接口interface

- 接口是一个或多个方法签名的集合
- 只要某个类型拥有该接口的所有方法签名，即算实现该接口，无需显示声明实现了哪个接口，这称为 Structural Typing
- 接口只有方法声明，没有实现，没有数据字段
- 接口可以匿名嵌入其它接口，或嵌入到结构中
- 将对象赋值给接口时，会发生拷贝，而接口内部存储的是指向这个复制品的指针，既无法修改复制品的状态，也无法获取指针
- 只有当接口存储的类型和对象都为nil时，接口才等于nil
- 接口调用不会做receiver的自动转换
- 接口同样支持匿名字段方法
- 接口也可实现类似OOP中的多态
- 空接口可以作为任何类型数据的容器

接口是一个或者多个方法签名的集合，因此这个接口只有方法声明，没有实现，而且也没有数据字段。也就说只能够把方法的声明放到这个接口声明当中。

Go语言实现接口比较特殊，它的官方学术的说法叫做Structural Typing，它主要实现某一个类型所拥有该接口所有方法的签名，它就算实现了该接口。不需要显示的说明该接口，比如我实现了 A接口或者B接口。

在使用的时候，如果实现了某一个接口，不需要进行一个特别的转换，直接就可以使用，不管你其它是否同时实现了其它的接口，都可以拿来使用，这段感觉像什么呢？有一端感觉就像USB接口。另外一端是针对各式各样不同的接口，那么同样是一根线，当需要连接电视的时候，只需要判断我有连接电视的接口就可以了。其它能连接手机，电话，或者其他什么东西，都无所谓。当需要连接电话的时候，并不需要管它是否能连接电视，只需要知道我能够连接那台手机，这就可以了。那么接口就是这样使用的感觉。

怎么来定义这个接口
定义语句：
同样是 **type 关键字**，然后是创建名称，之后是 **interface 关键字**。它里面有俩个方法

```go
type USB interface { // 定义一个名为USB的接口
    Name() string // 名为USB接口中的方法，Name()方法返回的是USB接口名称,返回类型为strint
    Connect()     // 名为USB接口中的方法，Connect()方法返回的是连接方法
}
```

这样就算把一个接口定义完成。

定义完接口，怎么样用一个结构对它进行一个实现呢？
比如说定一个 PhoneConnecter struct结构。就让它实现调用USB接口，只有能实现这个接口，才能算是连接某一台设备了。
代码如下：

```go
package main

import (
    "fmt"
)

type USB interface { // 定义一个名为USB的接口
    Name() string // 名为USB接口中的方法，Name()方法返回的是USB接口名称,返回类型为strint
    Connect()     // 名为USB接口中的方法，Connect()方法返回的是连接方法
}

type PhoneConnecter struct { // 定义名为PhoneConnecter的结构来实现对USB接口的调用
    name string // 在PhoneConnecter结构中定一个字段为strint类型的name内部字段参数属性
}

func (pc PhoneConnecter) Name() string { // 为PhoneConnecter结构添加名为Name()方法,并且返回类型为string
    return pc.name // 返回PhoneConnecter的名称，这里就是pc.name的值
}

func (pc PhoneConnecter) Connect() { // 为PhoneConnecter结构添加名为Connect()方法
    fmt.Println("Connect:", pc.name) // 打印Connect的结果
```

```
}

func main() {
    var usb_name USB                // 声明定义一个变量,变量类型是USB类型,它就是一个接口.
    usb_name = PhoneConnecter{}     // 初始化PhoneConnecter结构，赋值给变量usb_name
    usb_name.name = "PhoneConnecter" // 把usb_name（PhoneConnecter.name）中的name赋值PhoneConnecter
    usb_name.Connect()              // 调用Connect()方法
}
```

```
/usr/local/go/bin/go build -i [/home/jiemin/code/GOlang/go/src/interface]
# _/home/jiemin/code/GOlang/go/src/interface
./interface.go:27: usb_name.name undefined (type USB has no field or method name, but does have Name)
错误: 进程退出代码 2.
```

代码报错，发现需要把`usb_name.name = "PhoneConnecter"`这个赋值放到`usb_name = PhoneConnecter{}`中进行初始化。
修改代码如下：

```
package main

import (
    "fmt"
)

type USB interface { // 定义一个名为USB的接口
    Name() string // 名为USB接口中的方法，Name()方法返回的是USB接口名称,返回类型为strint
    Connect()     // 名为USB接口中的方法，Connect()方法返回的是连接方法
}

type PhoneConnecter struct { // 定义名为PhoneConnecter的结构来实现对USB接口的调用
    name string // 在PhoneConnecter结构中定一个字段为strint类型的name内部字段参数属性
}

func (pc PhoneConnecter) Name() string { // 为PhoneConnecter结构添加名为Name()方法,并且返回类型为string
    return pc.name // 返回PhoneConnecter的名称，这里就是pc.name的值
}

func (pc PhoneConnecter) Connect() { // 为PhoneConnecter结构添加名为Connect()方法
    fmt.Println("Connect:", pc.name) // 打印Connect的结果
}

func main() {
    var usb_name USB                                // 声明定义一个变量,变量类型是USB类型,它就是一个接口.
    usb_name = PhoneConnecter{name: "PhoneConnecter"} // 初始化PhoneConnecter结构，赋值给变量usb_name
    usb_name.Connect()                              // 调用Connect()方法
}
```

```
/home/jiemin/code/GOlang/go/src/interface/interface [/home/jiemin/code/GOlang/go/src/interface]
Connect: PhoneConnecter
成功: 进程退出代码 0.
```

正常输出结果。刚才代码不能使用原因是USB接口没有字段。也就说取不到 `PhoneConnecter` 结构中的name字段。如果使用对`PhoneConnecter` 结构字面值对name进行`usb_name = PhoneConnecter{name: "PhoneConnecter"}`初始化，它就自动的将 `PhoneConnecter` 放到(赋值)给 `PhoneConnecter` 结构当中的 `name` 字段。接下来就看到，调用 `usb_name.Connect()` 方法后，就会打印输出 `fmt.Println("Connect:", pc.name)` 的值。在这里，使用了 `var usb_name USB` 中的 `usb_name` 的类型是 `USB` ，由于这里的 `PhoneConnecter` 实现了 `USB` 的这个接口，因此就可以将它赋值给它。

实际上我们可以进行这样的简写：

```
func main() {
        usb_name := PhoneConnecter{name: "PhoneConnecter"} // 初始化PhoneConnecter结构，赋值给变量usb_name
        usb_name.Connect()                                 // 调用Connect()方法
}
```

其实这样的简写，就不能够很明确的它是否真的实现这个USB的接口。可能是我们一厢情愿搞错了。

接下来就会有一个名为方法Disconnect()。它就一个要求，要求什么呢？要求传进来一个实现USB接口的变量

```
package main

import (
    "fmt"
)

type USB interface { // 定义一个名为USB的接口
    Name() string // 名为USB接口中的方法，Name()方法返回的是USB接口名称,返回类型为strint
    Connect()     // 名为USB接口中的方法，Connect()方法返回的是连接方法
}

type PhoneConnecter struct { // 定义名为PhoneConnecter的结构来实现对USB接口的调用
```

```
    name string // 在PhoneConnecter结构中定一个字段为strint类型的name内部字段参数属性
}

func (pc PhoneConnecter) Name() string { // 为PhoneConnecter结构添加名为Name()方法,并且返回类型为string
    return pc.name // 返回PhoneConnecter的名称，这里就是pc.name的值
}

func (pc PhoneConnecter) Connect() { // 为PhoneConnecter结构添加名为Connect()方法
    fmt.Println("Connected:", pc.name) // 打印Connect的结果
}

func main() {
    usb_name := PhoneConnecter{
        name: "PhoneConnecter",
    } // 初始化PhoneConnecter结构，赋值给变量usb_name,其实这样的简写，就不能够很明确的它是否真的实现这个USB的接口。
    usb_name.Connect()   // 调用Connect()方法
    Disconnect(usb_name) // 调用Disconnect()方法，将usb_name传递进去
}

func Disconnect(usb USB) { // 一个名为方法Disconnect()。它就一个要求，要求什么呢？要求传进来一个实现USB接口的变量
    fmt.Println("Disconnected.") // 打印Disconnected.
}
```

```
/home/jiemin/code/GOlang/go/src/interface/interface [/home/jiemin/code/GOlang/go/src/interface]
Connected: PhoneConnecter
Disconnected.
成功: 进程退出代码 0.
```

