Golang Struct

‹ Previous Next ›

Golang Struct

Learn to create user-defined types by combining with other types.

Introduction

A **struct** (short for "structure") is a collection of data fields with declared data types. Golang has the ability to declare and create own data types by combining one or more types, including both built-in and user-defined types. Each data field in a struct is declared with a known type, which could be a built-in type or another user-defined type.

Structs are the only way to create concrete user-defined types in Golang. Struct types are declared by composing a fixed set of unique fields. Structs can improve modularity and allow to create and pass complex data structures around the system. You can also consider Structs as a template for creating a data record, like an employee record or an e-commerce product.

The declaration starts with the keyword **type**, then a name for the new struct, and finally the keyword **struct**. Within the curly brackets, a series of data fields are specified with a name and a type.

Example

```
type identifier struct{
  field1 data_type
```

```
  field2 data_type
  field3 data_type
}
```

Declaration of a Struct Type

A struct type `rectangle` is declared that has three data fields of different data-types. Here, struct used without instantiate a new instance of that type.

Example

```
package main

import "fmt"

type rectangle struct {
    length  float64
    breadth float64
    color   string
}

func main() {
    fmt.Println(rectangle{10.5, 25.10, "red"})
}
```

The `rectangle` struct and its fields are not exported to other packages because identifiers are started with an lowercase letter. In Golang, identifiers are exported to other packages if the name starts with an uppercase letter, otherwise the accessibility will be limited within the package only.

Creating Instances of Struct Types

The `var` keyword initializes a variable `rect`. Using **dot** notation, values are assigned to the struct fields.

Example

```
package main

import "fmt"

type rectangle struct {
    length  int
    breadth int
    color   string

    geometry struct {
        area      int
        perimeter int
    }
}

func main() {
    var rect rectangle // dot notation
    rect.length = 10
    rect.breadth = 20
    rect.color = "Green"

    rect.geometry.area = rect.length * rect.breadth
    rect.geometry.perimeter = 2 * (rect.length + rect.breadth)

    fmt.Println(rect)
    fmt.Println("Area:\t", rect.geometry.area)
    fmt.Println("Perimeter:", rect.geometry.perimeter)
}
```

The struct is printed to the terminal, showing the values have been assigned.

Creating a Struct Instance Using a Struct Literal

Creates an instance of `rectangle` struct by using a struct literal and assigning values to the fields of the struct.

Example

```
package main

import "fmt"

type rectangle struct {
    length  int
    breadth int
    color   string
}

func main() {
    var rect1 = rectangle{10, 20, "Green"}
    fmt.Println(rect1)

    var rect2 = rectangle{length: 10, color: "Green"} // breadth value skipp
    fmt.Println(rect2)

    rect3 := rectangle{10, 20, "Green"}
    fmt.Println(rect3)

    rect4 := rectangle{length: 10, breadth: 20, color: "Green"}
    fmt.Println(rect4)

    rect5 := rectangle{breadth: 20, color: "Green"} // length value skipped
    fmt.Println(rect5)
}
```

Struct Instantiation Using new Keyword

An instance of a **struct** can also be created with the `new` keyword. It is then possible to assign data values to the data fields using dot notation.

Example

```
package main

import "fmt"

type rectangle struct {
    length  int
    breadth int
    color   string
}

func main() {
    rect1 := new(rectangle) // rect1 is a pointer to an instance of rectangl
    rect1.length = 10
    rect1.breadth = 20
    rect1.color = "Green"
    fmt.Println(rect1)

    var rect2 = new(rectangle) // rect2 is an instance of rectangle
    rect2.length = 10
    rect2.color = "Red"
    fmt.Println(rect2)
}
```

Two instances of the `rectangle` struct are instantiated, **rect1** points to the address of the instantiated struct and **rect2** is the name of a struct it represents.

Struct Instantiation Using Pointer Address Operator

Creates an instance of `rectangle` struct by using a pointer address operator is denoted by **&** symbol.

