标准库

前言

在这个多样化的编程语言生态系统中，Go语言凭借其简洁、高效、并发支持等特性逐渐崭露头角。

作为一门开源的静态编程语言，Go语言自带了丰富的标准库，为开发者提供了强大的工具和功能。

本文将深入介绍Go语言几种标准库，帮助读者更好地了解和利用这些库，提高编程效率。

引入

最简单、最常用的标准库就是fmt，fmt包实现了类似C语言printf和scanf的格式化I/O。格式化动作源自C语言但更简单。这个不必多说。

附上go语言标准库中文文档https://studygolang.com/pkgdoc

字节、字符串、utf-8

• utf8是什么, 字节是什么, 字符串和字节又有什么关系

◦ 位(Bit)是一种用于计量存储容量和数据传输的数据单位。一个位可以表示一个0或1，也就是一个二进制数。位是计算机信息技术的基本概念。

◦ 字节(Byte)是一种用于计量存储容量和数据传输的数据单位，通常由8位组成。一个字节可可以表示0到255之间的一个数值。不同的编码方式，如ASCII、UTF-8等，可以用不同的字节来表示不同的字符。

◦ utf8是一种编码方式，它可以用1到4个字节来表示一个Unicode字符。Unicode字符是一种国际标准，它可以表示世界上所有的文字。

◦ 字符串是一种数据类型，它由多个字符组成，可以表示文本信息。在go语言中，字符串是不可变的，也就是说，一旦创建了一个字符串，就不能修改它的内容，只能通过重新赋值改变变量的值。

◦ 字符串和字节之间的关系是，字符串可以转换为字节切片，也就是一个包含多个字节的数组。这样可以方便地对字符串进行分割、拼接、修改等操作。反过来，字节切片也可以转换为字符串，这样可以方便地输出或存储字节切片的内容。

◦ 例如，以下代码展示了如何将一个字符串转换为字节切片，然后修改其中的一个字节，再转换回字符串：

```
package main

import "fmt"

func main() {
    s := "Hello, 世界"
    fmt.Println(s)

    b := []byte(s)
    fmt.Println(b)

    b[7] = 229
    fmt.Println(b)

    s = string(b)
    fmt.Println(s)
}
```

Math

math包提供了基本的数学常数和数学函数。

直接代码示例：

```
package main

import (
    "fmt"
    "math"
)

func main() {

    //注意传入和返回参数类型一般都是float64（int类型自己强转一下）

    // math.Max math.Min 对比两个数字取最大值或最小值，
    x, y := 1.1, 2.2
    fmt.Println(math.Max(x, y))
    fmt.Println(math.Min(x, y))

    // math.Abs 取绝对值
    z := -1.3
    fmt.Println(math.Abs(z))

    // math.Sqrt 返回x的二次方根
    x = 2.0
    fmt.Println(math.Sqrt(x))

    // math.Pow 返回x^y
    x, y = 2.0, 3.0
    fmt.Println(math.Pow(x, y))
}
```

Strings

strings包实现了用于操作字符的简单函数。

代码示例：

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    s := "Hello, 世界"

    // 判断字符串是否包含某个字串
    fmt.Println(strings.Contains(s, "世界"))

    // 判断字符串是否以某个字串开头或结尾
    fmt.Println(strings.HasPrefix(s, "Hello"))
    fmt.Println(strings.HasSuffix(s, "Hello"))

    // 统计字符串中某个字串出现的次数
    fmt.Println(strings.Index(s, "l"))

    // 替换字符串中某个字串为另一个字串
    fmt.Println(strings.Replace(s, "世界", "world", -1))

    // 将字符串全部转换为大写或者小写
    fmt.Println(strings.ToUpper(s))
    fmt.Println(strings.ToLower(s))

    // 将字符串按照某个分隔符分隔为一个切片
    fmt.Println(strings.Split(s, ","))

    //  将一个切片按照某个分隔符拼接为一个字符串
    fmt.Println(strings.Join([]string{"a", "b", "c"}, "-"))

    // 返回将字符串按照空白分割的多个字符串。
    // 如果字符串全部是空白或者是空字符串的话，会返回空切片。
    fmt.Println(strings.Fields(s))
}
```

Time

time包提供了时间的显示和测量用的函数。日历的计算采用的是公历。

示例代码：

