方法method

- Go 中虽没有class，但依旧有method
- 通过显示说明receiver来实现与某个类型的组合
- 只能为同一个包中的类型定义方法
- Receiver 可以是类型的值或者指针
- 不存在方法重载
- 可以使用值或指针来调用方法，编译器会自动完成转换
- 从某种意义上来说，方法是函数的语法糖，因为receiver其实就是方法所接收的第1个参数（Method Value vs. Method Expression）
- 如果外部结构和嵌入结构存在同名方法，则优先调用外部结构的方法
- 类型别名不会拥有底层类型所附带的方法
- 方法可以调用结构中的非公开字段

怎么为函数些方法呢？其实和写函数非常类似。最不过多了一个**概念叫做receiver**。就是一个接收者，那么编译器就根据接受者的类型来判断它是那一个结构的方法。

还是使用 **func** **关键字**，然后直接是一**括号()** ，先不写名称，先写括号()，括号()当中就是一个**接收者**。比如在括号里面
写一个a，就是取一个局部变量名称a，它的接收者的类型是什么呢？**类型就是结构A**。然后要**取一个方法的名称()**。

定义method语法是：

```
1.  func (a A) Print() {
2.      fmt.Println("A")
3.  }
```

在main函数中声明一个结构A，之后调用这个Print()的方法。这样就完成了一个

```
1.  func (a A) Print() {
2.      fmt.Println("A")
3.  }
```

方法的书写。如果还有一些复杂的东西，可以在`func (a A) Print()` 中的 `Print()` 写上一些`Print(d int, a strint)`参数 ，这个就是参数列表，实际上是完全一样的。其实这个**receiver** 就是这个函数的第一个接收者，而且是强制性规定的。这个时候它就变成了一个方法。这个方法

```
1.  func (a A) Print() {
2.      fmt.Println("A")
3.  }
```

和这个结构

```
1.  type A struct {
2.          Name string
3.  }
```

就连接到了一起。

详细代码：

```go
package main

import (
    "fmt"
)

type A struct {
    Name string
}

type B struct {
    Name string
}

func main() {
    a := A{}  //声明一个结构A
    a.Print() //调用名为Print()方法
}

func (a A) Print() { //定义接收者为结构A名为Print的method
    fmt.Println("A") //打印A
}
```

```
/home/jiemin/code/GOlang/go/src/method/method  [/home/jiemin/code/GOlang/go/src/method]
A
成功: 进程退出代码 0.
```

那么结构B也需要一个Print()函数。但是Go语言当中又没有方法重载的概念。那么可不可以将

```go
func (a A) Print() { //定义一个名为Print的method
        fmt.Println("A") //打印A
}
```

改成B

```go
func (b B) Print() { //定义一个名为Print的method
        fmt.Println("B") //打印B
}
```

详细代码:

```go
package main

import (
    "fmt"
)

type A struct {
    Name string
}

type B struct {
    Name string
```

```
}

func main() {
    a := A{}  //声明一个结构A
    b := B{}  //声明一个结构B
    a.Print() //调用名为Print()方法
    b.Print() //调用名为Print()方法
}

func (a A) Print() { //定义接受者为结构A的名为Print的method
    fmt.Println("A") //打印A
}

func (b B) Print() { //定义接受者为结构B的名为Print的method
    fmt.Println("B") //打印B
}
```

```
/home/jiemin/code/GOlang/go/src/method/method  [/home/jiemin/code/GOlang/go/src/method]
A
B
成功: 进程退出代码 0.
```

虽然Go语言当中没有方法重载的概念，但是 这个方法 `(a A) Print()` 是和某一个类型

```
type A struct {
        Name string
}
```

互绑定的。所以说如果绑定的对象不同 `func (a A) Print()` 和 `func (a B) Print()` ，这个`Print()`名称实际上不一样的。调用的时候 `a.Print()` 整个整体是不一样的。

如果说调用的都是 `Print()` 显然就好像是用了方法重载，Go语言当中是不允许的。但是我们调用的是是通过 `a.Print()` 和 `b.Print()` 这样是属于俩个不同的东西，它们存在不同的地方，所以说它们是不一样的。但是如果想要这样的话

```
func (a A) Print() { //定义接受者为结构A的名为Print的method
        fmt.Println("A") //打印A
}
```

```
和

func (a A) Print(b int) { //定义接受者为结构A的名为Print的method,同时在Print()中存在int类型的b参数
        fmt.Println("A") //打印A
}
```