可以看到输出了Disconnected. 。它就成功的调用了 `Disconnect()` 的方法。这个`Disconnect`函数成功调用了，我们传递进去的要求 它的`Disconnect(usb USB)` 当中的类型是 `USB` 这个接口，然后将 `PhoneConnecter` 传递进去，但是它没有报错而且成功的执行了，这就说明`PhoneConnecter` 成功实现了 `USB` 这个接口 ，所以说才能成功的将它调用进去。

然后既然讲到这个接口以及这个结构，就要谈到一个和嵌入结构类似的嵌入接口

嵌入接口

虽然说接口只包含这一个方法，当时当需要比较一个复杂一些 多层关系的时候，就会用到嵌入接口。那么嵌入接口怎么用呢，实际上是非常简单的。

比如说USB结构本质上是什么呢？是一个Connecter，是一个连接器，只不过它的标准叫做USB，那么这里有一个Connecter这样一个接口，它有一个什么方法呢？显然就是Connect() 这一个方法。这个时候就可以把Connecter作为嵌入接口放到USB结构当中。那么显然的，这个USB接口就拥有了Connecter这个方法Connect()。

具体代码：

```
package main

import (
    "fmt"
)

type USB interface { // 定义一个名为USB的接口
    Name() string // 名为USB接口中的方法，Name()方法返回的是USB接口名称,返回类型为strint
    Connecter     // 把Connecter作为嵌入接口放到USB结构当中。USB接口就拥有了Connecter这个方法Connect()
}

type Connecter interface { // 定义一个名为Connecter的接口
    Connect() // 名为Connecter接口中的方法，Connect()方法
}

type PhoneConnecter struct { // 定义名为PhoneConnecter的结构来实现对USB接口的调用
    name string // 在PhoneConnecter结构中定一个字段为strint类型的name内部字段参数属性
}

func (pc PhoneConnecter) Name() string { // 为PhoneConnecter结构添加名为Name()方法,并且返回类型为string
    return pc.name // 返回PhoneConnecter的名称，这里就是pc.name的值
}

func (pc PhoneConnecter) Connect() { // 为PhoneConnecter结构添加名为Connect()方法
    fmt.Println("Connected:", pc.name) // 打印Connect的结果
}

func main() {
    usb_name := PhoneConnecter{
```

```
        name: "PhoneConnecter",
    } // 初始化PhoneConnecter结构，赋值给变量usb_name,其实这样的简写，就不能够很明确的它是否真的实现这个USB的接口。
    usb_name.Connect()   // 调用Connect()方法
    Disconnect(usb_name) // 调用Disconnect()方法，将usb_name传递进去
}

func Disconnect(usb USB) { // 一个名为方法Disconnect()。它就一个要求，要求什么呢？要求传进来一个实现USB接口的变量
    fmt.Println("Disconnected.") // 打印Disconnected.
}
```

```
/home/jiemin/code/GOlang/go/src/interface/interface [/home/jiemin/code/GOlang/go/src/interface]
Connected: PhoneConnecter
Disconnected.
成功: 进程退出代码 0.
```

这个程序成功的运行，也就证明了我们成功使用了嵌入接口。

现在回到Disconnect这个方法当中来。这段代码成功的输出了Disconnected.。但是不知道谁断开连接？这时候就要判断，由于我们这个结构

```
type PhoneConnecter struct { // 定义名为PhoneConnecter的结构来实现对USB接口的调用
        name string // 在PhoneConnecter结构中定一个字段为strint类型的name内部字段参数属性
}
```

当中的name，这个USB结构它是没有字段的。所以它暂时取不到这个name，那我们可以用一个什么方法可以取到呢？那么最简单的方法呢就是通过这个判断一个类型，也就说判断这个放在USB接口当中实际上一个什么结构，那么这个就要用到一个类型判断。怎么判断呢？

```
type USB interface { // 定义一个名为USB的接口
    Name() string // 名为USB接口中的方法，Name()方法返回的是USB接口名称,返回类型为strint
    Connecter     // 把Connecter作为嵌入接口放到USB结构当中。USB接口就拥有了Connecter这个方法Connect()
}

type Connecter interface { // 定义一个名为Connecter的接口
    Connect() // 名为Connecter接口中的方法，Connect()方法
}

type PhoneConnecter struct { // 定义名为PhoneConnecter的结构来实现对USB接口的调用
    name string // 在PhoneConnecter结构中定一个字段为strint类型的name内部字段参数属性
}

func (pc PhoneConnecter) Name() string { // 为PhoneConnecter结构添加名为Name()方法,并且返回类型为string
    return pc.name // 返回PhoneConnecter的名称，这里就是pc.name的值
}

func (pc PhoneConnecter) Connect() { // 为PhoneConnecter结构添加名为Connect()方法
    fmt.Println("Connected:", pc.name) // 输出Connected: 打印一下pc.name的值
}

func Disconnect(usb USB) { // 一个名为方法Disconnect()。它就一个要求，要求什么呢？要求传进来一个实现USB接口的变量
    if pc, ok := usb.(PhoneConnecter); ok { //
        fmt.Println("Disconnected:", pc.name) // 输出Disconnected: 打印一下pc.name的值
        return                                // 返回一个返回值
    }
    fmt.Println("Unknown decive.") // 打印Unknown decive.
}
```

这里也是一个OK patemu 的模式。比如：这里取到一个pc,然后有一个ok，然后这里在输入usb，然后进行一个类型判断，它是这样一个格式: usb.()

括号当中要放入要判断一个类型，比如这里要判断 PhoneConnecter ，然后是一个**分号(;)**,然后在判断ok是否成立。如果成立，就输出`fmt.Println("Disconnected:", pc.name)`。如果ok不成立就输出`fmt.Println("Unknown decive.")`

具体代码如下：

```
package main

import (
    "fmt"
)

type USB interface { // 定义一个名为USB的接口
    Name() string // 名为USB接口中的方法，Name()方法返回的是USB接口名称,返回类型为strint
    Connecter     // 把Connecter作为嵌入接口放到USB结构当中。USB接口就拥有了Connecter这个方法Connect()
}

type Connecter interface { // 定义一个名为Connecter的接口
    Connect() // 名为Connecter接口中的方法，Connect()方法
```

```
}

type PhoneConnecter struct { // 定义名为PhoneConnecter的结构来实现对USB接口的调用
    name string // 在PhoneConnecter结构中定一个字段为strint类型的name内部字段参数属性
}

func (pc PhoneConnecter) Name() string { // 为PhoneConnecter结构添加名为Name()方法,并且返回类型为string
    return pc.name // 返回PhoneConnecter的名称，这里就是pc.name的值
}

func (pc PhoneConnecter) Connect() { // 为PhoneConnecter结构添加名为Connect()方法
    fmt.Println("Connected:", pc.name) // 输出Connected: 打印一下pc.name的值
}

func main() {
    usb_name := PhoneConnecter{
        name: "PhoneConnecter",
    } // 初始化PhoneConnecter结构，赋值给变量usb_name,其实这样的简写，就不能够很明确的它是否真的实现这个USB的接口。
    usb_name.Connect()   // 调用Connect()方法
    Disconnect(usb_name) // 调用Disconnect()方法，将usb_name传递进去
}

func Disconnect(usb USB) { // 一个名为方法Disconnect()。它就一个要求，要求什么呢？要求传进来一个实现USB接口的变量
    if pc, ok := usb.(PhoneConnecter); ok {//取pc，ok是类型判断,usb.()括号里面放入PhoneConnecter类型,然后判断ok是否成立
        fmt.Println("Disconnected:", pc.name) // 输出Disconnected: 打印一下pc.name的值
        return                                // 返回一个返回值
    }
    fmt.Println("Unknown decive.") // 打印Unknown decive.
}
```

```
/home/jiemin/code/GOlang/go/src/interface/interface [/home/jiemin/code/GOlang/go/src/interface]
Connected: PhoneConnecter
Disconnected: PhoneConnecter
成功: 进程退出代码 0.
```