```
package main

import (
    "fmt"
```

```
    "time"
)

func main() {
    // 获取当前时间
    t := time.Now()
    fmt.Println(t)

    // 获取当前时间的年月日时分秒
    fmt.Println(t.Year())
    fmt.Println(t.Month())
    fmt.Println(t.Day())
    fmt.Println(t.Hour())
    fmt.Println(t.Minute())
    fmt.Println(t.Second())

    // 获取当前时间的星期
    fmt.Println(t.Weekday())

    // 获取当前时间的纳秒
    fmt.Println(t.Nanosecond())

    // 获取当前时间的时区
    fmt.Println(t.Location())

    // 获取当前时间的时间戳
    fmt.Println(t.Unix())

    // 格式化当前时间为字符串
    // 在time包中，格式化字符串使用的格式化字符分别是：
    // "%Y"或"2006": 代表年
    // "%m"或"01": 代表月
    // "%d"或"02": 代表天
    // "%H"或"15": 代表小时
    // "%M"或"04": 代表分钟
    // "%s"或"05": 代表秒
    // 它们类似于格式化字符时的"%f"、"%d"、"%s"等
    // "2006-01-02 15:04:05"这个样式的选择不是随意的
    // 它来源于一个叫做Mon Jan 2 15:04:05 -0700 MST 2006的参考时间，
    // 这个时间是Go语言的创始人Rob Pike在2006年1月2日下午3点4分5秒时收到的一封电子邮件的
    // 这个时间恰好包含了所有可能的时间格式，而且每个数字都不重复，所以可以作为一个通用的时
    fmt.Println(t.Format("2006-01-02 15:04:05"))

    // 解析字符串为时间
    t, err := time.Parse("2006-01-02 15:04:05", "2023-11-07 16:28:16")
    if err != nil {
       fmt.Println(err)
    } else {
       fmt.Println(t)
    }
```

```
    // 计算两个时间的差值
    t1 := time.Date(2023, 11, 7, 16, 28, 16, 0, time.UTC)
    t2 := time.Date(2023, 11, 8, 16, 28, 16, 0, time.UTC)
    d := t1.Sub(t2)
    fmt.Println(d)

    // 判断两个时间是否相等
    fmt.Println(t1.Equal(t2))

    // 判断一个时间是否在另一个时间之前或之后
    fmt.Println(t1.Before(t2))
    fmt.Println(t1.After(t2))

    // 添加或减少一个时间的年月日时分秒
    fmt.Println(t1.Add(time.Hour * 24))
    fmt.Println(t1.Add(-time.Hour * 24))
}
```

Strconv

strconv包实现了基本数据类型和其字符串表示的相互转换。

提一个小问题，你知道int类型怎么转换为对应的string类型呢（98->"98"），强制类型转换？可以试试

```
package main

import "fmt"

func main() {
    x := 98
    str := string(x)
    fmt.Println(str)
}
```

这时候就要用到我们的strconv标准库函数了。

代码示例：

```
package main

import (
    "fmt"
    "strconv"
)

func main() {
    // 字符串转换为整型
    str := "12"
    fmt.Println(strconv.Atoi(str))

    // 整型转换为字符串
    x := 12
    fmt.Println(strconv.Itoa(x))
}
```

Testing

testing 提供对 Go 包的自动化测试的支持。通过 `go test` 命令，能够自动执行如下形式的任何函数：

```
func TestXxx(*testing.T)
```

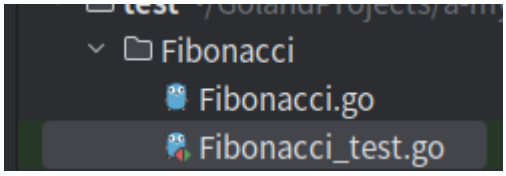
其中 Xxx 可以是任何字母数字字符串（但第一个字母不能是 [a-z]），用于识别测试例程。

在这些函数中，使用 Error, Fail 或相关方法来发出失败信号。

要编写一个新的测试套件，需要创建一个名称以 _test.go 结尾的文件，该文件包含 `TestXxx` 函数，如上所述。
将该文件放在与被测试的包相同的包中。该文件将被排除在正常的程序包之外，但在运行 “go test” 命令时将被包含。

Example:

新建一个Fibonacci文件夹，里面写一个Fibonacci,go文件，包含实现斐波那契的实现；在写一个Fibonacci_test.go，为Fibonacci.go的测试。



Fibonacci.go

