接口

设计意图

接口定义

在Java和Go语言中，均提供了定义接口(interface)支持，接口中仅包含方法，不包括方法的实现

Java

```
public interface Readable {
    int read(CharBuffer buffer);
}
```

Go

```
type Reader interface {
    Read(p []byte) (n int, err error)
}
```

接口，实际上是定义了一组行为，将大自然中的事物通过行为的方式抽象描述出来。也就是说，接口定义的是能做什么

动物能够移动(Move)，进食(Eat)和呼吸(Breath)

人和狮子从种类区分的话，属于完全不同的物种，但是他们都能够移动、进食和呼吸，换而言之，都是动物

接口是对同一组事物的不同视角的分类

所有的生物均能进行遗传(Descend)

草莓和ARAS病毒都能够将自身DNA或者RNA遗传给后代，所以他们都是生物

进而言之，接口就是定义了一组规则，用来描述"如果你是...则必须能..."的理念

例如Animal接口定义了Move、Eat以及Breath三个方法，那么当狮子实现了该接口时，就表示狮子是动物，必须能移动、进食和呼吸

以上面的抽象代码为例，如果某个类型实现了Reader接口，表示类型为"阅读者"，那么必须能"读"一些东西

接口是用来模拟真实世界的一个工具

依赖反转原则

在设计原则中，有一条非常重要的原则，即依赖反转原则

高层次模块不应依赖于低层次模块的具体实现细节，两者都应该依赖于抽象

依赖反转原则简单来说就是添加一层抽象(接口)，该抽象模块定义了模块A所依赖的所有行为

而模块B则实现该接口，并在运行时通过依赖注入的方式注入进模块A。如此一来，将来若想要替换掉模块B，只需要重新实现该接口，并在少量的代码中进行改动即可

接口其实就是一个标准，具体的实现遵循该标准，那么上层依赖模块就可以随意地替换底层实现

例如国内目前绝大多数的插座电压均为220V，那么电器(充电器、空调、冰箱)在进行设计时，就可以拿着这个标准进行设计，不管设计成三脚还是两脚插头，只要遵循输入电压为220V即可

接口的设计其实是为了模拟真实世界的多样性与组合型，使得系统对拓展和对某些模块的替换更加地便利

Go接口定义

定义

```
type InterfaceName interface {
    FuncName(p Type) ReturnType
}
```

与Java接口定义相同，只允许定义未有实现的方法，而不允许定义成员变量

实现

Go实现一个接口相当简单，只要实现了接口中所有方法，就认为该类型实现了某一个接口

```
type Reader interface {
    Read()
}

type Kindle struct {}

func (k *Kindle) Read() {
    fmt.Println("Kindle Read")
}
```

不需要implements关键字

指针接收者的方法要比具体类型接收者的方法更加灵活，所以指针接收者的方法会更多一些

接口的值

从接口定义中就可以看出，接口是一个类型。既然是一个类型，那么必然有值

```
var a int   // 对于静态语言而言，类型在编译时期即确定
a = 10      // 此时变量a的类型为int，值为10
```

需要注意的是，接口和基本数据类型之间存在细微差别。接口和基本数据类型都会有一个'编译时的类型(通常称为静态类型)，该类型在编译时期即确定。所以说，接口的类型并不是一个值

接口的值由两部分组成: 接口的动态类型和该类型的值，前者称为动态类型，后者称为动态值

运行时的类型确定和Java的RTTI作用基本相同，都是在运行时确定某接口类型的具体类型

```
var w io.Writer              // 1
w = os.Stdout                // 2
w.Write([]byte("hello"))     // 3
```

1. io.Writer是一个标准库接口，其中包含了Write方法
2. 声明了变量w，其类型为io.Writer。变量在定义时即被初始化，接口类型也不例外。其零值就是将其动态类型和动态值均设置为nil

   一个接口值是否为nil取决于它的动态类型是否为nil，而不是动态值是否为nil

   将os.Stdout这一具体实现类型赋值给了w，相当于将具体类型隐式转换成了一个接口类型，那么此时w就有了动态类型和动态值

   动态类型为*os.File，动态值则为指向*os.file的指针 **此时，由于w有了动态类型(*os.File)，所以，此时w不为nil**
3. 调用该接口值的Write方法，相当于调用了(*os.File).Write()方法