就看到首先调用 `usb_name.Connect()` 它输出了 `fmt.Println("Connected:", pc.name)` 。然后又调用了 `Disconnect(usb_name)` 方法。那么它呢，这里

```
func Disconnect(usb USB) {
        if pc, ok := usb.(PhoneConnecter); ok {
                fmt.Println("Disconnected:", pc.name)
                return
        }
        fmt.Println("Unknown decive.")
}
```

进行了一个类型判断，它是不是传进来的是不是真的 `PhoneConnecter` ，它传进来之后，我们进行一个`if pc, ok := usb.(PhoneConnecter); ok {}`判断。判断之后，得出来一个True，它`usb.(PhoneConnecter)`就知道了`usb`，实际上就是 `PhoneConnecter` ，然后它就输出`fmt.Println("Disconnected:", pc.name)` ，它直接 `return` ，它就不用输出下面这个`fmt.Println("Unknown decive.")` 语句了。这个就是一个简单的类型断言。

那么有这个结构，那么go语言当中没有继承概念，那么go语言当中也有类似的存在呢？
同样是通过是接口实现的。在go语言当中不需要去显示去声明实现了哪一个接口，而是说有了它的接口定义的方法，只要和它的方法签名相同，它就默认你实现了这一个接口。不需要告诉编译器。这个好处在这里就稍微体现了一下。

如果说有一个`empty`接口

```
type empty interface{}
```

```
type empty interface{
}
```

它里面什么方法都没有。那说明了什么？那说明了不管我没有没方法，实际上都实现了`empty`这一个接口，因为它没有方法， 任何东西任何类型都可以说没有方法，然后用其它的方法，由于go的接口实现原理：它不管你又没有其它的方法，只要符合接口的标准，它就实现了接口。因此说，go语言当中所有的类型都实现了这个空接口。任何类型都可以感觉上是继承自它而来。要注意go语言当中没有继承，这只是给与自己的一个感觉，类似继承的方式，可以假定的这样认为。

这时候就可以将 `Disconnect()` 方法改造一下了。改造成什么呢？同样是这个 `func Disconnect(usb USB) {` 当中的 `USB` ，但是类型是什么呢？我们把它改造的更加广泛一点， 同意它传进来一个`interface{}`空接口，这个就可以表示 `Disconnect()` 可以将任何类型都传入进来。因为它是一个空接口，任何类型都实现了空接口；然后在这个方法内部进行一个判断。这个时候同样可以使用ok patemu，来试一下。

```
package main

import (
```

```
    "fmt"
)

type empty interface{}

type USB interface { // 定义一个名为USB的接口
    Name() string // 名为USB接口中的方法，Name()方法返回的是USB接口名称,返回类型为strint
    Connecter     // 把Connecter作为嵌入接口放到USB结构当中。USB接口就拥有了Connecter这个方法Connect()
}

type Connecter interface { // 定义一个名为Connecter的接口
    Connect() // 名为Connecter接口中的方法，Connect()方法
}

type PhoneConnecter struct { // 定义名为PhoneConnecter的结构来实现对USB接口的调用
    name string // 在PhoneConnecter结构中定一个字段为strint类型的name内部字段参数属性
}

func (pc PhoneConnecter) Name() string { // 为PhoneConnecter结构添加名为Name()方法,并且返回类型为string
    return pc.name // 返回PhoneConnecter的名称，这里就是pc.name的值
}

func (pc PhoneConnecter) Connect() { // 为PhoneConnecter结构添加名为Connect()方法
    fmt.Println("Connected:", pc.name) // 输出Connected: 打印一下pc.name的值
}

func main() {
    usb_name := PhoneConnecter{
        name: "PhoneConnecter",
    } // 初始化PhoneConnecter结构，赋值给变量usb_name,其实这样的简写，就不能够很明确的它是否真的实现这个USB的接口。
    usb_name.Connect()   // 调用Connect()方法
    Disconnect(usb_name) // 调用Disconnect()方法，将usb_name传递进去
}

func Disconnect(usb interface{}) { //传递进来一个空接口，表示Disconnect可以将任何类型都传入进来。因为它是一个空接口，任何类型都实现了空接
    if pc, ok := usb.(PhoneConnecter); ok { // 取pc，ok是类型判断，usb.()括号里面放入PhoneConnecter类型。然后判断ok是否成立
        fmt.Println("Disconnected:", pc.name) // 输出Disconnected: 打印一下pc.name的值
        return                                // 返回一个返回值
    }
    fmt.Println("Unknown decive.") // 打印Unknown decive.
}
```

```
/home/jiemin/code/GOlang/go/src/interface/interface [/home/jiemin/code/GOlang/go/src/interface]
Connected: PhoneConnecter
Disconnected: PhoneConnecter
成功: 进程退出代码 0.
```

和刚才输出的结果是一样的。但是这个空接口和这个一些其它的接口有一些不同之处。因为空接口它可以允许传递进来的类型太多了，可以传递一个int，strint等等，什么类型都可以传递进来。那么这个时候，我们如果都是用okpateme，都是 `pc, ok := usb.(PhoneConnecter); ok {}` 这样长的句子来一个判断的话，就有一些说显得不方便了，有一些资源的浪费。因为比如说我们当使用传递进来的USB的时候，我们知道它只可能传递进来的是 `PhoneConnecter` 或者什么什么的`Connecter` ，如果数量比较少，而且要进行后续判断的时候，像`pc, ok := usb.(PhoneConnecter); ok {}` 这里需要用到 ok 这个bool值进行一个判断的时候，我们可以使用这个ok pateme。但是当你可以接收的类型实在是太多的时候，而恰好需要对某一种类型做出分别处理的时候，更需要一种更加搞笑的判断方法，也更加简便的判断方法。

在go语言当中叫做 **type switch**，也就说说使用一种switch结构，来进行一个类型的判断。那么比如说这个

```
func Disconnect(usb interface{}) { //传递进来一个空接口，表示Disconnect可以将任何类型都传入进来。因为它是一个空接口，任何类型都实现了空接口
        if pc, ok := usb.(PhoneConnecter); ok { // 取pc，ok是类型判断，usb.()括号里面放入PhoneConnecter类型。然后判断ok是否成立
                fmt.Println("Disconnected:", pc.name) // 输出Disconnected: 打印一下pc.name的值
                return                                // 返回一个返回值
        }
        fmt.Println("Unknown decive.") // 打印Unknown decive.
}
```

语言怎么进行一个修改呢？

首先也是一个**switch 关键字** ， 然后一个 **v:=** 这里v是局部变量，也就是value。这里是usb.()，与前面这个`usb.(PhoneConnecter)`写法 不同的是，不写入 `PhoneConnecter` ，我们不用去猜测它可能是什么类型。我们只写入一个 **type** 也就是说让系统去猜它是什么类型。**`v := usb.(type)` 当 `usb.(type)` 判断完之后，这里`v := usb.(type)`的v就担当着pc这个值；就是说是类型变量的名称**。接下里就是一个 **case** ，比如说这里的case 就需要判断了，判断 `PhoneConnecter` 。如果当它是 `PhoneConnecter` 的时候，这里输出 `fmt.Println("Disconnected:", pc.name)` ，前面讲过的知识，知道它还有一个 **default:**

default 就是把这个`fmt.Println("Unknown decive.")`语句放进去。这时候来看一下具体代码怎么写：

```
func Disconnect(usb interface{}) { //传递进来一个空接口，表示Disconnect可以将任何类型都传入进来。因为它是一个空接口。任何类型都实现了空接
    /* 首先也是一个switch 关键字,然后一个 v:=  这里是局部变量，这里是usb.()
    不写入 PhoneConnecter,我们不用去猜测它可能是什么类型。我们只写入一个usb.(type)也就是说让系统去猜它是什么类型
```

```
    接下里就是一个 case,比如说这里的case就需要判断PhoneConnecter
    如果当它是PhoneConnecter的时候,这里输出 fmt.Println("Disconnected:", pc.name)
    根据前面讲过的知识，知道它还有一个 default:
    default 就是把这个fmt.Println("Unknown decive.")语句放进去 */
    switch v := usb.(type) {
    case PhoneConnecter:
        fmt.Println("Disconnected:", v.name) // 输出Disconnected: 打印一下v.name的值
    default:
        fmt.Println("Unknown decive.") // 打印Unknown decive.
    }
}
```