```
package Fibonacci

// 0 1 1 2 3 5 8 13 21 34 55 ..

func Fib(n int) int {
    dp := make([]int, n+1)
    dp[0], dp[1] = 0, 1
    for i := 2; i <= n; i++ {
       dp[i] = dp[i-1] + dp[i-2]
    }
    return dp[n]
}
```

Fibonacci_test.go

```
package Fibonacci

import (
    "fmt"
    "testing"
)

func TestFib(t *testing.T) {
    fmt.Println(Fib(1))
    fmt.Println(Fib(2))
    fmt.Println(Fib(3))
    fmt.Println(Fib(4))
```

```
    fmt.Println(Fib(5))
    fmt.Println(Fib(6))
    fmt.Println(Fib(7))
    fmt.Println(Fib(8))
    fmt.Println(Fib(9))
    fmt.Println(Fib(10))
}
```

bufio && os

- 缓冲是一种用来提高输入输出效率的技术，它可以将数据暂时存放在内存中的一部分空间，这部分

空间就叫做缓冲区。

- 缓冲区的作用是减少对底层设备或网络的访问次数，从而提高数据的读写速度，降低系统的开销，

提升用户的体验。

- 总的来说，缓冲输入输出是一种提高性能的方法，它可以将多个小的读写操作合并为一个大的读写

操作，从而减少系统调用的次数。

bufio是go语言的一个标准库包，它提供了一些有用的函数和类型来处理缓冲输入输出。

例如，以下代码展示了如何使用bufio包中的一些函数和类型：

```
package main

import (
    "bufio"
    "fmt"
    "os"
)

func main() {
    // 打开一个文件
    f, err := os.Open("test.txt")
    if err != nil {
       panic(err)
    }
    defer f.Close() // 记得关闭文件

    // 创建一个缓冲读取器
    r := bufio.NewReader(f)

    // 读取文件的一行
    line, err := r.ReadString('\n')
    if err != nil {
```

```
        panic(err)
    }
    fmt.Println(line)

    // 读取文件的一个字节
    b, err := r.ReadByte()
    if err != nil {
        panic(err)
    }
    fmt.Println(b)

    // 创建一个文件
    f, err = os.Create("test2.txt")
    if err != nil {
        panic(err)
    }
    defer f.Close() // 延时关闭

    // 创建一个缓冲写入器
    w := bufio.NewWriter(f)

    // 写入一个字符串到文件
    _, err = w.WriteString("Hello, 世界\n")
    if err != nil {
        panic(err)
    }
    // 刷新缓冲区， 将数据写入文件
    w.Flush()

    // 写入一个字节到文件
    err = w.WriteByte(65)
    if err != nil {
        panic(err)
    }
    w.Flush()
}
```

相信大家在之前都遇到过这样一个问题（没遇到过的就去试试QWQ），在输入字符串的时候，怎么输入中间有空格的字符串呢？普通的fmt包函数是无法实现的，就得用bufio包来读取输入了。

Example:

```
package main

import (
    "bufio"
    "fmt"
    "os"
)

func main() {
    //s := ""
    //fmt.Scanln(&s)
    //fmt.Println(s)
```

```
    // 打开标准输入
    r := bufio.NewReader(os.Stdin)

    // 读取标准输入的一行
    line, err := r.ReadString('\n')
    if err != nil {
       fmt.Println(err)
    }
    fmt.Println(line)

    // 打开标准输出
    w := bufio.NewWriter(os.Stdout)

