# 程序语言的常见设计错误(1) - 片面追求短小

我经常以自己写"非常短小"的代码为豪。有一些人听了之后很赞赏,然后说他也很喜欢写短小的代码,接着就开始说 C 语言其实有很多巧妙的设计,可以让代码变得非常短小。然后我才发现,这些人所谓的"短小"跟我所说的"短小"完 全不是一回事。

我的程序的"短小"是建立在语义明确,概念清晰的基础上的。在此基础上,我力求去掉冗余的,绕弯子的,混淆的代码,让程序更加直接,更加高效的表达我心中设想的"模型"。这是一种在概念级别的优化,而程序的短小精悍只是它的一种"表象"。就像是整理一团电线,并不是把它们揉成一团然后塞进一个盒子里就好。这样的做法只会给你以后的工作带来更大的麻烦,而且还有安全隐患。

所以我的这种短小往往是在语义和逻辑 层面的,而不是在语法上死抠几行代码。我绝不会为了程序显得短小而让它变得难以理解或者容易出错。相反,很多其它人所追求的短小,却是盲目的而没有原则的。在很多时候这些小伎俩都只是在语法层面,比如想办法把两行代码"搓"成一行。可以说,这种"片面追求短小"的错误倾向,造就了一批语言设计上的错误,以及一批"擅长于"使用这些错误的程序员。

现在我举几个简单的"片面追求短小"的语言设计。

## 自增减操作

很多语言里都有 i++ 和 ++i 这两个"自增"操作和 i-- 和 --i 这两个"自减"操作(下文合称"自增减操作"。很多人喜欢在代码里使用自增减操作,因为这样可以"节省一行代码"。殊不知,节省掉的那区区几行代码比起由此带来的混淆和错误,其实是九牛之一毛。

从理论上讲,自增减操作本身就是错误的设计。因为它们把对变量的"读"和"写"两种根本不同的操作,毫无原则的合并在一起。这种对读写操作的混淆不清,带来了非常难以发现的错误。相反,一种等价的,"笨"一点的写法,i=i+1,不但更易理解,而且在逻辑上更加清晰。

有些人很在乎 i++ 与 ++i 的区别,去追究 (i++) + (++i) 这类表达式的含义,追究 i++ 与 ++i 谁的效率更高。这些其实都是徒劳的。比如,i++ 与 ++i 的效率差别,其实来自于早期 C 编译器的愚蠢。因为 i++ 需要在增加之后返回 i 原来的值,所以它其实被编译为:

```
(tmp = i, i = i + 1, tmp)
```

#### 但是在

```
for (int i = 0; i < max; i++)
```

这样的语句中,其实你并不需要在 i++ 之后得到它自增前的值。所以有人说,在这里应该用 ++i 而不是 i++,否则你就会浪费一次对中间变量 tmp 的赋值。而其实呢,一个良好设计的编译器应该在两种情况下都生成相同的代码。这是因为在 i++ 的情况,代码其实先被转化为:

```
for (int i = 0; i < max; (tmp = i, i = i + 1, tmp))
```

由于 tmp 这个临时变量从来没被用过,所以它会被编译器的"dead code elimination"消去。所以编译器最后实际上得到了:

```
for (int i = 0; i < max; i = i + 1)
```

所以,"精通"这些细微的问题,并不能让你成为一个好的程序员。很多人所认为的高明的技巧,经常都是因为早期系统设计的缺陷所致。一旦这些系统被改进,这些技巧就没什么用处了。

真正正确的做法其实是:完全不使用自增减操作,因为它们本来就是错误的设计。

好了,一个小小的例子,也许已经让你意识到了片面追求短小程序所带来的认知上,时间上的代价。很可惜的是,程序语言的设计者们仍然在继续为此犯下类似的错误。一些新的语言加入了很多类似的旨在"缩短代码","减少打字量"的雕虫小技。也许有一天你会发现,这些雕虫小技所带来的,除了短暂的兴奋,其实都是在浪费你的时间。

### 赋值语句返回值

在几乎所有像 C,C++,Java 的语言里,赋值语句都可以被作为值。之所以设计成这样,是因为你就可以写这样的 代码:

```
if (y = 0) \{ ... \}
```

而不是

```
y = 0;
if (y) \{ ... \}
```

程序好像缩短了一行,然而,这种写法经常引起一种常见的错误,那就是为了写 if (y == 0) { ... } 而把 == 比较操作少打了一个 =,变成了 if (y = 0) { ... }。很多人犯这个错误,是因为数学里的 = 就是比较两个值是否相等的意思。

不小心打错一个字,就让程序出现一个 bug。不管 y 原来的值是多少,经过这个"条件"之后,y 的值都会变成 0。所以这个判断语句会一直都为"假",而且一声不吭的改变了 y 的值。这种 bug 相当难以发现。这就是另一个例子,说明片面追求短小带来的不应有的问题。

正确的做法是什么呢?在一个类型完备的语言里面,像 y=0 这样的赋值语句,其实是不应该可以返回一个值的,所以它不允许你写:

```
x = y = 0
或者
if (y = 0) { ... }
```

这样的代码。

x = y = 0 的工作原理其实是这样:经过 parser 它其实变成了 x = (y = 0) (因为 = 操作符是"右结合"的)。 x = (y = 0) 这个表达式也就是说 x 被赋值为 y = 0 的值。注意,我说的是 y = 0 这整个表达式的值,而不是 y 的值。所以这里的 y = 0 既有副作用又是值,它返回 y 的"新值"。

正确的做法其实是:y=0 不应该具有一个值。它的作用应该是"赋值"这种"动作",而不应该具有任何"值"。即使牵强一点硬说它有值,它的值也应该是 void。这样一来 x=y=0 和 if (y=0) 就会因为"类型不匹配"而被编译器拒绝接受,从而避免了可能出现的错误。

仔细想一想,其实 x = y = 0 和 if (y = 0) 带来了非常少的好处,但它们带来的问题却耗费了不知道多少人多少时间。这就是我为什么把它们叫做"小聪明"。

# 思考题:

1. Google 公司的代码规范里面规定,在任何情况下 for 语句和 if 语句之后必须写花括号,即使 C 和 Java 允许你在其只包含一行代码的时候省略它们。比如,你不能这样写

2. 当我第二次到 Google 实习的时候,发现我一年前给他们写的代码,很多被调整了结构。几乎所有如下结构的代码:

```
if (condition) {
  return x;
} else {
  return y;
}

都被人改成了:

if (condition) {
  return x;
}

return y;
```

请问这里省略了一个 else 和两个花括号,会带来什么好处或者坏处?

(提示,改过之后的代码不如原来的好。为什么?)

3. 根据本文对于自增减操作的看法,再参考传统的图灵机的设计,你是否发现图灵机的设计存在类似的问题?你如

何改造图灵机,使得它不再存在这种问题?

(提示,注意图灵机的"读写头"。)

4. 参考这个《Go 语言 $\lambda$ 门指南》,看看你是否能从中发现由于"片面追求短小"而产生的,别的语言里都没有的设计错误?