详细代码运行结果：

```
package main

import (
    "fmt"
)

type empty interface{}

type USB interface { // 定义一个名为USB的接口
    Name() string // 名为USB接口中的方法，Name()方法返回的是USB接口名称,返回类型为strint
    Connecter     // 把Connecter作为嵌入接口放到USB结构当中。USB接口就拥有了Connecter这个方法Connect()
}

type Connecter interface { // 定义一个名为Connecter的接口
    Connect() // 名为Connecter接口中的方法，Connect()方法
}

type PhoneConnecter struct { // 定义名为PhoneConnecter的结构来实现对USB接口的调用
    name string // 在PhoneConnecter结构中定一个字段为strint类型的name内部字段参数属性
}

func (pc PhoneConnecter) Name() string { // 为PhoneConnecter结构添加名为Name()方法,并且返回类型为string
    return pc.name // 返回PhoneConnecter的名称，这里就是pc.name的值
}

func (pc PhoneConnecter) Connect() { // 为PhoneConnecter结构添加名为Connect()方法
    fmt.Println("Connected:", pc.name) // 输出Connected: 打印一下pc.name的值
}

func main() {
    usb_name := PhoneConnecter{
        name: "PhoneConnecter",
    } // 初始化PhoneConnecter结构，赋值给变量usb_name,其实这样的简写，就不能够很明确的它是否真的实现这个USB的接口。
    usb_name.Connect()   // 调用Connect()方法
    Disconnect(usb_name) // 调用Disconnect()方法，将usb_name传递进去
}

func Disconnect(usb interface{}) { //传递进来一个空接口，表示Disconnect可以将任何类型都传入进来。因为它是一个空接口，任何类型都实现了空接
    /* 首先也是一个switch 关键字,然后一个 v:=  这里是局部变量，这里是usb.()
    不写入 PhoneConnecter,我们不用去猜测它可能是什么类型。我们只写入一个usb.(type)也就是说让系统去猜它是什么类型
    接下里就是一个 case,比如说这里的case就需要判断PhoneConnecter
    如果当它是PhoneConnecter的时候,这里输出 fmt.Println("Disconnected:", pc.name)
    根据前面讲过的知识，知道它还有一个 default:
    default 就是把这个fmt.Println("Unknown decive.")语句放进去 */
    switch v := usb.(type) {
    case PhoneConnecter:
        fmt.Println("Disconnected:", v.name) // 输出Disconnected: 打印一下v.name的值
    default:
        fmt.Println("Unknown decive.") // 打印Unknown decive.
    }
}
```

```
/home/jiemin/code/GOlang/go/src/interface/interface  [/home/jiemin/code/GOlang/go/src/interface]
Connected: PhoneConnecter
Disconnected: PhoneConnecter
成功: 进程退出代码 0
```

看到它成功的输出了结果。这个就是一个 type switch 的用法。一般来说都是针对于，当参数接收的是空接口的时候，这种用的很多，可以根据自己的一个实际情况，可以选择用okpateme或者 type switch 。

那么既然像类型都有互相转换的过程，那么我们的接口是不是也能进行一个相互转换呢？

接口的相互转换要注意一点：只能够向拥有超级的接口转换为它子集的接口。什么意思呢？

比如说

```
type USB interface { // 定义一个名为USB的接口
    Name() string // 名为USB接口中的方法，Name()方法返回的是USB接口名称,返回类型为strint
    Connecter     // 把Connecter作为嵌入接口放到USB结构当中。USB接口就拥有了Connecter这个方法Connect()
}

type Connecter interface { // 定义一个名为Connecter的接口
    Connect() // 名为Connecter接口中的方法，Connect()方法
}
```

这里USB和Connecter，USB由于嵌入这个Connecter，它不仅包含了Connecter这个接口当中的Connect()方法，它还有着另外的方法叫做 Name() string ，而这个Connecter 它只包含一个方法，叫做 Connect() 方法。所以说理论上讲这个 Connecter 不能转换 USB ，但是 USB 可以转换为这个 Connecter 。接下来我们实践一下。

```
package main

import (
    "fmt"
)

type empty interface{}

type USB interface { // 定义一个名为USB的接口
    Name() string // 名为USB接口中的方法，Name()方法返回的是USB接口名称,返回类型为strint
    Connecter     // 把Connecter作为嵌入接口放到USB结构当中。USB接口就拥有了Connecter这个方法Connect()
}

type Connecter interface { // 定义一个名为Connecter的接口
    Connect() // 名为Connecter接口中的方法，Connect()方法
}

type PhoneConnecter struct { // 定义名为PhoneConnecter的结构来实现对USB接口的调用
    name string // 在PhoneConnecter结构中定一个字段为strint类型的name内部字段参数属性
}

func (pc PhoneConnecter) Name() string { // 为PhoneConnecter结构添加名为Name()方法,并且返回类型为string
    return pc.name // 返回PhoneConnecter的名称，这里就是pc.name的值
}

func (pc PhoneConnecter) Connect() { // 为PhoneConnecter结构添加名为Connect()方法
    fmt.Println("Connected:", pc.name) // 输出Connected: 打印一下pc.name的值
}

func main() {
    pc := PhoneConnecter{name: "PhoneConnecter"} // 声明PhoneConnecter结构当中name属性的值，赋值给pc
    var usb_name Connecter                       // 声明usb_name类型是Connecter
    usb_name = Connecter(pc)                     // 这个usb_name是通过强制类型转换，将PhoneConnecter转换Connecter类型
    usb_name.Connect()                           // 调用Connect()方法
}

func Disconnect(usb interface{}) { //传递进来一个空接口，表示Disconnect可以将任何类型都传入进来。因为它是一个空接口，任何类型都实现了空接
    /* 首先也是一个switch 关键字,然后一个 v:=  这里是局部变量，这里是usb.()
    不写入 PhoneConnecter,我们不用去猜测它可能是什么类型。我们只写入一个usb.(type)也就是说让系统去猜它是什么类型
    接下里就是一个 case,比如说这里的case就需要判断PhoneConnecter
    如果当它是PhoneConnecter的时候,这里输出 fmt.Println("Disconnected:", pc.name)
    根据前面讲过的知识，知道它还有一个 default:
    default 就是把这个fmt.Println("Unknown decive.")语句放进去 */
    switch v := usb.(type) {
    case PhoneConnecter:
        fmt.Println("Disconnected:", v.name) // 输出Disconnected: 打印一下v.name的值
    default:
        fmt.Println("Unknown decive.") // 打印Unknown decive.
    }
}
```

```
/home/jiemin/code/GOlang/go/src/interface/interface [/home/jiemin/code/GOlang/go/src/interface]
Connected: PhoneConnecter
成功: 进程退出代码 0.
```

可以看到成功的输出了 fmt.Println("Connected:", pc.name) ，也就是说成功的将这个，其实是这个 USB ，实现了 USB interface，然后将它转换为 Connecter 的 interface ，然后它只有这一个方法叫做 Connect() 。