    // 写入一个字符串到标准输出
    _, err = w.WriteString("Hello, 世界\n")
    if err != nil {
       fmt.Println(err)
    }
    w.Flush()
}
```

Encoding/json

JSON（JavaScript Object Notation）是一种轻量级的数据交换格式，易于人阅读和编写，同时也易于机器解析和生成。它基于JavaScript的一个子集，但是JSON是独立于语言的文本格式，许多编程语言都可以使用JSON。

我们主要encoding/json包里关于jaon的序列化和反序列化：

在Go语言中，序列化（Serialization）和反序列化（Deserialization）是处理数据结构和数据格式转换的两个重要概念。它们通常用于数据存储和网络通信。

序列化（Serialization）

序列化是指将程序中的数据结构或对象状态转换为可以存储或传输的格式（通常是字符串或字节序列）的过程。这种格式可以被写入文件、存储在数据库中，或者通过网络发送到其他计算机。

原因：

数据持久化：将数据结构保存到文件或数据库中，以便在程序重启后能够恢复。

网络通信：在网络上传输数据时，需要将数据结构转换为一种中间格式，以便在不同的系统和语言之间传输。

跨平台兼容性：确保数据可以在不同的操作系统和硬件平台上使用。

反序列化（Deserialization）

反序列化是序列化的逆过程，即将序列化后的数据（通常是字符串或字节序列）转换回程序中的数据结构或对象。

原因：

数据恢复：从文件或数据库中读取序列化的数据，并将其恢复为程序中的数据结构，以便程序可以操作和使用。

网络通信：接收从网络上传输的数据，并将其转换回原始的数据结构，以便程序可以进一步处理。

数据整合：在微服务架构中，不同服务之间可能需要交换数据，反序列化是将接收到的数据整合到本地数据结构中的关键步骤。

Example:

```
import "encoding/json"

type Person struct {
    Name string `json:"name"`
    Age  int    `json:"age"`
}

func main() {
    p := Person{Name: "John Doe", Age: 30}
    jsonData, err := json.Marshal(p)
    if err != nil {
        // 处理错误
    }
    fmt.Println(string(jsonData)) // 输出: {"name":"John Doe","age":30}
}
import "encoding/json"

type Person struct {
    Name string `json:"name"`
    Age  int    `json:"age"`
}

func main() {
    jsonData := `{"name":"Jane Doe","age":25}`
    var p Person
    err := json.Unmarshal([]byte(jsonData), &p)
    if err != nil {
        // 处理错误
    }
    fmt.Println(p.Name) // 输出: Jane Doe
}
```

os.OpenFile

os.OpenFile是go语言的一个标准库函数，它可以打开或创建一个文件，并返回一个文件对象。它有三

个参数：

- name：文件的名称，可以是相对路径或绝对路径。
- flag：文件的打开模式，可以是以下常量的组合：

◦ os.O_RDONLY：只读模式

◦ os.O_WRONLY：只写模式

◦ os.O_RDWR：读写模式

◦ os.O_APPEND：追加模式，写入的数据会添加到文件末尾

◦ os.O_CREATE：创建模式，如果文件不存在，会创建一个新文件

◦ os.O_EXCL：排他模式，如果文件已存在，会返回一个错误

◦ os.O_SYNC：同步模式，每次写入操作会等待数据写入磁盘

◦ os.O_TRUNC：截断模式，如果文件已存在，会清空文件内容

• perm：文件的权限，可以是以下常量的组合：

◦ os.FileMode(0)：无权限

◦ os.FileMode(1)：执行权限

◦ os.FileMode(2)：写入权限

◦ os.FileMode(4)：读取权限

◦ os.FileMode(7)：读写执行权限例如，以下代码展示了如何使用os.OpenFile函数：

一个简单的example:

```
package main

import (
    "fmt"
    "os"
)

func main() {
    // 打开一个只读的文件
    f, err := os.OpenFile("test.txt", os.O_RDONLY, 0)
    if err != nil {
       panic(err)
    }
    defer f.Close() // 延迟关闭

    // 读取文件内容
    b := make([]byte, 1024)
    n, err := f.Read(b)
    if err != nil {
       panic(err)
    }
    fmt.Println(string(b[:n]))
}
```

面向对象编程

Go语言是一种静态类型的编程语言，它支持面向对象的编程范式，但是它的面向对象和其他语言有一

些不同。在本文档中，我们将介绍Go语言的面向对象的基本概念和用法，包括结构体、接口、方法、

继承、多态和封装。

Go语言的面向对象是通过组合和接口来实现的，而不是通过传统的继承和多态。Go语言的面向对象更

加灵活和简洁，也更加符合现实世界的模型。Go语言的面向对象可以让我们编写更抽象和通用的代

码，也可以提高代码的可维护性和可复用性。

复习

结构体

结构体是Go语言中最基本的自定义类型，它可以用来表示一个实体，比如一个人、一个动物、一个图

形等。结构体的定义如下：

```
type Person struct {
    name string
    age int
    gender string
}
```

上面的代码定义了一个名为Person的结构体类型，它有三个字段：name, age, gender。我们可以使用

结构体字面量来创建一个结构体的实例，比如：