这样子就显然不可以了。因为这俩个方法是针对同一个类型的，就没有办法来分辨要调用的`Print()`是哪一个了。

那么既然**receiver** 也是一个参数，只不是强制性的第一个参数。那么既然是参数了就会**涉及到是 指针传递 还是 值传递 ？** 。

```
package main

import (
    "fmt"
)

```

```go
type A struct {
    Name string
}

type B struct {
    Name string
}

func main() {
    a := A{}            //声明一个结构A
    a.Print()           //调用名为Print()方法
    fmt.Println(a.Name) //打印a.Name的值

    b := B{}            //声明一个结构B
    b.Print()           //调用名为Print()方法
    fmt.Println(b.Name) //打印b.Name的值
}

func (a *A) Print() { //定义接受者为结构A的指针传递名为Print的method
    a.Name = "AAA"   //修改a.Name的值
    fmt.Println("A") //打印A
}

func (b B) Print() { //定义接受者为结构B的不是指针传递名为Print的method
    b.Name = "BBB"   //修改b.Name的值
    fmt.Println("B") //打印B
}
```

```
/home/jiemin/code/GOlang/go/src/method/method  [/home/jiemin/code/GOlang/go/src/method]
A
AAA
B

成功: 进程退出代码 0.
```

首先打印`fmt.Println("A")`出来是这个A，打印出来A之后呢，我们在这个接受者为结构A的Print方法修改`a.Name`的值，修改成功，看到打印出来了AAA，接着我们打印`fmt.Println("B")`出来了B，是调用到了`b.Print()` 这个方法，然后在这个方法当中修改了 `b.Name` 的值。但是打印出来`b.Name`的的值是空白。所以说没有修改成功。所以说它这个**receiver**也是遵循正常参数规则，如果是 值传递或者值类型依值传递 它只是得到一个拷贝。如果是依 new()类型 和 指针传递 那么它得到是指针的拷贝。也就是说对于变量的操作，是直接操作原始的对象。而不是用指针，就是说值类型，不使用指针进行传递，它只是得到一个副本，任何修改，结束了这个方法之后都是无效的。

然后我们看到调用Print方法都没有改变，尽管`func (a *A) Print()`这个A变成了指针传递，但是我们调用上面也没有加上`*a.Print()` 星号这样一个调用方法 。为什么呢？这是和这个字段，通过指针来调用字段还是直接通过值来调用这个字段是一样的。Go会自动识别来判断它是通过怎么样的一个转换还是直接通过值来调用这个方法或者这个字段。这样对于操作这个结构是非常方便。

类型别名和方法的组合：

类型别名实际上只是名称改变。实际上底层的类型相关的方法，或者其它东西根本就不会附带过来。底层是int，只不过基本有了int属于，但是实际上方法是没有带过来的。那么它什么方法没有带过来呢？那这里就要涉及到类型都可以为它增加一个方法。因为struct结构是一种类型，可以增加一个方法。其它使用type关键字定义的就没有呢？答案是可以的。

比如代码里面有一个TZ，它的底层类型是int型。

```
package main

import (
    "fmt"
)

type TZ int //TZ底层类型是int型

func main() {
    var a TZ  //
    a.Print() //调用Print()方法

}

func (a *TZ) Print() { //定义接受者为TZ的指针传递名为Print的method
    fmt.Println("TZ") //打印TZ
}
```

```
/home/jiemin/code/GOlang/go/src/method/method [/home/jiemin/code/GOlang/go/src/method]
TZ
成功: 进程退出代码 0.
```

成功输出TZ，也就是说 `type TZ int` 底层是一个int型，都可以为她添加一个方法。这个

```
func (a *TZ) Print() { //定义接受者为TZ的指针传递名为Print的method
        fmt.Println("TZ") //打印TZ
}
```

就是go的灵活之处。它通过这种形式来进行一个非常自由的这种结构的绑定。

那么所说的底层类型为什么在之前讲过类型别名TZ和int类型转换，还是需要一个强制类型的转换。现在知道通过这个方法绑定可以为任何一种 **type关键字** 定义的类型进行一个方法的结合。这个就说明为什么要强制类型转换？因为这个所附带的方法，是不一定的，也就是说虽然使用了int底层类型，但是一些附带的方法，可能就不适合TZ类型。所以就不会将int类型所附带的一些方法，通过类型别名直接附带到TZ当中，这个和嵌入式结构有所不同。