接下来要实现一个结构，它只实现了 `Connecter` 但是没有实现 `USB` ，这里就简单举一个例子：
叫做TVConnecter struc{}

```go
package main

import (
    "fmt"
)

type empty interface{}

type USB interface { // 定义一个名为USB的接口
    Name() string // 名为USB接口中的方法，Name()方法返回的是USB接口名称,返回类型为strint
    Connecter     // 把Connecter作为嵌入接口放到USB结构当中。USB接口就拥有了Connecter这个方法Connect()
}

type Connecter interface { // 定义一个名为Connecter的接口
    Connect() // 名为Connecter接口中的方法，Connect()方法
}

type PhoneConnecter struct { // 定义名为PhoneConnecter的结构来实现对USB接口的调用
    name string // 在PhoneConnecter结构中定一个字段为strint类型的name内部字段参数属性
}

func (pc PhoneConnecter) Name() string { // 为PhoneConnecter结构添加名为Name()方法,并且返回类型为string
    return pc.name // 返回PhoneConnecter的名称，这里就是pc.name的值
}

func (pc PhoneConnecter) Connect() { // 为PhoneConnecter结构添加名为Connect()方法
    fmt.Println("Connected:", pc.name) // 输出Connected: 打印一下pc.name的值
}

type TVConnecter struct { // 声明一个名为TVConnecter结构
    name string
}

func (tv TVConnecter) Connect() { // 创建接收TVConnecter结构为参数tv的名为Connect()方法
    fmt.Println("Connected:", tv.name) // 输出Connected: 打印一下tv.name的值
}

func main() {
    tv := TVConnecter{name: "TVConnecter"} // 声明TVConnecter结构当中name属性的值，赋值给tv
    var usb_name USB                       // 声明usb_name类型是USB
    usb_name = USB(tv)                     // 这个usb_name是通过强制类型转换，将TVConnecter转换USB类型
    usb_name.Connect()                     // 调用Connect()方法
}

func Disconnect(usb interface{}) { //传递进来一个空接口，表示Disconnect可以将任何类型都传入进来。因为它是一个空接口，任何类型都实现了空接
    /* 首先也是一个switch 关键字,然后一个 v:=  这里是局部变量，这里是usb.()
    不写入 PhoneConnecter,我们不用去猜测它可能是什么类型。我们只写入一个usb.(type)也就是说让系统去猜它是什么类型
    接下里就是一个 case,比如说这里的case就需要判断PhoneConnecter
    如果当它是PhoneConnecter的时候,这里输出 fmt.Println("Disconnected:", pc.name)
    根据前面讲过的知识，知道它还有一个 default:
    default 就是把这个fmt.Println("Unknown decive.")语句放进去 */
    switch v := usb.(type) {
    case PhoneConnecter:
        fmt.Println("Disconnected:", v.name) // 输出Disconnected: 打印一下v.name的值
    default:
        fmt.Println("Unknown decive.") // 打印Unknown decive.
    }
}
```

```
/usr/local/go/bin/go build -i [/home/jiemin/code/GOlang/go/src/interface]
# _/home/jiemin/code/GOlang/go/src/interface
./interface.go:41: cannot convert tv (type TVConnecter) to type USB:
        TVConnecter does not implement USB (missing Name method)
错误: 进程退出代码 2.
```

错误是说，TVConnecter 没有实现 USB 这个方法，所以就不能把它转换成USB，那个就是转换之间存在的问题。而我们刚才是可以将这个 `PhoneConnecter` 转换，降级转换成Connecter的。这个就是在接口转换的时候要注意的一个问题。

接口使用的注意事项：

1、将对象赋值给接口的时候会发现一个拷贝，而这个接口内部存的是指向这个复制品的指针，所以说没有办法修改这个复制品的状态，也没有办法获取复制品的指针。也就说给它之后，能够进行一些比如说一些非常复制之间的操作，就没有办法修改它。然后当修改这个原来的的struct的时候，这个复制品的状态是不会改变的。就和我们传参时候是一个道理。这里就简单演示一下复制品的问题。

```go
package main

import (
    "fmt"
)

type empty interface{}

type USB interface { // 定义一个名为USB的接口
    Name() string // 名为USB接口中的方法，Name()方法返回的是USB接口名称,返回类型为strint
    Connecter     // 把Connecter作为嵌入接口放到USB结构当中。USB接口就拥有了Connecter这个方法Connect()
}

type Connecter interface { // 定义一个名为Connecter的接口
    Connect() // 名为Connecter接口中的方法，Connect()方法
}

type PhoneConnecter struct { // 定义名为PhoneConnecter的结构来实现对USB接口的调用
    name string // 在PhoneConnecter结构中定一个字段为strint类型的name内部字段参数属性
}

func (pc PhoneConnecter) Name() string { // 为PhoneConnecter结构添加名为Name()方法,并且返回类型为string
    return pc.name // 返回PhoneConnecter的名称，这里就是pc.name的值
}

func (pc PhoneConnecter) Connect() { // 为PhoneConnecter结构添加名为Connect()方法
    fmt.Println("Connected:", pc.name) // 输出Connected: 打印一下pc.name的值
}

func main() {
    pc := PhoneConnecter{name: "PhoneConnecter"} // 声明PhoneConnecter结构当中name属性的值，赋值给pc
    var usb_name Connecter                       // 声明usb_name类型是Connecter
    usb_name = Connecter(pc)                     // 这个usb_name是通过强制类型转换，将PhoneConnecter转换Connecter类型
    usb_name.Connect()                           // 调用Connect()方法

    pc.name = "PC"     // 修改pc.name的值
    usb_name.Connect() // 调用Connect()方法
}

func Disconnect(usb interface{}) { //传递进来一个空接口，表示Disconnect可以将任何类型都传入进来。因为它是一个空接口，任何类型都实现了空接
    /* 首先也是一个switch 关键字,然后一个 v:=  这里是局部变量，这里是usb.()
    不写入 PhoneConnecter,我们不用去猜测它可能是什么类型。我们只写入一个usb.(type)也就是说让系统去猜它是什么类型
    接下里就是一个 case,比如说这里的case就需要判断PhoneConnecter
    如果当它是PhoneConnecter的时候,这里输出 fmt.Println("Disconnected:", pc.name)
    根据前面讲过的知识，知道它还有一个 default:
    default 就是把这个fmt.Println("Unknown decive.")语句放进去 */
    switch v := usb.(type) {
    case PhoneConnecter:
        fmt.Println("Disconnected:", v.name) // 输出Disconnected: 打印一下v.name的值
    default:
        fmt.Println("Unknown decive.") // 打印Unknown decive.
    }
}
```

```
/home/jiemin/code/GOlang/go/src/interface/interface  [/home/jiemin/code/GOlang/go/src/interface]
Connected: PhoneConnecter
Connected: PhoneConnecter
成功: 进程退出代码 0.
```

