

# 日志项目实战&反射

作者: tony

# 目录

1. 日志库需求分析
2. 日志库接口设计
3. 文件日志库开发
4. Console日志开发
5. 日志使用以及测试

# 日志库需求分析

## 1. 日志库产生的背景

- A. 程序运行是个黑盒
- B. 而日志是程序运行的外在表现
- C. 通过日志，可以知道程序的健康状态

# 日志库需求分析

## 2. 日志打印级别设置

- A. Debug级别：用来调试程序，日志最详细。对程序性能影响比较大。
- B. Trace级别：用来追踪问题。
- C. Info级别：打印程序运行过程中比较重要的信息，比如访问日志
- D. Warn级别：警告日志，说明程序运行出现了潜在的问题
- E. Error级别：错误日志，程序运行发生错误，但不影响程序运行。
- F. Fatal级别：严重错误日志，发生的错误会导致程序退出

## 日志库需求分析

### 3. 日志存储的位置

A. 直接输出到控制台

B. 打印到文件里

C. 直接打印到网络中，比如kafka

## 日志库接口设计

### 4. 为什么使用接口？

- A. 定义日志库的规范或者标准
- B. 易于可扩展性
- C. 利于程序的可维护性

# 日志库设计

## 5. 日志库设计

A. 打印各个level的日志

B. 设置级别

C. 构造函数

# 文件日志库实现

## 5. 文件日志库实现



# 终端日志库实现

## 6. 终端日志库实现

# 日志库易用性封装

## 10. 日志库易用性封装

# 日志库使用以及测试

## 11.日志库使用以及测试

# 课后练习

11.把今天的日志库，自己从头实现一遍，变成自己的代码

# 目录

1. 变量介绍

2. 反射介绍

3. 结构体反射

4. 反射总结以及应用场景

# 变量介绍

## 1. 变量的内在机制

A. 类型信息，这部分是元信息，是预先定义好的

B. 值类型，这部分是程序运行过程中，动态改变的

```
var arr [10]int
arr[0] = 10
arr[1] = 20
arr[2] = 30
arr[3] = 40
arr[4] = 50
```

```
type Animal struct {
    Name string
    age int
}

var a Animal
```

# 反射介绍

## 2. 反射与空接口

- A. 空接口可以存储任何类型的变量
- B. 那么给你一个空接口，怎么里面存储的是什么东西？
- C. 在运行时动态的获取一个变量的类型信息和值信息，就叫反射

# 反射介绍

## 3. 怎么分析?

A. 内置包 `reflect`

B. 获取类型信息: `reflect.TypeOf`

C. 获取值信息: `reflect.ValueOf`



# 反射介绍

## 4. 基本数据类型分析

```
package main

import (
    "fmt"
    "reflect"
)

func main() {
    var x float64 = 3.4
    fmt.Println("type:", reflect.TypeOf(x))
}
```

## 反射介绍

### 5. Type.Kind(), 获取变量的类型

```
package main

import (
    "fmt"
    "reflect"
)

func main() {
    var x float64 = 3.4
    t := reflect.TypeOf(x)
    fmt.Println("type:", t.Kind())
}
```

```
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  
    Ptr  
    Slice  
    String  
    Struct  
    UnsafePointer  
)
```

## 反射介绍

### 6. reflect.ValueOf, 获取变量的值相关信息

```
var x float64 = 3.4
v := reflect.ValueOf(x)

//和reflect.TypeOf功能是一样的
fmt.Println("type:", v.Type())
fmt.Println("kind is float64:", v.Kind() == reflect.Float64)

fmt.Println("value:", v.Float())
```

## 反射介绍

### 7. 通过反射设置变量的值

```
var x float64 = 3.4
v := reflect.ValueOf(x)

fmt.Println("type:", v.Type())
fmt.Println("kind is float64:", v.Kind() == reflect.Float64)

v.SetFloat(6.8)
fmt.Println("value:", v.Float())
```

Panic, 程序崩溃了

## 反射介绍

### 8. 通过反射设置变量的值

```
var x float64 = 3.4
//传地址进去，不传地址的话，改变的是副本的值
//所以在reflect包里直接崩溃了!!!!
v := reflect.ValueOf(&x)

fmt.Println("type:", v.Type())
fmt.Println("kind is float64:", v.Kind() == reflect.Float64)

v.SetFloat(6.8)
fmt.Println("value:", v.Float())
```

