25｜方法：方法集合与如何选择receiver类型？

2021-12-10 Tony Bai

《Tony Bai · Go语言第一课》 课程介绍 >

讲述：Tony Bai
时长 15:57 大小 14.62M

你好，我是 Tony Bai。

在上一节中我们开启了 Go 方法的学习，了解了 Go 语言中方法的组成、声明和实质。可以说，我们已经正式入门 Go 方法了。

入门 Go 方法后，和函数一样，我们要考虑如何进行方法设计的问题。由于在 Go 语言中，**方法本质上就是函数**，所以我们之前讲解的、关于函数设计的内容对方法也同样适用，比如错误处理设计、针对异常的处理策略、使用 defer 提升简洁性，等等。

但关于 Go 方法中独有的 receiver 组成部分，却没有现成的、可供我们参考的内容。而据我了解，初学者在学习 Go 方法时，最头疼的一个问题恰恰就是**如何选择 receiver 参数的类型**。

那么，在这一讲中，我们就来学习一下不同 receiver 参数类型对 Go 方法的影响，以及我们选择 receiver 参数类型时的一些经验原则。

receiver 参数类型对 Go 方法的影响

要想为 receiver 参数选出合理的类型，我们先要了解不同的 receiver 参数类型会对 Go 方法产生怎样的影响。在上一节课中，我们分析了 Go 方法的本质，得出了“Go 方法实质上是**以方法的 receiver 参数作为第一个参数的普通函数**”的结论。

对于函数参数类型对函数的影响，我们是很熟悉的。那么我们能不能将方法等价转换为对应的函数，再通过分析 receiver 参数类型对函数的影响，从而间接得出它对 Go 方法的影响呢？

我们可以基于这个思路试试看。我们直接来看下面例子中的两个 Go 方法，以及它们等价转换后的函数：

复制代码

```
1 func (t T) M1() <=> F1(t T)
2 func (t *T) M2() <=> F2(t *T)
```

这个例子中有方法 M1 和 M2。M1 方法是 receiver 参数类型为 T 的一类方法的代表，而 M2 方法则代表了 receiver 参数类型为 *T 的另一类。下面我们分别来看看不同的 receiver 参数类型对 M1 和 M2 的影响。

首先，当 receiver 参数的类型为 T 时：
当我们选择以 T 作为 receiver 参数类型时，M1 方法等价转换为`F1(t T)`。我们知道，Go 函数的参数采用的是值拷贝传递，也就是说，F1 函数体中的 t 是 T 类型实例的一个副本。这样，我们在 F1 函数的实现中对参数 t 做任何修改，都只会影响副本，而不会影响到原 T 类型实例。

据此我们可以得出结论：当我们的方法 M1 采用类型为 T 的 receiver 参数时，代表 T 类型实例的 receiver 参数以值传递方式传递到 M1 方法体中的，实际上是 **T 类型实例的副本**，M1 方法体中对副本的任何修改操作，都不会影响到原 T 类型实例。

第二，当 receiver 参数的类型为 *T 时：

当我们选择以 *T 作为 receiver 参数类型时，M2 方法等价转换为`F2(t *T)`。同上面分析，我们传递给 F2 函数的 t 是 T 类型实例的地址，这样 F2 函数体中对参数 t 做的任何修改，都会反映到原 T 类型实例上。

据此我们也可以得出结论：当我们的方法 M2 采用类型为 *T 的 receiver 参数时，代表 *T 类型实例的 receiver 参数以值传递方式传递到 M2 方法体中的，实际上是 **T 类型实例的地址**，M2 方法体通过该地址可以对原 T 类型实例进行任何修改操作。

我们再通过一个更直观的例子，证明一下上面这个分析结果，看一下 Go 方法选择不同的 receiver 类型对原类型实例的影响：

复制代码

```
package main

type T struct {
    a int
}

func (t T) M1() {
    t.a = 10
}

func (t *T) M2() {
    t.a = 11
}

func main() {
    var t T
    println(t.a) // 0

    t.M1()
    println(t.a) // 0

    p := &t
    p.M2()
    println(t.a) // 11
}
```