```
p := Person{name: "Alice", age: 20, gender: "female"}
```

上面的代码创建了一个Person类型的变量p，并给它的字段赋值。我们可以使用点号（.）来访问结构

体的字段，比如：

```
fmt.Println(p.name) // 输出Alice
fmt.Println(p.age) // 输出20
```

结构体可以看作是一种数据的容器，它可以存储不同类型的数据，但是它本身并没有行为，也就是

说，它不能做任何事情。为了让结构体具有行为，我们需要给它定义方法。

方法

方法是一种特殊的函数，它与一个特定的类型相关联，称为接收者类型。方法的定义如下：

```
func (receiver type) methodName(parameters) (results) {
    // ....
}
```

上面的代码定义了一个方法，它的接收者类型是type，方法名是methodName，参数列表是

parameters，返回值列表是results。我们可以在方法体中使用接收者变量来访问接收者的字段或者调

用其他方法。比如，我们可以给Person类型定义一个方法，用来打印自我介绍：

```
func (p Person) introduce() {
    fmt.Printf("Hello, I am %s, %d years old, %s.\n", p.name, p.age, p.gender)
}
```

上面的代码定义了一个名为introduce的方法，它的接收者类型是Person，它没有参数也没有返回值。

它使用fmt.Printf函数来打印接收者p的字段。我们可以使用点号（.）来调用方法，比如：

```
p.introduce() //输出 Hello, I am Alice, 20 years old, female.
```

上面的代码调用了p的introduce方法。方法可以看作是一种行为，它可以让结构体做一些事情，或者

与其他结构体交互。方法可以让结构体具有一些特性，比如能力、属性、状态等。为了让结构体具有

更多的特性，我们需要使用接口。

接口

接口是Go语言中一种抽象的类型，它定义了一组方法的签名，但是没有实现。接口的定义如下：

```
type interfaceName interface {
    methodName1(parameters) (results)
    methodName2(parameters) (results)
    // ...
}
```

上面的代码定义了一个名为interfaceName的接口类型，它有若干个方法的签名，比如

methodName1, methodName2等。接口可以用来表示一种抽象的特性，比如行为、能力、属性等。

比如，我们可以定义一个名为Animal的接口，用来表示动物的行为：

```
type Animal interface {
    eat()
    sleep()
    sound()
}
```

上面的代码定义了一个名为Animal的接口类型，它有三个方法的签名：eat, sleep, sound。这些方法

表示动物的一些共同的行为，但是不同的动物可能有不同的实现方式。为了让一个结构体实现一个接

口，我们需要让它定义所有接口中的方法。比如，我们可以定义一个名为Dog的结构体，用来表示狗：

```
type Dog struct {
    name string
}
```

上面的代码定义了一个名为Dog的结构体类型，它有一个字段name。为了让Dog实现Animal接口，我

们需要给它定义eat, sleep, sound三个方法，比如：

```
func (d Dog) eat() {
    fmt.Println(d.name, "is eating.")
}

func (d Dog) sleep() {
    fmt.Println(d.name, "is sleeping.")
}

func (d Dog) sound() {
    fmt.Println(d.name, "is barking.")
}
```

上面的代码给Dog类型定义了三个方法，它们分别打印狗的吃、睡、叫的信息。这样，Dog就实现了

Animal接口，我们可以说Dog是一种Animal。我们可以使用一个Animal类型的变量来存储一个Dog类

型的值，比如：

```
var a Animal
a = Dog{name: "Bob"}
```

上面的代码声明了一个Animal类型的变量a，并给它赋值为一个Dog类型的值。我们可以使用a来调用

Animal接口中的方法，比如：

```
a.eat() // 输出 Bob is eating.
a.sleep() // 输出 Bob is sleeping.
a.sound() // 输出 Bob is barking.
```

上面的代码调用了a的eat, sleep,
sound方法。这些方法的实现是由Dog类型提供的，也就是说，a的行

为是由它的动态类型（Dog）决定的，而不是它的静态类型（Animal）。这种行为的多样性称为多

态。

多态

多态是指同一个接口可以被不同的类型实现，从而具有不同的行为。多态可以让我们编写更通用和灵

活的代码，比如，我们可以定义一个函数，用来让任何一种动物发出声音：

