对 Go 语言的综合评价

以前写过一些对 Go 语言的负面评价。现在看来，虽然那些评价大部分属实，然而却由于言辞激烈，没有点明具体问题，难以让某些人信服。在经过几个月实际使用 Go 来构造网站之后，我觉得现在是时候对它作一些更加“客观”的评价了。

定位和优点

Go 比起 C 和 C++ 确实有它的优点，这是很显然的事情。它比起 Java 也有少数优点，然而相对而言更多是不足之处。所以我对 Go 的偏好在比 Java 稍低一点的位置。

Go 语言比起 C，C++ 的强项，当然是它的简单性和垃圾回收。由于 C 和 C++ 的设计有很多历史遗留问题，所以 Go 看起来确实更加优雅和简单。比起那些大量使用设计模式的 Java 代码，Go 语言的代码也似乎更简单一些。另外，Go 的垃圾回收机制比起 C 和 C++ 的全手动内存管理来说，大大降低了程序员的头脑负担。

但是请注意，这里的所谓“优点”都是相对于 C 之类的语言而言的。如果比起另外的一些语言，Go 的这种优点也许就很微不足道，甚至是历史的倒退了。

语法

Go 的简单性体现在它的语法和语义的某些方面。Go 的语法比 C 要稍好一些，有少数比 Java 更加方便的设计，然而却也有“倒退”的地方。而且这些倒退还不被很多人认为是倒退，反而认为是进步。我现在举出暂时能想得起来的几个方面：

- 进步：Go 有语法支持一种类似 struct literal 的构造，比如你可以写这样的代码来构造一个 S struct：

  ```
  S { x: 1, y: 2, }
  ```

  这比起 Java 只能用构造函数来创建对象是一个不错的方便性上的改进。这些东西可能借鉴于 JavaScript 等语言的设计。

- 倒退：类型放在变量后面，却没有分隔符。如果变量和它的类型写成像 Pascal 那样的，比如 `x : int`，那也许还好。然而 Go 的写法却是 `x int`，没有那个冒号，而且允许使用 `x, y int` 这样的写法。这种语法跟 var，函数参数组合在一起之后，就产生了扰乱视线的效果。比如你可以写一个函数是这样开头的：

  ```
  func foo(s string, x, y, z int, c bool) {
    ...
  }
  ```

  注意 x, y, z 那个位置，其实是很混淆的。因为看见 `x` 的时候我不能立即从后面那个符号（`, y`）看到它是什么类型。所以在 Go 里面我推荐的写法是把 `x` 和 `y` 完全分开，就像 C 和 Java 那样，不过类型写在后面：

  ```
  func foo(s string, x int, y int, z int, c bool) {
    ...
  }
  ```

  这样一来就比较清晰了，虽然我愿意再多写一些冒号。每一个参数都是“名字 类型”的格式，所以我一眼就看到 x 是 int。虽然多打几个字，然而节省的是“眼球 parse 代码”的开销。

- 倒退：类型语法。Go 使用像 `[]string` 这样的语法来表示类型。很多人说这种语法非常“一致”，但经过一段时间我却没有发现他们所谓的一致性在哪里。其实这样的语法很难读，因为类型的各部分之间没有明确的分隔标识符，如果和其他一些符号，比如 * 搭配在一起，你就需要知道一些优先级规则，然后费比较大的功夫去做“眼球 parse”。比如，在 Go 代码里你经常看到 `[]*Struct` 这样的类型，注意 `*Struct` 要先结合在一起，再作为 `[]` 的“类型参数”。这种语法缺乏足够的分隔符作为阅读的“边界信号”，一旦后面的类型变得复杂，就很难阅读了。比如，你可以有 `*[]*Struct` 或者 `*[]*pkg.Struct` 这样的类型。所以这其实还不如像 C++ 的 `vector<struct*>` 这样的写法，也就更不如 Java 或者 Typed Racket 的类型写法来得清晰和简单。