在这个示例中，我们为基类型 T 定义了两个方法 M1 和 M2，其中 M1 的 receiver 参数类型为 T，而 M2 的 receiver 参数类型为 *T。M1 和 M2 方法体都通过 receiver 参数 t 对 t 的字段 a 进行了修改。

但运行这个示例程序后，我们看到，方法 M1 由于使用了 T 作为 receiver 参数类型，它在方法体中修改的仅仅是 T 类型实例 t 的副本，原实例并没有受到影响。因此 M1 调用后，输出 t.a 的值仍为 0。

而方法 M2 呢，由于使用了 *T 作为 receiver 参数类型，它在方法体中通过 t 修改的是实例本身，因此 M2 调用后，t.a 的值变为了 11，这些输出结果与我们前面的分析是一致的。

了解了不同类型的 receiver 参数对 Go 方法的影响后，我们就可以总结一下，日常编码中选择 receiver 的参数类型的时候，我们可以参考哪些原则。

选择 receiver 参数类型的第一个原则

基于上面的影响分析，我们可以得到选择 receiver 参数类型的第一个原则：**如果 Go 方法要把对 receiver 参数代表的类型实例的修改，反映到原类型实例上，那么我们应该选择 *T 作为 receiver 参数的类型。**

这个原则似乎很好掌握，不过这个时候，你可能会有个疑问：如果我们选择了 *T 作为 Go 方法 receiver 参数的类型，那么我们是不是只能通过 *T 类型变量调用该方法，而不能通过 T 类型变量调用了呢？这个问题恰恰也是上节课我们遗留的一个问题。我们改造一下上面例子看一下：

复制代码

```
type T struct {
    a int
}

func (t T) M1() {
    t.a = 10
}

func (t *T) M2() {
    t.a = 11
}

func main() {
    var t1 T
    println(t1.a) // 0
    t1.M1()
    println(t1.a) // 0
    t1.M2()
```

```
19      println(t1.a) // 11
20
21      var t2 = &T{}
22      println(t2.a) // 0
23      t2.M1()
24      println(t2.a) // 0
25      t2.M2()
26      println(t2.a) // 11
27  }
```

我们先来看看类型为 T 的实例 t1。我们看到它不仅可以调用 receiver 参数类型为 T 的方法 M1，它还可以直接调用 receiver 参数类型为 *T 的方法 M2，并且调用完 M2 方法后，t1.a 的值被修改为 11 了。

其实，T 类型的实例 t1 之所以可以调用 receiver 参数类型为 *T 的方法 M2，都是 Go 编译器在背后自动进行转换的结果。或者说，t1.M2() 这种用法是 Go 提供的“语法糖”：Go 判断 t1 的类型为 T，也就是与方法 M2 的 receiver 参数类型 *T 不一致后，会自动将`t1.M2()`转换为`(&t1).M2()`。

同理，类型为 *T 的实例 t2，它不仅可以调用 receiver 参数类型为 *T 的方法 M2，还可以调用 receiver 参数类型为 T 的方法 M1，这同样是因为 Go 编译器在背后做了转换。也就是，Go 判断 t2 的类型为 *T，与方法 M1 的 receiver 参数类型 T 不一致，就会自动将`t2.M1()`转换为`(*t2).M1()`。

通过这个实例，我们知道了这样一个结论：**无论是 T 类型实例，还是 *T 类型实例，都既可以调用 receiver 为 T 类型的方法，也可以调用 receiver 为 *T 类型的方法**。这样，我们在为方法选择 receiver 参数的类型的时候，就不需要担心这个方法不能被与 receiver 参数类型不一致的类型实例调用了。

选择 receiver 参数类型的第二个原则

前面我们第一个原则说的是，当我们要在方法中对 receiver 参数代表的类型实例进行修改，那我们要为 receiver 参数选择 *T 类型，但是如果我们不需要在方法中对类型实例进行修改呢？这个时候我们是为 receiver 参数选择 T 类型还是 *T 类型呢？

