**go interface**

Go语言里面设计最精妙的是interface，它让面向对象，内容组织的实现非常方便。如果说goroutine和channel是支撑Go语言的并发模型的基石，那么接口是Go语言整个类型系统的基石，让Go语言在基础编程哲学的探索上达到前所未有的高度。Go语言的接口不单单只是借口，下面开始记录下我所学到的Go的interface的特性吧！

1. **Java语言的interface**

在Go语言出现之前，接口主要作为不同组件之间的契约存在。对契约的实现是强制的，你必须声明你确实实现了该接口。为了实现一个接口，你需要从该接口继承，如Java程序：

// 声明一个接口

public interface myInterface {

int myCompareTo(Object other);

}

// 实现该接口

public class Client implements myInterface{

int i = 0;

public int myCompareTo(Object other) {

int j = (int)other;

return i >= j ? 1 : -1;

}

public static void main(String[] args) {

Client client = new Client();

System.out.println(client.myCompareTo(3));

}

}

这类接口我们称之为侵入式接口。“侵入式”的主要表现在于实现类需要明确声明自己实现了该接口。这种强制性的接口继承是面向对象编程思想发展过程中的一个遭受相当多质疑的特征。由于这种侵入性接口存在的争议性，所以Go语言设计出一套不一样的interface。

1. **什么是Go的interface**

简单的说，interface是一组method的组合，我们通过interface来定义对象的一组行为。如我们定义一组“能吃能唱”的行为：

// 定义一个接口interface

type Species interface {

Eat()

Sing()

}

这显然“人”、“鸟”都能做到的，首先定义人和鸟的struct：

// 定义一个struct表示“人”

type Men struct {

name string

age int

}

// 定义一个struct表示“鸟”

type Birds struct {

name string

color string

}

然后再让“人”和“鸟”对象实现该Men接口，首先是“人”：

func (m \*Men) Eat() {

fmt.Printf("您好，我们人类吃饭")

}

func (m \*Men) Sing() {

fmt.Printf("\t 我们唱的歌我们人类都听得懂")

}

然后是“鸟”：

func (b \*Birds) Eat() {

fmt.Printf("我们鸟吃虫")

}

func (b \*Birds) Sing() {

fmt.Printf("\t 我们唱的歌只有鸟类懂")

}

最后，让我们来看看实现的结果：

func main() {

m := Men{"fanly", 26}

b := Birds{"乌鸦", "黑色"}

fmt.Printf("人类的行为：")

m.Eat()

m.Sing()

fmt.Printf("\n 鸟的行为：")

b.Eat()

b.Sing()

}

运行结果：

[ `run main` | done: 376.0215ms ]

人类的行为：您好，我们人类吃饭 我们唱的歌我们人类都听得懂

鸟的行为：我们鸟吃虫 我们唱的歌只有鸟类懂

整个程序代码如下：

package main

import (

"fmt"

)

// 定义一个接口interface

type Species interface {

Eat()

Sing()

}

// 定义一个struct表示“人”

type Men struct {

name string

age int

}

// 定义一个struct表示“鸟”

type Birds struct {

name string

color string

}

// 实现接口中Eat方法

func (m Men) Eat() {

fmt.Printf("您好，我们人类吃饭")

}

// 实现接口中Sing方法

func (m Men) Sing() {

fmt.Printf("\t 我们唱的歌我们人类都听得懂")

}

func (b Birds) Eat() {

fmt.Printf("我们鸟吃虫")

}

func (b Birds) Sing() {

fmt.Printf("\t 我们唱的歌只有鸟类懂")

}

func main() {

// 创建Men和Birds对象

m := Men{"fanly", 26}

b := Birds{"乌鸦", "黑色"}

fmt.Printf("人类的行为：")

m.Eat()

m.Sing()

fmt.Printf("\n 鸟的行为：")

b.Eat()

b.Sing()

}

通过上面代码可以看出，interface可以被任意的对象实现。我们看到Species接口被Men和Birds实现。同理，一个对象可以实现任意多个interface。最后需要指出：任意的类型都实现了空interface（即，interface{}），也就是说包含0个method的interface。