由于这个int类型是一个内置类型。所以没有办法对他进行绑定。为什么呢？因为方法的绑定只能够相同包当中起到一个作用。所以说这个TZ定义到了其它的包当中，不是这个main包当中，那么

```
func (a *TZ) Print() { //定义接受者为TZ的指针传递名为Print的method
        fmt.Println("TZ") //打印TZ
}
```

这个方法就是无效的，因为它不知道去那个包当中寻找这一个类型来和自己进行一个绑定 。如果要对int类型进行一个高级的操作，高级的封装，那就可以把这个int作为一个底层类型，然后在对自定义类型来进行一些方法的绑定或者一些其它的操作。就可以实现一些更加轻松更加高级的操作，而且更加方便。

对比：

什么对比呢？

Method Value 和 Method Expression之间的对比。

这一个 `func (a *TZ) Print() 中的` **receiver** 就是function第一个参数，而且是强制性的第一个参数，所以说它和方法和函数之间实际上有一种互通。 那么这种互通是怎么实现的呢？这里就演示一下。

第一种什么叫Method Value呢？

就是这种形式 `a.Print()` 这种方法中，我们在Print方法

```
func (a *TZ) Print() { //定义接受者为TZ的指针传递名为Print的method
        fmt.Println("TZ") //打印TZ
}
```

中已经声明了 `a *TZ` 这种 receiver。那就通过一种 像类一样调用的方法的形式， `a.Print()` 方法调用 。

什么叫做Method Expression呢?

Method Expression就是我们直接通过类型，而不是说某一种类型变量来进行调用。直接通过一种类型然后将我们的变量作为receiver，作为第一个参数传给那一个方法。举个列子：

比如说接收的是TZ的指针，然后作为一个类型，之后是一个点. 之后我们看到有一个提示叫做Print()的方法。那么这里是没有receiver的，所以说要自己传进一个receiver。那么怎么传呢？由于我们是传的是指针，所以说我们要取地址。

golang代码：

```
package main

import (
    "fmt"
)

type TZ int //TZ底层类型是int型

func main() {
    var a TZ  //
    a.Print() //调用Print()方法
    /*比如说接收的是TZ的指针，然后作为一个类型，之后是一个点.
    之后我们看到有一个提示叫做Print()的方法。
    那么这里是没有receiver的，所以说要自己传进一个receiver。
    那么怎么传呢？由于我们是传的是指针，所以说我们要取地址。*/
    (*TZ).Print(&a)
}

func (a *TZ) Print() { //定义接受者为TZ的指针传递名为Print的method
    fmt.Println("TZ") //打印TZ
}
```

```
/home/jiemin/code/GOlang/go/src/method/method [/home/jiemin/code/GOlang/go/src/method]
TZ
TZ
成功: 进程退出代码 0.
```

成功打印俩个TZ，也就说这`a.Print()`和`(*TZ).Print(&a)`俩种方法都成功调用了 `Print()` 方法。所以说这个只是官方一个区别，官方为了区别就给他们取名

`a.Print()` 这种方法叫做**Method Value**

`(*TZ).Print(&a)` 这种方法叫做**Method Expression**

那么既然我们有嵌入结构，也就是我们刚才讲的struct，想刚才有那种自定义的A和B结构，既然有字段名称冲突，当然也有方法名称冲突，这个解决规则是和字段一样的。根据优先级，从最高级开始找，找到最低级。找不到呢，就会编译错误。找到了呢，就是调用它。调用最先找到的那个方法，如果说同级当中有同名方法，这个显然就是不能通过编译的。

方法的访问权限:

在go语言当中，首字母大写是导出字段，是能够被外包引用的。首字母小写是私有字段，不能被外包引用。那么方法可不可访问私有字段呢？来试一下，

```
1.  package main
2.
3.  import (
4.      "fmt"
5.  )
6.
7.  type A struct {
8.      name string
9.  }
10.
11. func main() {
12.     a := A{}  // 声明一个结构A
13.     a.Print() // 调用Print()方法
14.
15. }
16.
17. func (a *A) Print() { //定义接受者为A的指针传递名为Print的method
18.     a.name = "123"      // 修改a.name的值
19.     fmt.Println(a.name) //打印a.name
20. }
```

```
/home/jiemin/code/GOlang/go/src/method/method  [/home/jiemin/code/GOlang/go/src/method]
123
成功: 进程退出代码 0.
```