这也得分情况。一般情况下，我们通常会为 receiver 参数选择 T 类型，因为这样可以缩窄外部修改类型实例内部状态的“接触面”，也就是尽量少暴露可以修改类型内部状态的方

法。

不过也有一个例外需要你特别注意。考虑到 Go 方法调用时，receiver 参数是以值拷贝的形式传入方法中的。那么，**如果 receiver 参数类型的 size 较大**，以值拷贝形式传入就会导致较大的性能开销，这时我们选择 *T 作为 receiver 类型可能更好些。

以上这些可以作为我们**选择 receiver 参数类型的第二个原则**。

到这里，你可能会发出感慨：即便有两个原则，这似乎依旧很容易掌握！不要大意，这可没那么简单，这两条只是基础原则，还有一条更难理解的原则在下面呢。

不过在讲解这第三条原则之前，我们先要了解一个基本概念：**方法集合**（Method Set），它是我们理解第三条原则的前提。

方法集合

在了解方法集合是什么之前，我们先通过一个示例，直观了解一下为什么要有方法集合，它主要用来解决什么问题：

复制代码

```
type Interface interface {
    M1()
    M2()
}

type T struct{}

func (t T) M1()  {}
func (t *T) M2() {}

func main() {
    var t T
    var pt *T
    var i Interface

    i = pt
    i = t // cannot use t (type T) as type Interface in assignment: T does not
}
```

在这个例子中，我们定义了一个接口类型 Interface 以及一个自定义类型 T。Interface 接口类型包含了两个方法 M1 和 M2，它们的基类型都是 T，但它们的 receiver 参数类型不同，一个为 T，另一个为 *T。在 main 函数中，我们分别将 T 类型实例 t 和 *T 类型实例 pt 赋值给 Interface 类型变量 i。

运行一下这个示例程序，我们在`i = t`这一行会得到 Go 编译器的错误提示，Go 编译器提示我们：**T 没有实现 Interface 类型方法列表中的 M2，因此类型 T 的实例 t 不能赋值给 Interface 变量**。

可是，为什么呀？为什么 *T 类型的 pt 可以被正常赋值给 Interface 类型变量 i，而 T 类型的 t 就不行呢？如果说 T 类型是因为只实现了 M1 方法，未实现 M2 方法而不满足 Interface 类型的要求，那么 *T 类型也只是实现了 M2 方法，并没有实现 M1 方法啊？

有些事情并不是表面看起来这个样子的。了解方法集合后，这个问题就迎刃而解了。同时，**方法集合也是用来判断一个类型是否实现了某接口类型的唯一手段**，可以说，“**方法集合决定了接口实现**”。更具体的分析，我们等会儿再讲。

那么，什么是类型的方法集合呢？

Go 中任何一个类型都有属于自己的方法集合，或者说方法集合是 Go 类型的一个“属性”。但不是所有类型都有自己的方法呀，比如 int 类型就没有。所以，对于没有定义方法的 Go 类型，我们称其拥有空方法集合。

接口类型相对特殊，它只会列出代表接口的方法列表，不会具体定义某个方法，它的方法集合就是它的方法列表中的所有方法，我们可以一目了然地看到。因此，我们下面重点讲解的是非接口类型的方法集合。

为了方便查看一个非接口类型的方法集合，我这里提供了一个函数 dumpMethodSet，用于输出一个非接口类型的方法集合：

复制代码

```
1 func dumpMethodSet(i interface{}) {
2     dynTyp := reflect.TypeOf(i)
3
4     if dynTyp == nil {
5         fmt.Printf("there is no dynamic type\n")
```

```
        return
    }

    n := dynTyp.NumMethod()
    if n == 0 {
        fmt.Printf("%s's method set is empty!\n", dynTyp)
        return
    }

    fmt.Printf("%s's method set:\n", dynTyp)
    for j := 0; j < n; j++ {
        fmt.Println("-", dynTyp.Method(j).Name)
    }
    fmt.Printf("\n")
}
```