   而在编译时无法得知一个接口值的动态类型会是什么，所以，一定会使用动态分发来在运行时获取实际的方法地址 对应于Java RTTI

Go接口实现

```
type iface struct {
    tab  *itab
    data unsafe.Pointer
}
```

itab是接口类型最为核心的组成部分，在itab内部，包含了接口的编译类型以及运行时的动态类型

itab

```
type itab struct {          // 32bytes
    inter *interfacetype    // 静态类型
    _type *_type            // 动态类型
    hash  uint32            // _type.hash的拷贝值
    _     [4]byte           // 用于内存对齐
    fun   [1]uintptr        // 存储所需函数指针
}
```

data 用于保存实际运行时的数据，是一个指向原始数据的指针

空接口: interface{}

interface{} 表示没有方法的接口定义，所以所有的类型均实现了interface{}，也就是说，可将将任意值赋给空接口类型 这一点与Java完全不同

```
var any interface {}
any = 10
any = true
any = 1.5
```

正是因为有了空接口类型，才能够使得fmt.Println、errof这类函数能够接收任意类型的参数

Go空接口实现

```
type eface struct {
    _type *_type
    data  unsafe.Pointer
}
```

_type 为Go语言类型的运行时表示，包括了一些元信息，包括大小、哈希值等等

data 用于保存实际运行时的数据，是一个指向原始数据的指针

interface{} 和接口类型的实现不完全相同

interface{} 并不代表任意类型，其类型就是interface{}，只不过在运行时进行了隐式转换而已 从实现上就可以看出来，interface{} 在内存中始终占用2个字的内存空间，其它类型则不一定如此

```
func Hugo(v interface{}) {
    /* ... */
}

func main() {
    Hugo(10)
    Hugo("25")
}
```

Hugo函数接收interface{}类型的参数，那么不管实参是什么类型，都会在函数调用时进行隐式的类型转换，将其转换成interface{}类型

```
func Zelda(v []interface{}) {
    /* ... */
}

func main() {
    Zelda([]int{1, 2, 3})
}
```

该代码在编译时即抛出异常的原因在于，[]interface{} 和 []int 是完全不同的类型，且Go不会在运行时帮助用户进行整个slice的隐式类型转换

如果想要调用Zelda()方法，需要逐个的将slice中的数据显式地转换成interface{}类型，再传入至Zelda()函数中

接口的两种不同实现

前面提到了，如果一个类型实现了接口的所有方法，我们就认为该类型实现了该接口，但是没有提到使用值接收者实现，还是使用指针接收者实现

事实上，在Go中，如果将接口和指针组合起来，将会产生非常令人迷惑的行为，一个最直观的例子就是在实现某一个接口时，不允许值接收者和指针接收者共同存在

```
type ReaderWriter interface {
    Read()
    Write()
}

type Kindle struct {}

func (k Kindle) Read() {
    fmt.Println("Kindle Read")
}

func (k *Kindle) Write() {
    fmt.Println("Kindle Write")
}

var k ReaderWriter
k = Kindle()
```

当运行该代码时，编译器会告知Kindle并未实现ReaderWriter接口

所以，在实现接口时，方法要么全部是指针接收者，要么全部是值接收者

当实现方法选择指针接收者时，必须显式地使用指针进行赋值

```
func (k *Kindle) Write() {
    fmt.Println("Kindle Write")
}
var k ReaderWriter
k = &Kindle{} // right
k = Kindle{}  // wrong
```

error接口

Go没有异常机制，所以会使用显式错误返回来表明函数调用过程中是否发生错误，返回的error类型，实际上是一个接口类型

```
type error interface {
    Error() string
}
```

error接口的定义相当简单，只包含一个导出的Error()方法，返回字符串类型

```
type errorString struct {
    s string
}

func (e *errorString) Error() string {
    return e.s
}

func New(text string) error {
    return &errorString{text}
}
```

在Go源码内部，errorSring类型实现了errox接口，并使用一个可导出的方法返回一个新的error实例

构造error最简单的方法是使用 errors.New()方法

直接使用errors.New()方法比较少见，因为fmt包额外提供了fmt.Erorrf()封装函数，支持字符串格式化功能

类型断言

类型断言是一个作用在接口值上的操作，写作 x.(T) 其中x为接口类型变量，T则为要断言的类型

x.(T)

若T为具体类型，那么类型断言会检查x的动态类型是否为T。若断言成功，结果即为x的动态值，类型当然就是T

若T为接口类型，那么类型断言会检查x的动态类型是否满足T。若断言成功，结果仍为接口值，不过此时的类型为接口类型T

类型分支

当某个函数接收 interface{} 类型参数时，需要在函数内部来确定其真实的动态类型，若想要实现此需求，可使用 x.(type) 类型分支

```
func Bar(v interface{}) {
    switch v.(type) {
    case string:
        fmt.Println("v is string")
    case int:
        fmt.Println("v is int")
    default:
        panic("unknown type")
    }
}
```

需要注意的是，x.(type)仅能在switch中进行使用