Go语言的非侵入式接口存在着以下优势：

一. Go语言的标准库，再也不需要绘制类库的继承树图，在Go中，类的继承树并无意义，你只需要知道这个类实现了哪些方法，每个方法是什么作用就行了。

二. 实现类的时候，只需要关心自己应该提供哪些方法，不用再纠结接口需要拆的多细才合理。接口由使用方按需定义，而不用事前规划。

三. 不用为了实现一个接口而导入一个包，因为多引用一个外部的包，就意味着更多的耦合。接口由使用方按自身需要来定义，使用方无需关心是否有其他模块定义过类似的接口。

1. **Go中interface赋值**

interface在Go语言中分成两种情况：

1. 将对象实例赋值给接口； 2. 将一个interface赋值给另一个interface。

我们来看第一种情况。 如果我们定义了一个interface的变量，那么这个变量里面可以存储实现这个interface的任意类型的对象。例如上面例子中，我们定义了一个Species接口，那么这个接口变量可以存储Men或者Birds的值，这点和Java中的接口类似，现在先来看看Java中是如何实现的：

public class Hello {

public static void main(String[] args) {

// 定义接口species变量s

species s;

// 将men类的变量赋值给接口变量s

s = new men();

System.out.println(s.eat() + "\t" + s.sing());

// 重新将birds类的变量赋值给s，得到的结果就不一样

s = new birds();

System.out.println(s.eat() + "\t" + s.sing());

}

}

interface species {

String eat();

String sing();

}

class men implements species{

public String eat() {

return "Men Eat";

}

public String sing() {

return "Men sing";

}

}

class birds implements species{

public String eat() {

return "Birds eat";

}

public String sing() {

return "Birds sing";

}

}

运行结果如下：

Executing: javacexec.bat D:\tools\javas\projects\Hello.java

Men Eat Men sing

Birds eat Birds sing

[Finished]

同样的，在Go语言中，我们同样可以将不同的实现变量赋值给接口，在上文Go代码中的main函数做个修改，代码如下所示：

func main() {

// 创建Men和Birds对象

m := Men{"fanly", 26}

b := Birds{"乌鸦", "黑色"}

// 定义Species接口类型的变量s

var s Species

// 将Men的变量m赋值给接口变量s

s = m

fmt.Printf("人类的行为：")

s.Eat()

s.Sing()

// 同样的，将Birds的变量b赋值给接口变量s

s = b

fmt.Printf("\n 鸟的行为：")

s.Eat()

s.Sing()

}

代码执行结果如下：

[ `run main` | done: 495.0283ms ]

人类的行为：您好，我们人类吃饭 我们唱的歌我们人类都听得懂

鸟的行为：我们鸟吃虫 我们唱的歌只有鸟类懂

现在来看第二种情况，也就是所谓的“嵌入interface”。Go语言里面真正吸引人的是其内置的逻辑语法，如果一个one接口作为two接口的一个潜入字段，那么two接口包含了one接口里面的所有method。 如在Go源码中的io包下main的io.ReadWriter，它包含了io包下面的Reader和Writer两个interface。

// ReadWriter is the interface that groups the basic Read and Write methods.

type ReadWriter interface {

Reader

Writer

}

这个接口组合了Reader和Writer两个interface，这两个接口如下所示：

type Reader interface {

Read(p []byte) (n int, err error)

}

// Writer is the interface that wraps the basic Write method.

//

// Write writes len(p) bytes from p to the underlying data stream.

// It returns the number of bytes written from p (0 <= n <= len(p))

// and any error encountered that caused the write to stop early.

// Write must return a non-nil error if it returns n < len(p).

type Writer interface {

Write(p []byte) (n int, err error)

}

io.ReadWriter就等同于:

type ReadWriter interface {

Read(p []byte) (n int, err error)

Write(p []byte) (n int, err error)

}

因为这两种写法的表意完全相同：ReadWriter接口既能做Reader接口的所有事情，又能做Writer接口的所有事情。在Go语言包中，还有众多类似的组合接口，比如ReadWriteCloser、ReadWriteSeeker、ReadSeeker和WriteCloser等。