- 倒退：过度地“语法重载”，比如 switch, for 等关键字。Go 的 switch 关键字其实包含了两种不同的东西。它可以是 C 里面的普通的 switch（Scheme 的 case），也可以是像 Scheme 的 cond 那样的嵌套分支语句。这两种语句其实是语义完全不同的，然而 Go 的设计者为了显得简单，把它们合二为一，而其实引起了更大的混淆。这是因为，就算你把它们合二为一，它们仍然是两种不同的语义结构。把它们合并的结果是，每次看到 switch 你都需要从它们“头部”的不同点把这两种不同的结构区分开来，增加了人脑的开销。正确的作法是把它们分开，就像 Scheme 那样。其实我设计语言的时候有时候也犯同样的错误，以为两个东西“本质”上是一样的，所以合二为一，结果经过一段时间，发现其实是不一样的。所以不要小看了 Scheme，很多你认为是“新想法”的东西，其实早就被它那非常严谨的委员会给抛弃在了历史的长河中。

Go 语言里面还有其他一些语法设计问题，比如强制把 { 放在一行之后而且不能换行，if 语句的判断开头可以嵌套赋

值操作等等。这些试图让程序显得短小的作法，其实反而降低了程序理解的流畅度。

所以总而言之，Go 的语法很难被叫做“简单”或者“优雅”，它的简单性其实在 Java 之下。

工具链

Go 提供了一些比较方便的工具。比如 gofmt，godef 等，使得 Go 代码的编程比起单用 Emacs 或者 VIM 来编辑 C 和 C++ 来说是一个进步。使用 Emacs 编辑 Go 就已经能实现某些 IDE 才有的功能，比如精确的定义跳转等等。

这些工具虽然好用，但比起像 Eclipse, IntelliJ 和 Visual Studio 这样的 IDE，差距还是相当大的。比起 IDE，Go 的工具链缺乏各种最基本的功能，比如列出引用了某个变量的所有位置，重命名等 refactor 功能，好用的 debugger （GDB 不算好用）等等。

Go 的各种工具感觉都不大成熟，有时候你发现有好几个不同的 package 用于解决同一个问题，搞不清楚哪一个好些。而且这些东西配置起来不是那么的可靠和简单，都需要折腾。每一个小功能你都得从各处去寻找 package 来配置。有些时候一个工具配置了之后其实没有起作用，要等你摸索好半天才发现问题出现在哪里。这种没有组织，没有计划的工具设计，是很难超过专业 IDE 厂商的连贯性的。

Go 提供了方便的 package 机制，可以直接 import 某个 GitHub repository 里的 Go 代码。不过我发现很多时候这种 package 机制带来的更多是麻烦事和依赖关系。所以 Go 的推崇者们又设计了一些像 godep 的工具，用来绕过这些问题，结果 godep 自己也引起一些稀奇古怪的问题，导致有时候新的代码其实没有被编译，产生莫名其妙的错误信息（可能是由于 godep 的 bug）。

我发现很多人看到这些工具之后总是很狂热的认为它们就能让 Go 语言一统天下，其实还差得非常之远。而且如此年轻的语言就已经出现这么多的问题，我觉得所有这些麻烦事累积下来，多年以后恐怕够呛。

内存管理

比起 C 和 C++ 完全手动的内存管理方式，Go 有垃圾回收（GC）机制。这种机制大大减轻了程序员的头脑负担和程序出错的机会，所以 Go 对于 C/C++ 是一个进步。

然而进步也是相对的。Go 的垃圾回收器是一个非常原始的 mark-and-sweep，这比起像 Java，OCaml 和 Chez Scheme 之类的语言实现，其实还处于起步阶段。

当然如果真的遇到 GC 性能问题，通过大量的 tuning，你可以部分的改善内存回收的效率。我也看到有人写过一些文章介绍他们如何做这些事情，然而这种文章的存在说明了 Go 的垃圾回收还非常不成熟。GC 这种事情我觉得大部分时候不应该是让程序员来操心的，否则就失去了 GC 比起手动管理的很多优势。所以 Go 代码想要在实时性比较高的场合，还是有很长的路要走的。

由于缺乏先进的 GC，却又带有高级的抽象，所以 Go 其实没法取代 C 和 C++ 来构造底层系统。Go 语言的定位对我来说越来越模糊。

没有“generics”