下面我们利用这个函数，试着输出一下 Go 原生类型以及自定义类型的方法集合，看下面代码：

复制代码

```
type T struct{}

func (T) M1() {}
func (T) M2() {}

func (*T) M3() {}
func (*T) M4() {}

func main() {
    var n int
    dumpMethodSet(n)
    dumpMethodSet(&n)

    var t T
    dumpMethodSet(t)
    dumpMethodSet(&t)
}
```

运行这段代码，我们得到如下结果：

复制代码

```
int's method set is empty!
*int's method set is empty!
main.T's method set:
- M1
```

```
- M2

*main.T's method set:
- M1
- M2
- M3
- M4
```

我们看到以 int、*int 为代表的 Go 原生类型由于没有定义方法，所以它们的方法集合都是空的。自定义类型 T 定义了方法 M1 和 M2，因此它的方法集合包含了 M1 和 M2，也符合我们预期。但 *T 的方法集合中除了预期的 M3 和 M4 之外，居然还包含了类型 T 的方法 M1 和 M2！

不过，这里程序的输出并没有错误。

这是因为，Go 语言规定，*T 类型的方法集合包含所有以 *T 为 receiver 参数类型的方法，以及所有以 T 为 receiver 参数类型的方法。这就是这个示例中为何 *T 类型的方法集合包含四个方法的原因。

这个时候，你是不是也找到了前面那个示例中为何`i = pt`没有报编译错误的原因了呢？我们同样可以使用 dumpMethodSet 工具函数，输出一下那个例子中 pt 与 t 各自所属类型的方法集合：

复制代码

```
type Interface interface {
    M1()
    M2()
}

type T struct{}

func (t T) M1()  {}
func (t *T) M2() {}

func main() {
    var t T
    var pt *T
    dumpMethodSet(t)
    dumpMethodSet(pt)
}
```

运行上述代码，我们得到如下结果：

复制代码

```
main.T's method set:
- M1

*main.T's method set:
- M1
- M2
```

通过这个输出结果，我们可以一目了然地看到 T、*T 各自的方法集合。

我们看到，T 类型的方法集合中只包含 M1，没有 Interface 类型方法集合中的 M2 方法，这就是 Go 编译器认为变量 t 不能赋值给 Interface 类型变量的原因。

在输出的结果中，我们还看到 *T 类型的方法集合除了包含它自身定义的 M2 方法外，还包含了 T 类型定义的 M1 方法，*T 的方法集合与 Interface 接口类型的方法集合是一样的，因此 pt 可以被赋值给 Interface 接口类型的变量 i。

到这里，我们已经知道了所谓的**方法集合决定接口实现**的含义就是：如果某类型 T 的方法集合与某接口类型的方法集合相同，或者类型 T 的方法集合是接口类型 I 方法集合的超集，那么我们就说这个类型 T 实现了接口 I。或者说，方法集合这个概念在 Go 语言中的主要用途，就是用来判断某个类型是否实现了某个接口。

有了方法集合的概念做铺垫，选择 receiver 参数类型的第三个原则已经呼之欲出了，下面我们就来看看这条原则的具体内容。

选择 receiver 参数类型的第三个原则

理解了方法集合后，我们再理解第三个原则的内容就不难了。这个原则的选择依据就是 **T 类型是否需要实现某个接口**。

如果 **T 类型需要实现某个接口**，那我们就要使用 T 作为 receiver 参数的类型，来满足接口类型方法集合中的所有方法。

如果 T 不需要实现某一接口，但 *T 需要实现该接口，那么根据方法集合概念，*T 的方法集合是包含 T 的方法集合的，这样我们在确定 Go 方法的 receiver 的类型时，参考原则一和原则二就可以了。