看到它完全就无视我们对原来`pc.name = "PC"`这个对象的修改。因此说它拿到的是一个复制品，是一个拷贝。

2、只有当接口存储的类型和对象，俩个要求都属于nil的时候，接口才等于nil。这个怎么来证明呢？

```go
package main

```

```go
import (
    "fmt"
)

type empty interface{}

type USB interface { // 定义一个名为USB的接口
    Name() string // 名为USB接口中的方法，Name()方法返回的是USB接口名称,返回类型为strint
    Connecter     // 把Connecter作为嵌入接口放到USB结构当中。USB接口就拥有了Connecter这个方法Connect()
}

type Connecter interface { // 定义一个名为Connecter的接口
    Connect() // 名为Connecter接口中的方法，Connect()方法
}

type PhoneConnecter struct { // 定义名为PhoneConnecter的结构来实现对USB接口的调用
    name string // 在PhoneConnecter结构中定一个字段为strint类型的name内部字段参数属性
}

func (pc PhoneConnecter) Name() string { // 为PhoneConnecter结构添加名为Name()方法,并且返回类型为string
    return pc.name // 返回PhoneConnecter的名称，这里就是pc.name的值
}

func (pc PhoneConnecter) Connect() { // 为PhoneConnecter结构添加名为Connect()方法
    fmt.Println("Connected:", pc.name) // 输出Connected: 打印一下pc.name的值
}

func main() {
    var nu_interface interface{}     // 声明nu_interface是一个接口,它本身存储的是nil,什么都没有存
    fmt.Println(nu_interface == nil) // 判断nu_interface是否等于nil

    var p *int = nil                 // 创建指向int类型的指针参数p,并且把nil赋值给p
    nu_interface = p                 // 这个nu_interface指向p,它存了一个指向nil,指向对象是nil,但它指向存储的类型是nil,它是一个指针
    fmt.Println(nu_interface == nil) // 判断nu_interface是否等于nil
}

func Disconnect(usb interface{}) { //传递进来一个空接口，表示Disconnect可以将任何类型都传入进来。因为它是一个空接口，任何类型都实现了空接
    /* 首先也是一个switch 关键字,然后一个 v:=  这里是局部变量，这里是usb.()
    不写入 PhoneConnecter,我们不用去猜测它可能是什么类型。我们只写入一个usb.(type)也就是说让系统去猜它是什么类型
    接下里就是一个 case,比如说这里的case就需要判断PhoneConnecter
    如果当它是PhoneConnecter的时候,这里输出 fmt.Println("Disconnected:", pc.name)
    根据前面讲过的知识，知道它还有一个 default:
    default 就是把这个fmt.Println("Unknown decive.")语句放进去 */
    switch v := usb.(type) {
    case PhoneConnecter:
        fmt.Println("Disconnected:", v.name) // 输出Disconnected: 打印一下v.name的值
    default:
        fmt.Println("Unknown decive.") // 打印Unknown decive.
    }
}
```

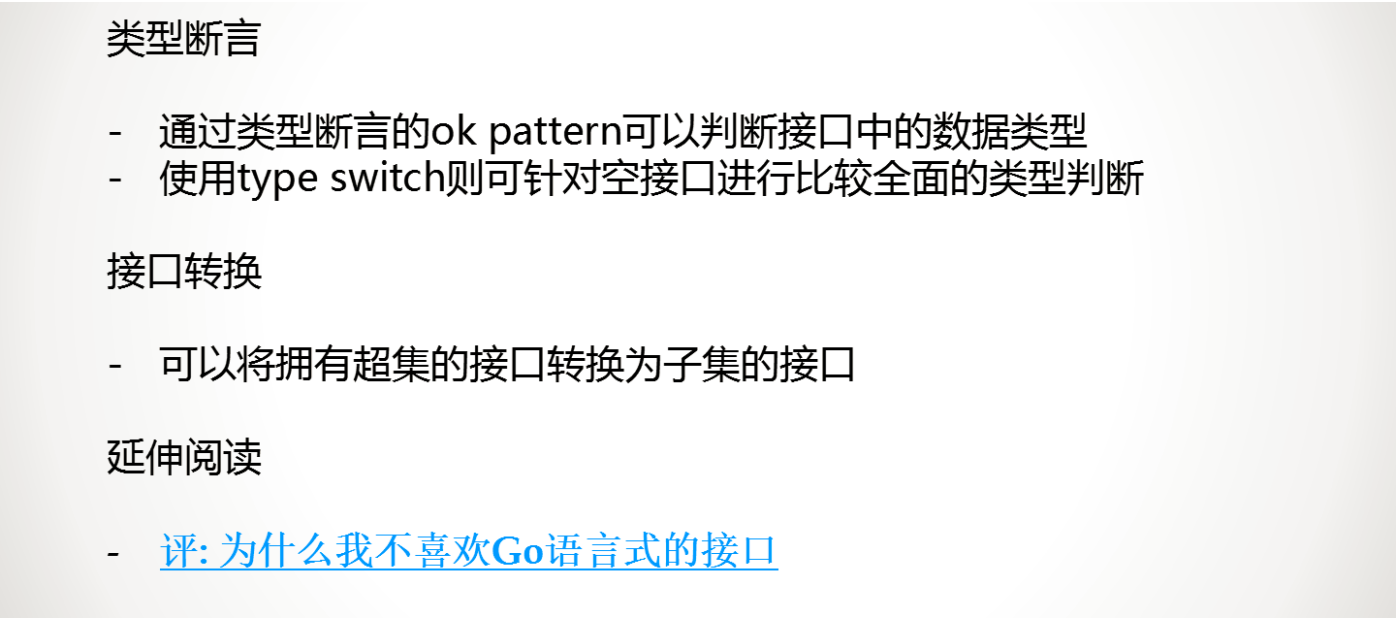
```
/home/jiemin/code/GOlang/go/src/interface/interface  [/home/jiemin/code/GOlang/go/src/interface]
true
false
成功: 进程退出代码 0.
```

3、接口调用不会做receiver的自动转换。 也就是说调用的时候，它接收的是一个指针类型，就必须传递一个指针给它。它接收的不是指针就不能传给指针给它。这个和调用结构的方法是不同的。结构它会进行一个自动的转换。那么为什么不能够进行一个自动的转换呢？这里就设计到一个方法集的问题。那么方法集的问题可以到官网查看一下。

那么简单的讲就是 指针的方法的方法集包含了不是指针传递进来的receiver这一个方法集。也就是如果传递进来的是一个指针，它既可以调用它这个receiver是指针的方法，也可以调用它这个receiver不是指针的方法。但是如果传递进去是非指针的方法集，就是说值拷贝的方法集，它是不能够调用这个指针的方法集，而是只能够调用属于自己的那一部分。这个就是为什么不对接口的调用不做receiver的自动转换。

4、接口同样支持匿名字段方法，使用方法和匿名结构是一样的。接口和结构实际上很多地方是感觉想通的，由于讲结构它是可以实现类似面向对象编程(OOP)中的多态，那么接口也可以实现类似的方法，但是它并没有这个继承的概念，这里要注意。

5、空接口可以作为任何类型的容器



最近在Go语言的QQ群里看到关于图灵社区有牛人老赵吐槽许式伟《Go语言编程》的各种争论.

我之前也看了老赵吐槽许式伟《Go语言编程》的文章, 当时想老赵如果能将许大书中不足部分补充完善了也是一个好事情. 因此, 对老赵的后续文章甚是期待.

谁知道看了老赵之后的两篇吐槽Go语言的文章, 发现完全不是那回事情, 吐槽内容偏差太远. 本来没想掺和进来, 但是看到QQ群里和图灵社区有很多人甚至把老赵的文章当作真理一样. 实在忍不住, 昨天注册了帐号, 进来也说下我的观点.

这是老赵的几篇文章:

- Go是一门有亮点的语言，老许是牛人，但这本书着实一般
- 为什么我认为goroutine和channel是把别的平台上类库的功能内置在语言里
- 为什么我不喜欢Go语言式的接口（即Structural Typing）

补充说明:

因为当前这篇文章主要是针对老赵的不喜欢Go语言式的接口做 评论. 因为标题的原因, 也造成了很大的争议性(因为很多人说我理解的很多观点和老赵的原文不相符).

后面我会对Go语言的一些特性一些简单的介绍, 但是不会是现在这种方式.

> 所谓Go语言式的接口，就是不用显示声明类型T实现了接口I，只要类型T的公开方法完全满足接口I的要求，就可以把类型T的对象用在需要接口I的地方。这种做法的学名叫做Structural Typing，有人也把它看作是一种静态的Duck Typing。除了Go的接口以外，类似的东西也有比如Scala里的Traits等等。有人觉得这个特性很好，但我个人并不喜欢这种做法，所以在这里谈谈它的缺点。当然这跟动态语言静态语言的讨论类似，不能简单粗暴的下一个“好”或“不好”的结论。

原文观点:

- Go的隐式接口其实就是静态的Duck Typing. 很多语言(主要是动态语言)早就有.
- 静态类型和动态类型没有绝对的好和不好.