比起 C++ 和 Java 来说，Go 缺乏 generics。虽然有人讨厌 Java 的 generics，然而它本身却不是个坏东西。Generics 其实就是 Haskell 等函数式语言里面所谓的 parametric polymorphism，是一种非常有用的东西，不过被 Java 抄去之后有时候没有做得全对。因为 generics 可以让你用同一块代码来处理多种不同的数据类型，它为避免重复，方便替换复杂数据结构等提供了方便。

由于 Go 没有 generics，所以你不得不重复写很多函数，每一个只有类型不同。或者你可以用空 interface {}，然而这个东西其实就相当于 C 的 void* 指针。使用它之后，代码的类型无法被静态的检查，所以其实它并没有 generics 来的严谨。

比起 Java，Go 的很多数据结构都是“hard code”进了语言里面，甚至创造了特殊的关键字和语法来构造它们（比如哈希表）。一旦遇到用户需要自己定义类似的数据结构，就需要把大量代码重写一遍。而且由于没有类似 Java collections 的东西，无法方便的换掉复杂的数据结构。这对于构造像 PySonar 那样需要大量实验才能选择正确的数据结构，需要实现特殊的哈希表等数据结构的程序来说，Go 语言的这些缺失会是一个非常大的障碍。

缺少 generics 是一个问题，然而更严重的问题是 Go 的设计者及其社区对于这类语言特性的盲目排斥。当你提到这些，Go 支持者就会以一种蔑视的态度告诉你：“我看不到 generics 有什么用！”这种态度比起语言本身的缺点来说更加有害。在经过了很长一段时间之后 Go 语言的设计者们开始考虑加入 generics，然后由于 Go 的语法设计偷工减料，再加上由于缺乏 generics 而产生的特例（比如 Go 的 map 的语法设计）已经被大量使用，我觉得要加入 generics 的难度已经非常大。

Go 和 Unix 系统一样，在出现的早期就已经因为不吸取前人的教训，背上了沉重的历史包袱。

多返回值

很多人都觉得 Go 的多返回值设计是一个进步，然而这里面却有很多蹊跷的东西。且不说这根本不是什么新东西（Scheme 很早就有了多返回值 let-values），Go 的多返回值却被大量的用在了错误的地方—Go 利用多返回值来表示出错信息。比如 Go 代码里最常见的结构就是：

```
ret, err := foo(x, y, z)
if err != nil {
        return err
}
```

如果 `foo` 的调用产生了错误，那么 `err` 就不是 nil。Go 要求你在定义了变量之后必须使用它，否则报错。这样它“碰巧”避免了出现错误 `err` 而不检查的情况。否则如果你想忽略错误，就必须写成

```
ret, _ := foo(x, y, z)
```

这样当 foo 出错的时候，程序就会自动在那个位置当掉。

不得不说，这种“歪打正着”的做法虽然貌似可行，从类型系统角度看，却是非常不严谨的。因为它根本不是为了这个目的而设计的，所以你可以比较容易的想出各种办法让它失效。而且由于编译器只检查 `err` 是否被“使用”，却不检查你是否检查了“所有”可能出现的错误类型。比如，如果 foo 可能返回两种错误 Error1 和 Error2，你没法保证调用者完全排除了这两种错误的可能性之后才使用数据。所以这种错误检查机制其实还不如 Java 的 exception 来的严谨。

另外，`ret` 和 `err` 同时被定义，而每次只有其中一个不是 nil，这种“或”的关系并不是靠编译器来保障，而是靠程序员的“约定俗成”。这样当 `err` 不是 nil 的时候，`ret` 其实也可以不是 nil。这些组合带来了挺多的混淆，让你每次看到 return 的地方都不确信它到底想返回一个错误还是一个有效值。如果你意识到这种“或”关系其实意味着你只应该用一个返回值来表示它们，你就知道其实 Go 误用了多返回值来表示可能的错误。

