is exhausted. MapRange follows the same iteration semantics as a range statement.

Example:

```
iter := reflect.ValueOf(m).MapRange()
for iter.Next() {
    k := iter.Key()
    v := iter.Value()
    ...
}
```

func (Value) Method

```
func (v Value) Method(i int) Value
```

Method returns a function value corresponding to v's i'th method. The arguments to a Call on the returned function should not include a receiver; the returned function will always use v as the receiver. Method panics if i is out of range or if v is a nil interface value.

func (Value) MethodByName

```
func (v Value) MethodByName(name string) Value
```

MethodByName returns a function value corresponding to the method of v with the given name. The arguments to a Call on the returned function should not include a receiver; the returned function will always use v as the receiver. It returns the zero Value if no method was found.

func (Value) NumField

```
func (v Value) NumField() int
```

NumField returns the number of fields in the struct v. It panics if v's Kind is not Struct.

func (Value) NumMethod

```
func (v Value) NumMethod() int
```

NumMethod returns the number of methods in the value's method set.

For a non-interface type, it returns the number of exported methods.

For an interface type, it returns the number of exported and unexported methods.

func (Value) OverflowComplex

```
func (v Value) OverflowComplex(x complex128) bool
```

OverflowComplex reports whether the complex128 x cannot be represented by v's type. It panics if v's Kind is not Complex64 or Complex128.

func (Value) OverflowFloat

```
func (v Value) OverflowFloat(x float64) bool
```

OverflowFloat reports whether the float64 x cannot be represented by v's type. It panics if v's Kind is not Float32 or Float64.

func (Value) OverflowInt

```
func (v Value) OverflowInt(x int64) bool
```

OverflowInt reports whether the int64 x cannot be represented by v's type. It panics if v's Kind is not Int, Int8, Int16, Int32, or Int64.

func (Value) OverflowUint

```
func (v Value) OverflowUint(x uint64) bool
```

OverflowUint reports whether the uint64 x cannot be represented by v's type. It panics if v's Kind is not Uint, Uintptr, Uint8, Uint16, Uint32, or Uint64.

func (Value) Pointer

```
func (v Value) Pointer() uintptr
```

Pointer returns v's value as a uintptr. It panics if v's Kind is not Chan, Func, Map, Pointer, Slice, or UnsafePointer.

If v's Kind is Func, the returned pointer is an underlying code pointer, but not necessarily enough to identify a single function uniquely. The only guarantee is that the result is zero if and only if v is a nil func Value.

If v's Kind is Slice, the returned pointer is to the first element of the slice. If the slice is nil the returned value is 0. If the slice is empty but non-nil the return value is non-zero.

It's preferred to use uintptr(Value.UnsafePointer()) to get the equivalent result.

func (Value) Recv

```
func (v Value) Recv() (x Value, ok bool)
```

Recv receives and returns a value from the channel v. It panics if v's Kind is not Chan. The receive blocks until a value is ready. The boolean value ok is true if the value x corresponds to a send on the channel, false if it is a zero value received because the channel is closed.

func (Value) Send

```
func (v Value) Send(x Value)
```

Send sends x on the channel v. It panics if v's kind is not Chan or if x's type is not the same type as v's element type. As in Go, x's value must be assignable to the channel's element type.

func (Value) Set

```
func (v Value) Set(x Value)
```

Set assigns x to the value v. It panics if CanSet returns false. As in Go, x's value must be assignable to v's type and must not be derived from an unexported field.

func (Value) SetBool

```
func (v Value) SetBool(x bool)
```

SetBool sets v's underlying value. It panics if v's Kind is not Bool or if CanSet() is false.

func (Value) SetBytes

```
func (v Value) SetBytes(x []byte)
```

SetBytes sets v's underlying value. It panics if v's underlying value is not a slice of bytes.

func (Value) SetCap

added in go1.2

```
func (v Value) SetCap(n int)
```

SetCap sets v's capacity to n. It panics if v's Kind is not Slice or if n is smaller than the length or greater than the capacity of the slice.

func (Value) SetComplex

```
func (v Value) SetComplex(x complex128)
```

SetComplex sets v's underlying value to x. It panics if v's Kind is not Complex64 or Complex128, or if CanSet() is false.

func (Value) SetFloat

```
func (v Value) SetFloat(x float64)
```

SetFloat sets v's underlying value to x. It panics if v's Kind is not Float32 or Float64, or if CanSet() is false.

func (Value) SetInt

```
func (v Value) SetInt(x int64)
```

SetInt sets v's underlying value to x. It panics if v's Kind is not Int, Int8, Int16, Int32, or Int64, or if CanSet() is false.

func (Value) SetIterKey

added in go1.18

```
func (v Value) SetIterKey(iter *MapIter)
```

SetIterKey assigns to v the key of iter's current map entry. It is equivalent to v.Set(iter.Key()), but it avoids allocating a new Value. As in Go, the key must be assignable to v's type and must not be derived from an unexported field.

func (Value) SetIterValue

added in go1.18

```
func (v Value) SetIterValue(iter *MapIter)
```

SetIterValue assigns to v the value of iter's current map entry. It is equivalent to v.Set(iter.Value()), but it avoids allocating a new Value. As in Go, the value must be assignable to v's type and must not be derived from an unexported field.

func (Value) SetLen

```
func (v Value) SetLen(n int)
```

SetLen sets v's length to n. It panics if v's Kind is not Slice or if n is negative or greater than the capacity of the slice.

