Go 抽象特性

作者：feiwo
原文链接：https://ld246.com/article/1610117570725
来源网站：链滴

一个有用的程序，往往是对真实世界抽象的描述，描述一个事物的属性，描述一个行为的功能流程。

结构体

Go语言并不是传统意义上的面向对象语言，它没有像Java中的class，而是使用struct来对真实世界事物进行抽象。struct之间没有继承，而是使用组合来达到类型Java继承的效果。

Java

```
// 在Java中使用class类进行一类事物的抽象
public class Person {
  String name;
  Integer age;

  public Person(String name, Integer age) {
    this.name = name;
    this.age = age;
  }
}

// 使用exetds关键字来表示类与类之间的继承关系 --- 即父子关系
public class Student extends Person {
  Float grade;

  public Student(String name, Integer age, Float grade) {
    this.grade = grade;
    super(name, age);
  }
}

public class Main {
```

```
  public static void main(String[] agrs) {
    Student stu = new Student("feiwo", 18, 90.8);
    System.out.println(stu.name) // out: feiwo
  }
}
```

Go

```
// 使用struct来对一类事物抽象的描述
type Person struct {
  Name string
  Age int
}

type Student struct {
  Person // 使用组合的方式，复用Person struct来达到继承类似的效果
  Grade float32
}

func main() {
  // Go和Java不同，没有构造函数，一般会采用字面量的形式初始化struct或自己创建一个函数用来
始化一个struct
  stu := Student{
    Person: {
      Name: "feiwo",
      Age: 18,
    },
    Grade: 90.8,
  }

  fmt.Println(stu.Name) // out: feiwo
}
```

方法定义

Go的方法定义，与其他面向对象语言不同，一般的，面向对象语言的方法都是定义在类中的，而**Go**用的是struct，且只能在struct中定义字段，而struct的方法则是采用函数挂载的形式进行结构体与方的绑定，匿名组合的struct方法是会传递类被嵌入的struct，如果有同名的方法，则会调用被嵌入的struct所绑定的方法。

Java

```
public class Person {
  String name;

  public Person(String name) {
    this.name = name;
  }
  // Java方法定义在类代码块中
  public void say() {
    System.out.println("hello "+ this.name);
  }
}
```

```
public class Main {

  public static void main(String[] args) {
    Person p = new Person("feiwo");
    p.say(); // out: hello feiwo
  }

}
```

Go

```
type Person struct {
  Name string
}

// go使用struct挂载函数的方式，将函数绑定到struct上
// 挂载在struct指针类型上
// 这里的p是Person结构体的指针，是Person实例中内存的地址
// 在这个方法里面修改Person属性的值，是会影响Person实例的属性值
func (p *Person) Say() {
  fmt.Println("hello "+ p.Name)
}

// 也可以挂载在struct类型
// 因为go里面所有参数都是只拷贝，这里的p变量，是Person结构体的拷贝
// 在这个方法里面修改Person结构体里的属性，并不会影响实例属性的值
func (p Person) Sleep() {

}

func main() {
  p := Person{
    Name: "feiwo",
  }

  p.Say() // out: hello feiwo
}
```

方法变量

Go的struct方法是可以赋值给一个变量的，这个变量就是方法值(Method Value)，方法值其实是一带有闭包的函数变量，接收值被隐式地绑定到方法值的闭包环境中，后续调用不需要再显式地传递调者。

```
type Person struct {
  Name string
}

func (p *Person) Say() {
  fmt.Println("hello " + p.Name)
}

func main() {
  p := &Person{Name: "feiwo"}
```

```
  f := p.Say

  f() // out: hello feiwo
}
```

接口

接口是一种编程规约，也是一组方法签名的集合。在Java中，我们通常需要面向接口编程，这样更加象，而不至于依赖某个具体的实现类，减少类与类之间的耦合，采用接口解耦，只要一个类实现了这接口，就可以将原先的实现类替换掉。

Java

```
public interface Reader {
  int Read(byte[] content)
}

public class FileRead implements Reader {

  @Overrid
  public int Read(byte[] content) {
    // TODO
  }
}

public class NetRead implements Reader {

  @Overrid
  public int Read(byte[] content) {
    // TODO
  }
}

public class Main {
  public static void main(String[] args) {

    FileRead f = new FileRead();
    R(f);

    NetRead n = new NetRead();
    R(n)

  }

  // 将接口类型作为参数，达到解耦具体的类型实例
  // 只要类实现了Reader的接口，都可作为参数传入
  public static void R(Reader reader) {
    // TODO
  }
}
```