其实如果一个语言有了像 Typed Racket 和 PySonar 所支持的 “union type”类型系统，这种多返回值就没有意义了。因为如果有了 union type，你就可以只用一个返回值来表示有效数据或者错误。比如你可以写一个类型叫做 {String, FileNotFound}，用于表示一个值要么是 String，要么是 FileNotFound 错误。如果一个函数有可能返回错误，编译器就强制程序员检查所有可能出现的错误之后才能使用数据，从而可以完全避免以上的各种混淆情况。对 union type 有兴趣的人可以看看 Typed Racket，它拥有我迄今为止见过最强大的类型系统（超越了 Haskell）。

所以可以说，Go 的这种多返回值，其实是“歪打”打着了一半，然后换着法子继续歪打，而不是瞄准靶心。

接口

Go 采用了基于接口（interface）的面向对象设计，你可以使用接口来表达一些想要进行抽象的概念。

然而这种接口设计却不是没有问题的。首先跟 Java 不同，实现一个 Go 的接口不需要显式的声明（implements），所以你有可能“碰巧”实现了某个接口。这种不确定性对于理解程序来说是有反作用的。有时候你修改了一个函数之后就发现编译不通过，抱怨某个位置传递的不是某个需要的接口，然而出错信息却不能告诉你准确的原因。要经过一番摸索你才发现你的 struct 为什么不再实现之前定义的一个接口。

另外，有些人使用接口，很多时候不过是为了传递一些函数作为参数。我有时候不明白，这种对于函数式语言再简单不过的事情，在 Go 语言里面为什么要另外定义一个接口来实现。这使得程序不如函数式语言那么清晰明了，而且修改起来也很不方便。有很多冗余的名字要定义，冗余的工作要做。

举一个相关的例子就是 Go 的 Sort 函数。每一次需要对某种类型 `T` 的数组排序，比如 `[]string`，你都需要

1. 定义另外一个类型，通常叫做 `TSorter`，比如 `StringSorter`
2. 为这个 `StringSorter` 类型定义三个方法，分别叫做 `Len`, `Swap`, `Less`
3. 把你的类型比如 `[]string` cast 成 `StringSorter`
4. 调用 `sort.Sort` 对这个数组排序

想想 sort 在函数式语言里有多简单吧？比如，Scheme 和 OCaml 都可以直接这样写：

```
(sort '(3 4 1 2) <)
```

这里 Scheme 把函数 < 直接作为参数传给 sort 函数，而没有包装在什么接口里面。你发现了吗，Go 的那个 interface 里面的三个方法，其实本来应该作为三个参数直接传递给 Sort，但由于受到 design pattern 等思想的局限，Go 的设计者把它们“打包”作为接口来传递。而且由于 Go 没有 generics，你无法像函数式语言一样写这三个函数，接受比较的“元素”作为参数，而必须使用它们的“下标”。由于这些方法只接受下标作为参数，所以 Sort 只能对数组进行排序。另外由于 Go 的设计比较“底层”，所以你需要另外两个参数: len 和 swap。

其实这种基于接口的设计其实比起函数式语言，差距是很大的。比起 Java 的接口设计，也可以说是一个倒退。

goroutine

Goroutine 可以说是 Go 的最重要的特色。很多人使用 Go 就是听说 goroutine 能支持所谓的“大并发”。

首先这种大并发并不是什么新鲜东西。每个理解程序语言理论的人都知道 goroutine 其实就是一些用户级的“continuation”。系统级的 continuation 通常被叫做“进程”或者“线程”。Continuation 是函数式语言专家们再了解不过的东西了，比如我的前导师 Amr Sabry 就是关于 continuation 的顶级专家之一。

Node.js 那种 “callback hell”，其实就是函数式语言里面常用的一种手法，叫做 continuation passing style (CPS)。由于 Scheme 有 call/cc，所以从理论上讲，它可以不通过 CPS 样式的代码而实现大并发。所以函数式语言只要支持 continuation，就会很容易的实现大并发，也许还会更高效，更好用一些。比如 Scheme 的一个实现 Gambit-C 就可以被用来实现大并发的东西。Chez Scheme 也许也可以，不过还有待确认。