```
func makeSound(a Animal) {
    a.soud()
}
```

上面的代码定义了一个名为makeSound的函数，它接受一个Animal类型的参数a，并调用它的sound

方法。这个函数可以接受任何实现了Animal接口的类型作为参数，比如Dog, Cat, Bird等。不同的类型

会有不同的sound方法的实现，从而产生不同的声音，比如：

```
makeSound(Cat{name: "Bob"}) // 输出 Bob is barking.
makeSound(Cat{name: "Tom"}) // 输出 Tom is meowing.
makeSound(Bird{name: "Jack"}) // 输出 Jack is chirping.
```

上面的代码调用了makeSound函数，并分别传入了Dog, Cat, Bird类型的值作为参数。这些值都实现

了Animal接口，所以都可以被接受。函数内部调用了它们的sound方法，从而产生了不同的声音。这

就是多态的效果，它可以让我们编写更抽象和通用的代码，而不需要关心具体的类型和实现。

继承

继承是指一个类型可以从另一个类型继承一些特性，比如字段或者方法。Go语言没有提供继承的语

法，但是可以通过组合的方式来实现继承的效果。组合是指一个类型可以包含另一个类型作为它的字

段，从而拥有它的特性。比如，我们可以定义一个名为Student的结构体，用来表示学生：

```
type Person struct {
    name string
    age int
    gender string
}

type Student struct {
    Person //匿名字段，表示Student包含了Person类型
    school string
    grade int
}
```

上面的代码定义了一个名为Student的结构体类型，它有两个字段：school, grade。它还有一个匿名

字段Person，表示Student包含了Person类型。这样，Student就继承了Person的所有字段和方法，

比如：

```
s := Student{Person: Person{name: "Bob", age: 18, gender: "male"}, school: "High
```

我们可以使用s来访问Student的字段和方法，比如：

```
fmt.Println(s.name) // 输出 Bob
fmt.Println(s.school) // 输出 High School
s.introduce() // 输出 Hello, I am Bob, 18 years old, male.
```

上面的代码使用s访问了Student的name和school字段，以及Person的introduce方法。我们也可以给

Student定义自己的方法，比如：

```
func (s Student) study() {
    fmt.Println(s.name, "is studying in grade", s.grade, "at", s.school)
}
```

上面的代码给Student类型定义了一个名为study的方法，它打印学生的学习信息。我们可以使用s来调

用这个方法，比如：

```
s.study() // 输出 Bob is studying in grade 12 at High School
```

上面的代码调用了s的study方法。这样，Student就拥有了Person的特性，同时也有自己的特性。这

种继承的效果是通过组合实现的，也就是说，Student是由Person和其他字段组合而成的。这种组合

的方式比传统的继承更灵活和简洁，因为它不需要指定父类和子类的关系，也不需要使用特殊的关键

字。

封装

封装是指隐藏一个类型的内部实现细节，只暴露一些必要的接口给外部使用。封装可以保护类型的数

据不被外部随意修改，也可以提高类型的可维护性和可复用性。Go语言没有提供封装的语法，但是可

以通过命名规则来实现封装的效果。Go语言中，如果一个标识符（变量、常量、类型、函数、方法

等）的首字母是大写的，那么它就是公开的（public），可以被其他包访问；如果一个标识符的首字

母是小写的，那么它就是私有的（private），只能在当前包内部访问。比如，我们可以定义一个名为

Account的结构体，用来表示银行账户：

```
type Account struct {
    id string // 私有字段，只能在当前包内部访问
```

```
    balance float64 // 私有字段，只能在当前包内部访问
}
```

上面的代码定义了一个名为Account的结构体类型，它有两个字段：id,
balance。这两个字段的首字母

都是小写的，所以它们都是私有的，只能在当前包内部访问。这样，我们就封装了Account的内部数

据，不让外部直接访问或修改。为了让外部可以使用Account，我们需要给它定义一些公开的方法，比

如：