Example

```
package main

import "fmt"

type rectangle struct {
    length  int
    breadth int
    color   string
}

func main() {
    var rect1 = &rectangle{10, 20, "Green"} // Can't skip any value
    fmt.Println(rect1)

    var rect2 = &rectangle{}
    rect2.length = 10
    rect2.color = "Red"
    fmt.Println(rect2) // breadth skipped

    var rect3 = &rectangle{}
    (*rect3).breadth = 10
    (*rect3).color = "Blue"
    fmt.Println(rect3) // length skipped
}
```

Nested Struct Type

Struct can be nested by creating a Struct type using other Struct types as the type for the fields of Struct. Nesting one struct within another can be a useful way to model more complex structures.

Example

```
package main

import "fmt"
```

```
type Salary struct {
    Basic, HRA, TA float64
}

type Employee struct {
    FirstName, LastName, Email string
    Age                       int
    MonthlySalary             []Salary
}

func main() {
    e := Employee{
        FirstName: "Mark",
        LastName:  "Jones",
        Email:     "mark@gmail.com",
        Age:       25,
        MonthlySalary: []Salary{
            Salary{
                Basic: 15000.00,
                HRA:   5000.00,
                TA:    2000.00,
            },
            Salary{
                Basic: 16000.00,
                HRA:   5000.00,
                TA:    2100.00,
            },
            Salary{
                Basic: 17000.00,
                HRA:   5000.00,
                TA:    2200.00,
            },
        },
    }
    fmt.Println(e.FirstName, e.LastName)
    fmt.Println(e.Age)
    fmt.Println(e.Email)
    fmt.Println(e.MonthlySalary[0])
    fmt.Println(e.MonthlySalary[1])
    fmt.Println(e.MonthlySalary[2])
}
```

Use Field Tags in the Definition of Struct Type

During the definition of a `struct` type, optional `string` values may be added to each field declaration.

Example

```
package main

import (
    "fmt"
    "encoding/json"
)

type Employee struct {
    FirstName  string `json:"firstname"`
    LastName   string `json:"lastname"`
    City string `json:"city"`
}

func main() {
    json_string := `
    {
        "firstname": "Rocky",
        "lastname": "Sting",
        "city": "London"
    }`

    emp1 := new(Employee)
    json.Unmarshal([]byte(json_string), emp1)
    fmt.Println(emp1)

    emp2 := new(Employee)
    emp2.FirstName = "Ramesh"
    emp2.LastName = "Soni"
    emp2.City = "Mumbai"
    jsonStr, _ := json.Marshal(emp2)
    fmt.Printf("%s\n", jsonStr)
}
```

The tags are represented as raw string values (wrapped within a pair of ``) and ignored by normal code execution.

Add Method to Struct Type

You can also add methods to `struct` types using a `method receiver`. A method **EmpInfo** is added to the **Employee** struct.

Example

```
package main

import "fmt"

type Salary struct {
    Basic, HRA, TA float64
}

type Employee struct {
    FirstName, LastName, Email string
    Age                        int
    MonthlySalary              []Salary
}

func (e Employee) EmpInfo() string {
    fmt.Println(e.FirstName, e.LastName)
    fmt.Println(e.Age)
    fmt.Println(e.Email)
    for _, info := range e.MonthlySalary {
        fmt.Println("====================")
        fmt.Println(info.Basic)
        fmt.Println(info.HRA)
        fmt.Println(info.TA)
    }
    return "----------------------"
}

func main() {

    e := Employee{
        FirstName: "Mark",
        LastName:  "Jones",
        Email:     "mark@gmail.com",
        Age:       25,
        MonthlySalary: []Salary{
            Salary{
```

```
                Basic: 15000.00,
                HRA:   5000.00,
                TA:    2000.00,
            },
            Salary{
                Basic: 16000.00,
                HRA:   5000.00,
                TA:    2100.00,
            },
            Salary{
                Basic: 17000.00,
                HRA:   5000.00,
                TA:    2200.00,
            },
        },
    }

    fmt.Println(e.EmpInfo())
}
```

Assign Default Value for Struct Field

Method of assigning a custom default value can be achieve by using constructor function. Instead of creating a struct directly, the `Info` function can be used to create an `Employee` struct with a custom default value for the **Name** and **Age** field.