当然具体实现上的效率也许有区别，然而我只是说，goroutine 其实并不是像很多人想象的那样全新的，革命性的，独一无二的东西。只要有足够的动力，其它语言都能添加这个东西。

defer

Go 实现了 defer 函数，用于避免在函数出错后忘了收拾残局（cleanup）。然而我发现这种 defer 函数有被滥用的趋势。比如，有些人把那种不是 cleanup 的动作也做成 defer，到后来累积几个 defer 之后，你就不再能一眼看得清楚到底哪块代码先运行哪块后运行了。位置处于前面的代码居然可以在后来运行，违反了代码的自然位置顺序关系。

当然这可以怪程序员不明白 defer 的真正用途，然而一旦你有了这种东西就会有人想滥用它。那种急于试图利用一个语言的每种 feature 的人，特别喜欢干这种事情。这种问题恐怕需要很多年的经验之后，才会有人写成书来教育大家。在形成统一的“代码规范”以前，我预测 defer 仍然会被大量的滥用。

所以我们应该想一下，为了避免可能出现的资源泄漏，defer 带来的到底是利多还是弊多。

库代码

Go 的标准库的设计里面带有浓郁的 Unix 气息。比起 Java 之类的语言，它的库代码有很多不方便的地方。有时候引入了一些函数式语言的方式，但却由于 Unix 思维的限制，不但没能发挥函数式语言的优点，而且导致了很多理解的复杂性。

一个例子就是 Go 处理字符串的方式。在 Java 里每个字符串里包含的字符，缺省都是 Unicode 的“code point”。然而在 Go 里面 string 类型里面每个元素都是一个 byte，所以每次你都得把它 cast 成“rune”类型才能正确的遍历每个字符，然后 cast 回去。这种把任何东西都看成 byte 的方式，就是 Unix 的思维方式，它引起过度底层和复杂的代码。

HTML template 库

我使用过 Go 的 template library 来生成一些网页。这是一种“基本可用”的模板方式，然而比起很多其他成熟的技术，却是相当的不足的。让我比较惊讶的是，Go 的 template 里面夹带的代码，居然不是 Go 语言自己，而是一种表达能力相当弱的语言，有点像一种退化的 Lisp，只不过把括号换成了 `{ {...} }` 这样的东西。

比如你可以写这样的网页模板：

```
{ {define "Contents"} }
{ {if .Paragraph.Length} }
<p>{ {.Paragraph.Content} }</p>
{ {end} }
{ {end} }
```

由于每个模板接受一个 struct 作为填充的数据，你可以使用 `.Paragraph.Content` 这样的代码，然而这不但很丑陋，而且让模板不灵活，不好理解。你需要把需要的数据全都放进同一个结构才能从模板里面访问它们。

任何超过一行的代码，虽然也许这语言可以表达，一般人为了避免这语言的弱点，还是在 .go 文件里面写一些“帮助函数”。用它们产生数据放进结构，然后传给模板，才能够表达模板需要的一些信息。而这每个帮助函数又需要一定的“注册”信息才能被模板库找到。所以这些复杂性加起来，使得 Go 的 HTML 模板代码相当的麻烦和混乱。

听说有人在做一个新的 HTML 模板系统，可以支持直接的 Go 代码嵌入。这些工作刚刚起步，而且难说最后会做成什么样子。所以要做网站，恐怕还是最好使用其他语言比较成熟的框架。

总结

优雅和简单性都是相对而言的。虽然 Go 语言在很多方面超过了 C 和 C++，也在某些方面好于 Java，然而它其实是没法和 Python 的优雅性相比的，而 Python 在很多方面却又不如 Scheme 和 Haskell。所以总而言之，Go 的简单性和优雅程度属于中等偏下。

由于没有明显的优势，却又有各种其它语言里没有的问题，所以在实际工程中，我目前更倾向于使用 Java 这样的语言。我不觉得 Go 语言和它的工具链能够帮助我迅速的写出 PySonar 那样精密的代码。另外我还听说有人使用 Java 来实现大并发，并没发现比起 Go 有什么明显的不足。

Alan Perlis 说，语言设计不应该是把功能堆积起来，而应该努力地减少弱点。从这种角度来看，Go 语言引入了一两个新的功能，同时又引入了相当多的弱点。

Go 也许暂时在某些个别的情况有特殊的强项，可以单独用于优化系统的某些部分，但我不推荐使用 Go 来实现复杂的算法和整个的系统。