成功修改了a.name的值。这个就和其它面对对象编程语言一样的道理，这个class当中和Print()方法是可以访问结构中的私有字段或者公有字段。所以说方法的访问权限是很高的，然后我们看到在这个main函数当中，可不可以访问呢？可以看一下

```
1.  package main
2.
3.  import (
4.      "fmt"
5.  )
6.
7.  type A struct {
8.      name string
9.  }
10.
11. func main() {
12.     a := A{}  // 声明一个结构A
13.     a.Print() // 调用Print()方法
14.     //a.name = "456"      // 修改a.name的值
```

```
    fmt.Println(a.name) //打印a.name

}

func (a *A) Print() { //定义接受者为A的指针传递名为Print的method
    a.name = "123"      // 修改a.name的值
    fmt.Println(a.name) //打印a.name
}
```

```
/home/jiemin/code/GOlang/go/src/method/method [/home/jiemin/code/GOlang/go/src/method]
123
123
成功: 进程退出代码 0.
```

在main函数中也是成功打印出来123，也就是在main函数当中也可以操作这个结构

```
type A struct {
        name string
}
```

中的`name string` 私有字段。

那么小写和大写到底区别在哪里呢？
它的区别是以包为级别，以package为级别的。也就说私有字段，只是在整个package当中实际上还是相当于公有的，以对整个package级别是公有的，如果超出了不是在main包当中或者在其它包当中，那么这个私有字段是访问不到的。就必须使用首字母大写的导出字段才可以被外部的包访问到。所以说对同一个包来讲，它其实都是可见的内容。

课堂作业

- 根据为结构增加方法的知识，尝试声明一个底层类型为int的类型，并实现调用某个方法就递增100。

如：a:=0，调用a.Increase()之后，a从0变成100。

作业：
代码如下：

```
package main

import (
    "fmt"
)

type method_work int // 声明一个类型，它的底层类型是int

func (mw *method_work) Increase(num int) { /* 创建一个方法接收值是method_work，要对它的值进行一名为Increase()的方法里面传入一个int类型的参数num */
    *mw += method_work(num) // 要对num类型进行强制性转换，使用method_work()转换
}

```

```
14. func main() {
15.     var me_wk method_work // 声明一个类是是method_work
16.     me_wk = 0             // 给me_wk赋值一个值，或者不用给，因为它身是int类型，int类型初始化就是零
17.     me_wk.Increase(100)   // 调用Increase()的方法，在里面传入100
18.     fmt.Println(me_wk)    // 打印me_wk的值
19. }
```

```
/home/jiemin/code/GOlang/go/src/method/method_work/method_work [/home/jiemin/code/GOlang/go/src/method/method_work]
100
成功: 进程退出代码 0.
```

意思：

首先声明一个类型 `type method_work int` 它底层类型就是int类型。然后它有一个方法

```
1. func (mw *method_work) Increase(num int) { //
2.         *mw += method_work(num) //
3. }
```

因为要对它的值进行一个修改，所以它 `*method_work` 是一个指针。它的方法名字叫做 `Increase(num int)` 它接收一个参数是int类型的num。在里面就要进行一个 mw 加等于 num这个参数。

接下来在main函数声明一个类型。调用它的 `me_wk.Increase(100)` 方法，方法里面传入100数值。然后打印一下它`fmt.Println(me_wk)`的值。因为用了一个类型别名的形式，它的底层类型为int类型。所以说要显示声明`method_work`类型 `var me_wk method_work` 然后给`me_wk`一个数值`me_wk = 0`，或者说不用给，因为它本身是int类型，int类型本身在系统初始化时候会有一个零值。

在

```
1. func (mw *method_work) Increase(num int) {
2.         *mw += num
3. }
```

方法中num是一个int类型，因为我们传递进来是一个100，它是intl类型，尽管这个 `method_work` 底层类型是int类型，但实际上它和int类型还有有所区别的。所以说我们在使用num的时候，我们要先将它进行强制类型的转换，转换到我们这个 `method_work` 类型

```
1. func (mw *method_work) Increase(num int) {
2.         *mw += method_work(num)
3. }
```

这个时候就看到`me_wk`成功输出了100。