Example

```
package main

import "fmt"

type Employee struct {
    Name string
    Age  int
}
```

```
func (obj *Employee) Info() {
    if obj.Name == "" {
        obj.Name = "John Doe"
    }
    if obj.Age == 0 {
        obj.Age = 25
    }
}

func main() {
    emp1 := Employee{Name: "Mr. Fred"}
    emp1.Info()
    fmt.Println(emp1)

    emp2 := Employee{Age: 26}
    emp2.Info()
    fmt.Println(emp2)
}
```

This is a technique rather than something that is part of the Golang specification.

Find Type of Struct

The **reflect** package support to check the underlying type of a struct.

Example

```
package main

import (
    "fmt"
    "reflect"
)

type rectangle struct {
    length  float64
    breadth float64
```

```
    color   string
}

func main() {
    var rect1 = rectangle{10, 20, "Green"}
    fmt.Println(reflect.TypeOf(rect1))         // main.rectangle
    fmt.Println(reflect.ValueOf(rect1).Kind()) // struct

    rect2 := rectangle{length: 10, breadth: 20, color: "Green"}
    fmt.Println(reflect.TypeOf(rect2))         // main.rectangle
    fmt.Println(reflect.ValueOf(rect2).Kind()) // struct

    rect3 := new(rectangle)
    fmt.Println(reflect.TypeOf(rect3))         // *main.rectangle
    fmt.Println(reflect.ValueOf(rect3).Kind()) // ptr

    var rect4 = &rectangle{}
    fmt.Println(reflect.TypeOf(rect4))         // *main.rectangle
    fmt.Println(reflect.ValueOf(rect4).Kind()) // ptr
}
```

Comparing Structs with the Different Values Assigned to Data Fields

Structs of the same type can be compared using comparison operator.

Example

```
package main

import "fmt"

type rectangle struct {
    length  float64
    breadth float64
    color   string
```

```
}

func main() {
    var rect1 = rectangle{10, 20, "Green"}
    rect2 := rectangle{length: 20, breadth: 10, color: "Red"}

    if rect1 == rect2 {
        fmt.Println("True")
    } else {
        fmt.Println("False")
    }

    rect3 := new(rectangle)
    var rect4 = &rectangle{}

    if rect3 == rect4 {
        fmt.Println("True")
    } else {
        fmt.Println("False")
    }
}
```

Copy Struct Type Using Value and Pointer Reference

`r2` will be the same as `r1`, it is a copy of `r1` rather than a reference to it. Any changes made to `r2` will not be applied to `r1` and vice versa. When `r3` is updated, the underlying memory that is assigned to `r1` is updated.

Example

```
package main

import "fmt"

type rectangle struct {
```

```
    length  float64
    breadth float64
    color   string
}

func main() {
    r1 := rectangle{10, 20, "Green"}
    fmt.Println(r1)

    r2 := r1
    r2.color = "Pink"
    fmt.Println(r2)

    r3 := &r1
    r3.color = "Red"
    fmt.Println(r3)

    fmt.Println(r1)
}
```

Output

```
{10 20 Green}
{10 20 Pink}
&{10 20 Red}
{10 20 Red}
```

As both `r1` and `r3` both reference the same underlying memory, their values are the same. Printing the values of `r3` and `r1` shows that the values are the same.

‹ Previous Next ›

- How to get struct variable information using reflect package?
- How to use wildcard or a variable in our URL for complex routing?
- How to read current directory using Readdir?
- How to use for and foreach loop?
- How to convert Boolean Type to String in Go?
- How to create a photo gallery in Go?
- Find element in a slice and move it to first position?
- Subtract N number of Year, Month, Day, Hour, Minute, Second, Millisecond, Microsecond and Nanosecond to current date-time.
- Golang Slice vs Map Benchmark Testing
- Golang Functions Returning Multiple Values

Golang Programs is designed to help beginner programmers who want to learn web development technologies, or start a career in website development. Tutorials, references, and examples are constantly reviewed to avoid errors, but we cannot warrant full correctness of all content.

Contact Us