如果说前面的两个原则更多聚焦于类型内部，从单个方法的实现层面考虑，那么这第三个原则则是更多从全局的设计层面考虑，聚焦于这个类型与接口类型间的耦合关系。

小结

好了，今天的课讲到这里就结束了，现在我们一起来回顾一下吧。

我们前面已经知道，Go 方法本质上也是函数。所以 Go 方法设计的多数地方，都可以借鉴函数设计的相关内容。唯独 Go 方法的 receiver 部分，我们是没有现成经验可循的。这一讲中，我们主要学习的就是如何为 Go 方法的 receiver 参数选择类型。

我们先了解了不同类型的 receiver 参数对 Go 方法行为的影响，这是我们进行 receiver 参数选型的前提。

在这个前提下，我们提出了 receiver 参数选型的三个经验原则，虽然课程中我们是按原则一到三的顺序讲解的，**但实际进行 Go 方法设计时，我们首先应该考虑的是原则三，即 T 类型是否要实现某一接口。**

如果 T 类型需要实现某一接口的全部方法，那么我们就需要使用 T 作为 receiver 参数的类型来满足接口类型方法集合中的所有方法。

如果 T 类型不需要实现某一接口，那么我们就可以参考原则一和原则二来为 receiver 参数选择类型了。也就是，如果 Go 方法要把对 receiver 参数所代表的类型实例的修改反映到原类型实例上，那么我们应该选择 *T 作为 receiver 参数的类型。否则通常我们会为 receiver 参数选择 T 类型，这样可以减少外部修改类型实例内部状态的“渠道”。除非 receiver 参数类型的 size 较大，考虑到传值的较大性能开销，选择 *T 作为 receiver 类型可能更适合。

在理解原则三时，我们还介绍了 Go 语言中的一个重要概念：**方法集合**。它在 Go 语言中的主要用途就是判断某个类型是否实现了某个接口。方法集合像“胶水”一样，将自定义

类型与接口隐式地“粘结”在一起，我们后面理解带有类型嵌入的类型时还会借助这个概念。

思考题

方法集合是一个很重要也很实用的概念，我们在下一节课还会用到这个概念帮助我们理解具体的问题。所以这里，我给你出了一道与方法集合有关的预习题。

如果一个类型 T 包含两个方法 M1 和 M2：

复制代码

```
type T struct{}

func (T) M1()
func (T) M2()
```

然后，我们再使用类型定义语法，又基于类型 T 定义了一个新类型 S：

复制代码

```
type S T
```

那么，这个 S 类型包含哪些方法呢？*S 类型又包含哪些方法呢？请你自己分析一下，然后借助 dumpMethodSet 函数来验证一下你的结论。

欢迎你把这节课分享给更多对 Go 方法感兴趣的朋友。我是 Tony Bai，我们下节课见。

上一篇 24 | 方法：理解“方法”的本质

下一篇 期中测试 | 一起检验下你的学习成果吧

精选留言 (6)

写留言

Geek_99b47c
2021-12-10

S 类型 和 *S 类型都没有包含方法，因为type S T 定义了一个新类型。
但是如果用 type S = T 则S和*S类型都包含两个方法。

在下宝龙、
2021-12-10

S类型和*S类型都是 空方法，因为S是新的类型，它不能调用T的方法，必须显示转换之后才可以调用，所以本身的S或*S类型都不具有任何的方法

展开

叶鑫
2021-12-10

真是太棒了

展开


liaomars
2021-12-10

老师：
如果 T 类型需要实现某个接口，那我们就要使用 T 作为 receiver 参数的类型，来满足接口类型方法集合中的所有方法。
这段描述感觉不对，根据上面举的例子来说，应该是使用 *T 作为 receiver参数的类型，来满足接口类型方法集合中的所有方法。

展开

共 1 条评论 >

无双
2021-12-10

可以讲一下go的指针吗？

展开

共 1 条评论 >

罗杰
2021-12-10

其实相比 Rust，Go 的糖更少，而且时而多，时而少，让开发者会很困惑，甚至前后矛盾。*T 和 T调用方法时编译器互相转换，哇，真贴心，真舒服。但是方法集合，又被 Go 反手打了一巴掌。的确解决了 C 语言的诸多问题，但对比 Rust 的一些处理方案，的确会让人不爽。

展开