```
func NewAccount(id string, balance float64) *Account {
    //创建一个新的Account实例，并返回它的指针
    return &Account{id: id, balance: balance}
}

func (a *Account) GetID() string {
    //返回账户的id
    return a.id
}

func (a *Account) GetBalance() float64 {
    //返回账户的余额
    return a.balance
}

func (a *Account) Deposit(amount float64) {
    //存款，增加账户的余额
    a.balance += amount
}

func (a *Account) Withdraw(amount float64) bool {
    //取款，减少账户的余额，如果余额不足，返回false，否则返回true
    if a.balance < amount {
        return false
    }
    a.balance -= amount
    return true
}
```

上面的代码给Account类型定义了五个方法，它们的首字母都是大写的，所以它们都是公开的，可以被

其他包访问。这些方法提供了Account的一些必要的接口，比如创建、查询、存款、取款等。我们可以

使用这些方法来操作Account，比如：

```
a := NewAccount("123456", 1000) //创建一个新的Account实例
fmt.Println(a.GetID()) //输出 123456
fmt.Println(a.GetBalance()) // 输出 1000
```

```
a.Deposit(500) // 存款500
fmt.Println(a.GetBalance()) // 输出 1500
ok := a.Withdraw(2000) // 取款2000
fmt.Println(ok) // 输出 false
fmt.Println(a.GetBalance()) // 输出 1500
```

上面的代码使用NewAccount函数创建了一个新的Account实例，并给它的id和balance赋值。然后，

使用GetID和GetBalance方法查询了账户的id和余额。接着，使用Deposit和Withdraw方法进行了存

款和取款操作，并打印了操作的结果。这些操作都是通过Account的公开方法进行的，而不是直接访问

或修改Account的私有字段。这样，我们就实现了Account的封装，保护了它的内部数据，同时提供了

一些必要的接口给外部使用。

作业

lv0

自己敲一遍上面代码，看看结果是否与自己预期的一样（这个可以不用交上来）

lv1

完成一道leetcode算法题：https://leetcode.cn/problems/reverse-words-in-a-string/description/?envType=study-plan-v2&envId=top-interview-150

完成后使用Testing包对其进行测试

lv2

编写一个程序，从一个文本文件中读取一系列的日期和事件，例如：

```
2024-11-17 国际大学生节
2024-12-01 世界爱滋病日
2024-12-03 世界残疾人日
```

然后，根据当前的日期，输出最近的一个事件和它的倒计时，例如：

```
最近的一个事件是：国际大学生节
还有1天
```

你可以将以下内容作为你的文件(events.txt)

```
2023-10-16 世界粮食日
2023-10-17 国际消除贫困日
2023-10-24 联合国日
2023-10-24 世界发展新闻日
2023-10-29 国际生物多样性日
2023-10-31 万圣节
2023-11-08 中国记者节
2023-11-09 消防宣传日
```

```
2023-11-11 光棍节
2023-11-14 世界糖尿病日
2023-11-17 国际大学生节
2023-12-01 世界爱滋病日
2023-12-03 世界残疾人日
2023-12-04 全国法制宣传日
2023-12-09 世界足球日
2023-12-13 南京大屠杀死难者国家公祭日
2023-12-25 圣诞节
2023-01-01 元旦
```

提示：你可以使用strings.Split函数来分割每一行的日期和事件，使用time.Parse和time.Now函数来解析和获取日期，使用bufio.NewReader和os.OpenFile函数来读取文件。

lv3

假设你正在开发一个博客系统，需要处理文章数据。定义以下结构体：

Author 结构体，包含 Name（姓名）和 Bio（个人简介）两个字段。

Post 结构体，包含 Title（标题），Content（内容），Author（作者，类型为 Author），以及 Tags（标签列表，类型为字符串切片）。

请完成以下任务：

1.创建一个 Author 实例和一个 Post 实例，初始化它们的字段。

2.将 Post 实例序列化为 JSON 字符串。

3.使用字符串操作将步骤2中得到的 JSON 字符串分割成键值对，并打印每个键值对。

4.从步骤3中得到的键值对中，提取 Author 字段，并将其反序列化为 Author 结构体。

5.打印反序列化后的 Author 结构体实例的详细信息。

要求：

确保序列化和可能的反序列化过程中处理了可能的错误。

使用 json.Marshal 函数进行序列化。

使用 strings 包中的函数进行字符串操作。

如果反序列化失败，打印出具体的错误信息。

lv4（可选，自己玩，有问题记得问）

推荐两个好玩的东西。自己去玩一下

github：

注册自己的账号，后续我们开始写大作业会叫你们直接推到github

linux：

开发者必备，可以用虚拟机或者双系统实现，任何发行版都可以，主要是上手玩玩，熟悉一下