Go

在**Go**中，接口变量没有初始化时，默认值为nil，nil这个关键字类似于Java语言中的null，接口变量

只有在初始化之后采用使用的意义。

```
// 使用interface关键字来定义接口
type Reader interface {
  Read(p byte[]) (int, error)
}

type Writer interface {
  Write()
}

// 和struct一样，接口也支持组合，将多个接口组合成一个接口
// Go官方鼓励使用小接口定义，然后组合成大接口，而不是定一个大而全的接口
type ReadWriter interface {
  Reader
  Writer
}

type FileRead struct {

}

// 只要方法名签名和接口中的方法签名一样
// 并且接口实现了接口中全部的方法，则表示这个结构体实现了这个接口
func (f *FileRead) Read(p []byte) (int, error) {
  // TODO
}

type NetRead struct {

}

func (n *NetRead) Read(p []byte) (int, error) {
  // TODO
}

func main() {
  fr := new(FileRead)
  nr := new(NetRead)

  R(fr)
  R(nr)
}

// 只要实现了Reader接口的结构体指针都可以作为参数传入
func R(r Reader) {
  // TODO
}
```

空接口

因为**Go**不是一个纯的面向对象语言，所以没有最终的父类，也就是Object，也没有泛型，在**Go**中则用空接口(interface{})来表示一个任意类型，由于空接口的方法集为空，所以任意类型都可以认为实了这个空接口，任意类型的实例都可赋值或传递给空接口。通常空接口，搭配类型断言来使用。

```
func Response(data interface{}) interface{} {
  switch data.(type) { // 类型断言
    case NoErrorData:
       return buildData(data)
     case PageData:
       return buildPageData(data)
    case ErrorData:
       return buildErrorData(data)
  }
}
```

接口的优点

1. 解耦： 复杂系统进行垂直和水平的分割是常用的设计手段，在层与层之间使用接口进行抽象和解是一种很好的编程策略

2. 实现泛型：使用空接口可以一定层度上解决 **Go**没有泛型的窘境

接口内部的实现

非空接口的数据结构

```
type iface struct {
  tab *itab // itab 是存放类型及方法指针信息，主要是自身类型和绑定的实例类型以及实例相关的函指针
  data unsafe.Pointer // 数据信息， 指向的是接口绑定的实例的副本，接口的初始化也是一种值拷贝
}

// itab存放.rodata只读存储段中，存放在静态分布的存储空间中，不受GC的限制，其内存不会被回收
type itab struct {
  inter *interfacetype // 接口自身的静态类型
  _type *_type // _type就是接口存放的具体实例的类型（动态类型）,iface里的data就是指向该类型值，一个是类型信息，一个是类型的值
  hash uint32 // 存放具体类型的Hash值
  _ [4]byte // 内存对齐
  fun [1]uintptr // 函数指针，可以理解为C++中的虚拟函数指针，提供一种动态绑定的机制，这里数组大小不是固定的，编译器负责填充
}

type _type struct {
    size       uintptr // 大小
    ptrdata    uintptr // size of memory prefix holding all pointers
    hash       uint32 // 类型的Hash
    tflag      tflag // 类型的特征标记
    align      uint8 // _type作为整体变量存放时的对齐字节数
    fieldalign uint8 // 当前结构字段的对齐字节数
    kind       uint8 // 基础类型枚举值和反射中的Kind一直，Kind决定了如何解析该类型
    alg        *typeAlg // 指向一个函数指针表，该表有两个函数，一个是计算类型Hash函数，一个是较两个类型是否相同的equle函数
    // gcdata stores the GC type data for the garbage collector.
    // If the KindGCProg bit is set in kind, gcdata is a GC program.
    // Otherwise it is a ptrmask bitmap. See mbitmap.go for details.
    gcdata    *byte // GC相关信息
    str       nameOff // 用来表示类型名称字符串在编译后二进制文件中的某个section的偏移量，由
```

```
接器负责填充
    ptrToThis typeOff // 表示类型元信息的指针，在编译后二进制文件中某个scetion的偏移量，由
接器负责填充
}

// 描述接口的类型
type interfacetype struct {
    typ     _type // 类型信息
    pkgpath name // 接口所属包的名字信息，name内存存放的不仅有名称还有描述信息
    mhdr    []imethod // 接口的方法元信息集合
}

// 接口方法元信息
type imethod struct {
    name nameOff // 方法名在编译后的secetion里面的偏移量
    ityp typeOff // 方法类型在编译后在secetion里面的偏移量
}
```

接口的动态调用

接口调用分为两个阶段：

1. 第一阶段是构建 iface动态数据结构，这个阶段是在接口实例化的时候完成。fr := new(FileRead)
2. 第二阶段是通过函数指针间接调用接口绑定的实例方法的过程。 fr.Read()

原文链接：Go 抽象特性