我的观点:

- Go的隐式接口Duck Typing确实不是新技术, 但是在主流静态编程语言中支持Duck Typing应该是很少的(不清楚目前是否只有Go语言支持).
- 静态类型和动态类型虽然没有绝对的好和不好, 但是每个都是有自己的优势的, 没有哪一个可以包办一切. 而Go是试图结合静态类型和动态类型(`interface`)各自的优势.

> 那么就从头谈起：什么是接口。其实通俗的讲，接口就是一个协议，规定了一组成员，例如.NET里的`ICollection`接口：

```
public interface ICollection {
     int Count { get; }
     object SyncRoot { get; }
     bool IsSynchronized { get; }
     void CopyTo(Array array, int index);
}
```

> 这就是一个协议的全部了吗？事实并非如此，其实接口还规定了每个行为的“特征”。打个比方，这个接口的`Count`除了需要返回集合内元素的数目以外，还隐含了它需要在O(1)时间内返回这个要求。这样一个使用了`ICollection`接口的方法才能放心地使用`Count`属性来获取集合大小，才能在知道这些特征的情况下选用正确的算法来编写程序，而不用担心带来性能问题，这才能实现所谓的“面向接口编程”。当然这种“特征”并不但指“性能”上的，例如`Count`还包含了例如“不修改集合内容”这种看似十分自然的隐藏要求，这都是`ICollection`协议的一部分。

原文观点:

- 接口就是一个协议, 规定了一组成员.
- 接口还规定了每个行为对应时间复杂度的"特征".
- 接口还规定了每个行为还包含是否会修改集合的隐藏要求.

我的观点:

- 第一条: 没什么可解释的, 应该是接口的通俗含义.
- 第二条: 但是接口还包含时间复杂度的"特征"就比较扯了. 请问这个特征是由语言特性来约束(语言如何约束?), 还只是由接口的文档作补充说明(这是语言的特性吗)?
- 第三条: 这个还算是吐槽到了点子上. Go的接口确实不支持C++类似的`const`修饰, 除了接口外的method也不支持(Go的`const`关键字是另一个语义).

但是, C++中有了`const`就真的安全了吗?

```
class Foo {
    private: mutable Mutex mutex_;

    public: void doSomething()const {
        MutexLocker locker(&mutex_);
        // const 已经被绕过了
    }
};
```

C++中方法`const`修饰唯一的用处就是增加各种编译麻烦, 对使用者无法作出任何承诺. 使用者更关心的是`doSomething`的要做什么, 上面的方法其实和`void doSomethingConst()`要表达的是类似的意思.

不管是静态库还是动态库, 哪个能从库一级保证某个函数是不能干什么的? 如果C++的`const`关键字并不能 真正的保证`const`, 而类似的实现细节(也包括前面提到的和时间复杂度相关的性能特征)必须有文档来补充. 那文档应该以什么形式提供(代码注释?Word文档?其他格式文档?)? 这些文档真多能保证每个都会有人看吗? 文档说到底还只是人之间的口头约定, 如果文档真的那么好使(还有实现), 那么汇编语言也可以解决一切问题.

在Go语言是如何解决`const`和性能问题的?

首先, 对于C语言的函数参数传值的语义, `const`是必然的结果. 但是, 如果参数太大要考虑性能的话, 就会考虑传指针(还是传值的语义), 通过传指针就不能保证`const`的语义了. 如果连使用的库函数都不能相信, 那怎么就能相信它对于的头文件所提供的`const`信息呢?

因为, `const`和性能是相互矛盾的. Go语言中如果想绝对安全, 那就传值. 如果想要性能(或者是返回副作用), 那就传指针:

```
type Foo int

// 要性能
func (self *Foo)Get() int {
    return *self
}
// 要安全
func (self Foo)GetConst() int {
    return self
}
```

Go语言怎么对待性能问题(还有单元测试问题)? 答案是集成`go test`测试工具. 在Go语言中测试代码是pkg(包含`package main`)的一个组成部分. 不仅是普通的pkg可以`go test`, `package main`也可以用`go test`进行测试.

我们给前面的代码加上单元测试和性能测试.

```
// foo_test.go

func TestGet(t *testing.T) {
    var foo Foo = 0
    if v := foo.Get(); v != 0 {
        t.Errorf("Bad Get. Need=%v, Got=%v", 0, v)
    }
}
func TestGetConst(t *testing.T) {
    var foo Foo = 0
    if v := foo.GetConst(); v != 0 {
        t.Errorf("Bad GetConst. Need=%v, Got=%v", 0, v)
    }
}

func BenchmarkGet(b *testing.B) {
```

```
    var foo Foo = 0
    for i := 0; i < b.N; i++ {
        _ = foo.Get()
    }
}
func BenchmarkGetConst(b *testing.B) {
    var foo Foo = 0
    for i := 0; i < b.N; i++ {
        _ = foo.GetConst()
    }
}
```

当然, 最终的测试结果还是给人来看的. 如果实现者/使用者故意搞破坏, 再好的工具也是没办法的.

> 由此我们还可以解释另外一些问题，例如为什么.NET里的List不叫做ArrayList，当然这些都只是我的推测。我的想法是，由于List与IList接口是配套出现的，而像IList的某些方法，例如索引器要求能够快速获取元素，这样使用IList接口的方法才能放心地使用下标进行访问，而满足这种特征的数据结构就基本与数组难以割舍了，于是名字里的Array就显得有些多余。
>
> 假如List改名为ArrayList，那么似乎就暗示着IList可以有其他实现，难道是LinkedList吗？事实上，LinkedList根本与IList没有任何关系，因为它的特征和List相差太多，它有的尽是些AddFirst、InsertBefore方法等等。当然，LinkedList与List都是ICollection，所以我们可以放心地使用其中一小部分成员，它们的行为特征是明确的。

原文观点:

- 推测: 因为为了和 `IList<T>` 接口配套出现的原因, 才没有将 `List<T>` 命名为 `ArrayList<T>`.
- 因为 `IList<T>` (这个应该是笔误, 我觉得作者是说 `List<T>`)索引器要求能够快速获取元素, 这样使用IList接口的方法才能放心地使用下标进行访问(实现的算法复杂度特征向接口方向传递了).
- 不能将 `List<T>` 改为 `ArrayList<T>` 的另一个原因是 `LinkedList<T>` . 因为 `List<T>` 和 `LinkedList<T>` 的时间复杂度不一样, 所以不能是一个接口(大概是一个算法复杂度一个接口的意思?).
- `LinkedList<T>` 与 `List<T>` 都属于 `ICollection<T>` 这个祖宗接口.

我的观点:

- 第一条: 我不知道原作者是怎么推测的. 接口的本意就是要和实现分离. 现在却完全绑定到一起了, 那这样还要接口做什么(一个 `Xxx<T>` 对应一个 `IXxx<T>` 接口)?
- 第二条: 因为运行时向接口传递了某个时间复杂度的实现, 就推导出接口的都符合某种时间复杂度, 逻辑上根本就不通!
- 第三条: 和前两个差不多的意思, 没什么可说的.
- 第四条: 这个应该是Go非入侵接口的优点. C++/Java就是因为接口的入侵性, 才导致了接口和实现无法完全分离. 因为, C++/Java大部分时间都在整理接口间/实现间的祖宗八代之间的关系了(重要的不是如何分类, 而是能做什么). 可以参考许式伟给的Java的例子(了解祖宗八代之间的关系真的很重要吗): http://docs.oracle.com/javase/1.4.2/docs/api/overview-tree.html.