func (Value) SetMapIndex

```
func (v Value) SetMapIndex(key, elem Value)
```

SetMapIndex sets the element associated with key in the map v to elem. It panics if v's Kind is not Map. If elem is the zero Value, SetMapIndex deletes the key from the map. Otherwise if v holds a nil map, SetMapIndex will panic. As in Go, key's elem must be assignable to the map's key type, and elem's value must be assignable to the map's elem type.

func (Value) SetPointer

```
func (v Value) SetPointer(x unsafe.Pointer)
```

SetPointer sets the `unsafe.Pointer` value `v` to `x`. It panics if `v`'s Kind is not `UnsafePointer`.

func (Value) SetString

```
func (v Value) SetString(x string)
```

SetString sets `v`'s underlying value to `x`. It panics if `v`'s Kind is not `String` or if `CanSet()` is false.

func (Value) SetUint

```
func (v Value) SetUint(x uint64)
```

SetUint sets `v`'s underlying value to `x`. It panics if `v`'s Kind is not `Uint`, `Uintptr`, `Uint8`, `Uint16`, `Uint32`, or `Uint64`, or if `CanSet()` is false.

func (Value) SetZero

added in go1.20

```
func (v Value) SetZero()
```

SetZero sets `v` to be the zero value of `v`'s type. It panics if `CanSet` returns false.

func (Value) Slice

```
func (v Value) Slice(i, j int) Value
```

Slice returns `v[i:j]`. It panics if `v`'s Kind is not `Array`, `Slice` or `String`, or if `v` is an unaddressable array, or if the indexes are out of bounds.

func (Value) Slice3

added in go1.2

```
func (v Value) Slice3(i, j, k int) Value
```

Slice3 is the 3-index form of the slice operation: it returns `v[i:j:k]`. It panics if `v`'s Kind is not `Array` or `Slice`, or if `v` is an unaddressable array, or if the indexes are out of bounds.

func (Value) String

```
func (v Value) String() string
```

String returns the string `v`'s underlying value, as a string. String is a special case because of Go's String method convention. Unlike the other getters, it does not panic if `v`'s Kind is not `String`. Instead, it returns a string of the form "<T value>" where `T` is `v`'s type. The `fmt` package treats Values specially. It does not call their String method implicitly but instead prints the concrete values they hold.

func (Value) TryRecv

```
func (v Value) TryRecv() (x Value, ok bool)
```

TryRecv attempts to receive a value from the channel v but will not block. It panics if v's Kind is not Chan. If the receive delivers a value, x is the transferred value and ok is true. If the receive cannot finish without blocking, x is the zero Value and ok is false. If the channel is closed, x is the zero value for the channel's element type and ok is false.

func (Value) TrySend

```
func (v Value) TrySend(x Value) bool
```

TrySend attempts to send x on the channel v but will not block. It panics if v's Kind is not Chan. It reports whether the value was sent. As in Go, x's value must be assignable to the channel's element type.

func (Value) Type

```
func (v Value) Type() Type
```

Type returns v's type.

func (Value) Uint

```
func (v Value) Uint() uint64
```

Uint returns v's underlying value, as a uint64. It panics if v's Kind is not Uint, Uintptr, Uint8, Uint16, Uint32, or Uint64.

func (Value) UnsafeAddr

```
func (v Value) UnsafeAddr() uintptr
```

UnsafeAddr returns a pointer to v's data, as a uintptr. It panics if v is not addressable.

It's preferred to use uintptr(Value.Addr().UnsafePointer()) to get the equivalent result.

func (Value) UnsafePointer

added in go1.18

```
func (v Value) UnsafePointer() unsafe.Pointer
```

UnsafePointer returns v's value as a [unsafe.Pointer](#). It panics if v's Kind is not Chan, Func, Map, Pointer, Slice, or UnsafePointer.

If v's Kind is Func, the returned pointer is an underlying code pointer, but not necessarily enough to identify a single function uniquely. The only guarantee is that the result is zero if and only if v is a nil func Value.

If v's Kind is Slice, the returned pointer is to the first element of the slice. If the slice is nil the returned value is nil. If the slice is empty but non-nil the return value is non-nil.

type **ValueError**

```
type ValueError struct {  
    Method string  
    Kind   Kind  
}
```

A **ValueError** occurs when a **Value** method is invoked on a **Value** that does not support it. Such cases are documented in the description of each method.

func (***ValueError**) **Error**

```
func (e *ValueError) Error() string
```

Notes

Bugs

- **FieldByName** and related functions consider struct field names to be equal if the names are equal, even if they are unexported names originating in different packages. The practical effect of this is that the result of `t.FieldByName("x")` is not well defined if the struct type `t` contains multiple fields named `x` (embedded from different packages). **FieldByName** may return one of the fields named `x` or may report that there are none. See <https://golang.org/issue/4876> for more details.

Source Files

[View all](#) 

abi.go	makefunc.go	value.go
deepequal.go	swapper.go	visiblefields.go
float32reg_generic.go	type.go	

Directories

[Expand all](#)

► [internal](#)

Why Go

[Use Cases](#)

[Case Studies](#)

Get Started

[Playground](#)

[Tour](#)

[Stack Overflow](#)

[Help](#)

Packages

[Standard Library](#)

[About Go Packages](#)

About

[Download](#)

[Blog](#)

[Issue Tracker](#)

[Release Notes](#)

[Brand Guidelines](#)

[Code of Conduct](#)

Connect

[Twitter](#)

[GitHub](#)

[Slack](#)

[r/golang](#)

[Meetup](#)

[Golang Weekly](#)

[Copyright](#)

[Terms of Service](#)

[Privacy Policy](#)

[Report an Issue](#)



Google