我靠，还报错

## 反射介绍

### 9. 通过反射设置变量的值

```
var x float64 = 3.4
//传地址进去，不传地址的话，改变的是副本的值
//所以在reflect包里直接崩溃了!!!!
v := reflect.ValueOf(&x)

fmt.Println("type:", v.Type())
fmt.Println("kind is float64:", v.Kind() == reflect.Float64)

//通过Elem()获取指针指向的变量，从而完成赋值操作。
//正常操作是通过*号来解决的，比如
//var *p int = new(int)
//*p = 100
v.Elem().SetFloat(6.8)
fmt.Println("value:", v.Float())
```

## 反射介绍

### 10.通过反射设置变量的值

```
var x float64 = 3.4
v := reflect.ValueOf(&x)

fmt.Println("type:", v.Type())
fmt.Println("kind is float64:", v.Kind() == reflect.Float64)

v.Elem().SetInt(100)
fmt.Println("value:", v.Float())
```

我靠，又犯贱了



# 结构体反射

## 11. 获取结构体类型相关信息

```
package main
import (
    "fmt"
    "reflect"
)

type S struct {
    A int
    B string
}

func main() {
    s := S{23, "skidoo"}
    v := reflect.ValueOf(s)
    t := v.Type()

    for i := 0; i < v.NumField(); i++ {
        f := v.Field(i)
        fmt.Printf("%d: %s %s = %v\n", i,
            t.Field(i).Name, f.Type(), f.Interface())
    }
}
```

# 结构体反射

## 12. 获取结构体类型相关信息

```
package main
import (
    "fmt"
    "reflect"
)

type S struct {
    A int
    B string
}

func main() {
    s := S{23, "skidoo"}
    v := reflect.ValueOf(s)
    t := v.Type()

    for i := 0; i < v.NumField(); i++ {
        f := v.Field(i)
        fmt.Printf("%d: %s %s = %v\n", i,
            t.Field(i).Name, f.Type(), f.Interface())
    }
}
```

# 结构体反射

## 13. 设置结构体相关字段的值

```
package main
import (
    "fmt"
    "reflect"
)

type S struct {
    A int
    B string
}

func main() {
    s := S{23, "skidoo"}
    v := reflect.ValueOf(&s)
    t := v.Type()

    v.Elem().Field(0).SetInt(100)
    for i := 0; i < v.Elem().NumField(); i++ {
        f := v.Elem().Field(i)
        fmt.Printf("%d: %s %s = %v\n", i,
            t.Elem().Field(i).Name, f.Type(), f.Interface())
    }
}
```

# 结构体反射

## 14. 获取结构体的方法信息

```
package main
import (
    "fmt"
    "reflect"
)
type S struct {
    A int
    B string
}
func (s *S) Test() {
    fmt.Println("this is a test")
}
func main() {
    s := S{23, "skidoo"}
    v := reflect.ValueOf(&s)
    t := v.Type()
    v.Elem().Field(0).SetInt(100)
    fmt.Println("method num:", v.NumMethod())
    for i := 0; i < v.NumMethod(); i++ {
        f := t.Method(i)
        fmt.Printf("%d method, name:%v, type:%v\n", i, f.Name, f.Type)
    }
}
```

# 结构体反射

## 15. 调用结构体中的方法

```
package main
import (
    "fmt"
    "reflect"
)
type S struct {
    A int
    B string
}
func (s *S) Test() {
    fmt.Println("this is a test")
}
func (s *S)SetA(a int) {
    s.A = a
}
func main() {
    s := S{23, "skidoo"}
    v := reflect.ValueOf(&s)
    m := v.MethodByName("Test")
    var args1 []reflect.Value
    m.Call(args1)
    setA := v.MethodByName("SetA")
    var args2 []reflect.Value
    args2 = append(args2, reflect.ValueOf(100))
    setA.Call(args2)
    fmt.Printf("s:%#v\n", s)
}
```

# 结构体反射

## 16. 获取结构体中tag信息

```
package main

import (
    "fmt"
    "reflect"
)

type S struct {
    F string `species:"gopher" color:"blue" json:"f"`
}

func main() {
    s := S{}
    st := reflect.TypeOf(s)
    field := st.Field(0)
    fmt.Println(field.Tag.Get("color"), field.Tag.Get("species"),
        field.Tag.Get("json"))
}
```

## 反射总结以及应用场景

### 17. 反射总结

A. 在运行时动态的获取一个变量的类型信息和值信息

### 18. 应用场景

A. 序列化和反序列化，比如json, protobuf等各种数据协议

B. 各种数据库的ORM，比如gorm, sqlx等数据库中间件

C. 配置文件解析相关的库，比如yaml、ini等

**这些场景，我们实现都不知道，  
数据的具体类型，所以要用反射**