> 这方面的反面案例之一便是Java了。在Java类库中，ArrayList和LinkedList都实现了List接口，它们都有get方法，传入一个下标，返回那个位置的元素，但是这两种实现中前者耗时O(1)后者耗时O(N)，两者大相近庭。那么好，我现在要实现一个方法，它要求从第一个元素开始，返回每隔P个位置的元素，我们还能面向List接口编程么？假如我们依赖下标访问，则外部一不小心传入LinkedList的时候，算法的时间复杂度就从期望的O(N/P)变成了O(N2/P)。假如我们选择遍历整个列表，则即便是ArrayList我们也只能得到O(N)的效率。话说回来，Java类库的List接口就是个笑话，连Stack类都实现了List，真不知道当年的设计者是怎么想的。
>
> 简单地说，假如接口不能保证行为特征，则“面向接口编程”没有意义。

原文观点:

- Java的 `ArrayList` 和 `LinkedList` 都实现了 `List` 接口, 但是 `get` 方法的时间复杂度不同.
- 假如接口不能保证行为特征，则“面向接口编程”没有意义。

我的观点:

- 第一条: 这其实是原作者列的一个前提, 是为了推出第二条的结论. 但是, 我觉得这里的逻辑同样是有问题的. 有这个例子只能说明接口有它的不足, 但是怎么就证明了 **则“面向接口编程”没有意义**?
- 第二条: 我要反问一句, 为什么非要在这里使用接口(难道是被C++/Java的面向对象洗脑了)? 接口有它合适的地方(面向逻辑层面), 也有它不合适的地方(面向底层算法层面). 在这里为什么不直接使用 `ArrayList` 或 `LinkedList` ?

> 而Go语言式的接口也有类似的问题，因为Structural Typing都只是从表面（成员名，参数数量和类型等等）去理解一个接口，并不关注接口的规则和含义，也没法检查。忘了是Coursera里哪个课程中提到这么一个例子：
>
> ```
> nterface IPainter {
>     void Draw();
> }
>
> nterface ICowBoy {
>      void Draw();
> }
> ```

> 在英语中Draw同时具有“画画”和“拔枪”的含义，因此对于画家（Painter）和牛仔（Cow Boy）都可以有Draw这个行为，但是两者的含义截然不同。假如我们实现了一个“小明”类型，他明明只是一个画家，但是我们却让他去跟其他牛仔决斗，这样就等于让他去送死嘛。另一方面，“小王”也可以既是一个“画家”也是个“牛仔”，他两种Draw都会，在C#里面我们就可以把他实现为：
>
> ```
> class XiaoWang : IPainter, ICowBoy {
>     void IPainter.Draw() {
>         // 画画
>     }
>
>     void ICowBoy.Draw() {
>         // 掏枪
>     }
> }
> ```
>
> 因此我也一直不理解Java的取舍标准。你说这样一门强调面向对象强调接口强调设计的语言，还要求强制异常，怎么就不支持接口的显示实现呢？

原文观点:

- 不同实现的`Draw`含义不同, 因此接口最好也能支持不同的实现.
- Java/Go之类的接口都没有C#的接口强大.

我的观点:

- 第一条: 不要因为自己有个锤子, 就把什么东西都当作钉子! 你这个是C#的例子(我不懂C#), 但是请不要往Go语言上套! 之前是C++搞出了个函数重载(语义还是相似的, 但是签名不同), 没想到C#还搞了个支持同一个单词不同含义的特性.
- 第二条: 只能说原作者真的不懂Go语言.

Go语言为什么不支持这些花哨的特性? 因为, 它们太复杂且没多大用处, 写出的代码不好理解(如果原作者不提示, 谁能发现`Darw`的不同含义这个坑?). Go语言的哲学是: "Less is more!".

看看Go语言该怎么做:

```
type Painter interface {
    Draw()
}
type CowBoyer interface {
    DrawTheGun()
}

type XiaoWang struct {
    // ...
}

func (self *XiaoWang)Draw() {
    // ...
}
func (self *XiaoWang)DrawTheGun() {
    // ...
}
```

`XiaoWang`需要关心的只是自己有哪些功能(`method`), 至于祖宗关系开始根本不用关心. 等到`XiaoWang`各种特性逐渐成熟稳定之后, 发现新来的`XiaoMing`也有类似的功能特征, 这个时候才会考虑如何用接口来描述`XiaoWang`和`XiaoMing`共同特征.

> 这就是我更倾向于Java和C#中显式标注异常的原因。因为程序是人写的，完全不会因为一个类只是因为存在某些成员，就会被当做某些接口去使用，一切都是经过“设计”而不是自然发生的。就好像我们在泰国不会因为一个人看上去是美女就把它当做女人，这年头的化妆和PS技术太可怕了。

原文观点:

- 接口是经过“设计”而不是自然发生的.
- 接口有不足, 因为在泰国不能根据`美女`这个接口来推断这个人是`女人`这个类型.

我的观点:

- Go的哲学是先构造具体对象, 然后再根据共性慢慢归纳出接口, 一开始不用关心祖宗八代的关系.
- 请问`女人`是怎么定义的, 难道这不是一个接口?

> 我这里再小人之心一把：我估计有人看到这里会说我只是酸葡萄心理，因为C#中没有这特性所以说它不好。还真不是这样，早在当年我还没听说Structural Typing这学名的时候就考虑过这个问题。我写了一个辅助方法，它可以将任意类型转化为某种接口，例如：
>
> ```
> XiaoMing xm = new XiaoMing();
> ICowBoy cb = StructuralTyping.From(xm).To<ICowBoy>();
> ```
>
> 于是，我们就很快乐地将只懂画画的小明送去决斗了。其内部实现原理很简单，只是使用Emit在运行时动态生成一个封装类而已。此外，我还在编译后使用`Mono.Cecil`分析程序集，检查`From`与`To`的泛型参数是否匹配，这样也等于提供了编译期的静态检查。此外，我还支持了协变逆变，还可以让不需要返

> 回值的接口方法兼容存在返回值的方法，这可比简单通过名称和参数类型判断要强大多了。

原文观点:

- C#接口的这个特性很NB...

我的观点:

我们看看Go是该怎么写(基于前面的Go代码, 没有`Draw`重载):

```
var xm interface{} = new(XiaoWang)
cb := xm.(Painter).(CowBoyer)
```

但是, 我觉得这样写真的很变态. Go语言是为了解决实际的工程问题的, 不是要像C++那样成为各种NB技术的大杂烩.

我始终认同一个观点: 任何语言都可以写出垃圾代码, 但是不能以这些垃圾代码来证明原语言也垃圾.

> 有了多种选择，我才放心地说我喜欢哪个。JavaScript中只能用回调编写代码，于是很多人说它是JavaScript的优点，说回调多么多么美妙我会深不以为然——只是没法反抗开始享受罢了嘛……
>
> 这篇文章好像吐槽有点多？不过这小文章还挺爽的。

这段不是接口相关, 懒得整理/吐槽了.

最后我只想说一个例子, 从C语言时代就很流行的`printf`函数. 我们看看Go语言中是什么样子(`fmt.Fprintf`):

```
func Fprintf(w io.Writer, format string, a ...interface{}) (n int, err error)
```

在Go语言中, `fmt.Fprintf`只关心怎么识别各种`a ...interface{}`, 怎么format这些参数, 至于怎么写, 写到哪里去那完全是`w io.Writer`的事情.

这里第一个参数的`w io.Writer`就是一个接口, 它不仅可以写到`File`, 也可以写到`net.Conn`, 准确的说是可以写到任何实现了`io.Writer`接口的对象中.

因为, Go语言接口的非入侵性, 我们可以独立实现自己的对象, 只要符合`io.Writer`接口就行, 然后就可以和`fmt.Fprintf`配合工作.

后面的可变参数`interface{}`同样是一个接口, 它代替了C语言的`void*`, 用于格式化输出各种类型的值. (更准确的讲, 除了基础类型, 参数`a`必须是一个实现了`Stringer`接口的扩展类型).

接口是一个完全正交的特性, 可以将`Fprintf`从各种`a ...interface{}`, 以及各种`w io.Writer`完全剥离出来. Go语言也是这样, `struct`等基础类型的内存布局还是和C语言中一样, 只是加了个`method`(在Go1.1中, `method value`就是一个普通闭包函数), 接口以及`goroutine`都是在没有破坏原有的类型语义基础上正交扩展(而不是像C++那样搞个构造函数, 以后又是析构函数的).

我到很想知道, 在C++/C#/Java之类的语言中, 是如何实现`fmt.Fprintf`的.