

大咖助阵 | Tony Bai: Go 程序员拥抱 C 语言简明指南

2022-03-29 Tony Bai

《深入C语言和程序运行原理》

课程介绍 >



讲述: Tony Bai

时长 27:42 大小 25.37M



你好，我是于航。这一讲是一期大咖加餐，我们邀请到了 Tony Bai 老师，来跟你聊聊 C 语言的一个优秀“后辈”，Go 语言的故事。

Go 在语法上跟 C 类似，但它却通过提供垃圾回收机制，从侧面解决了 C 程序容易发生内存泄露的问题，进而使得程序的构建变得更加简单。除此之外，Go 还提供了大量用于编写并发程序的内置工具和库，因此它被大量应用于构架需要满足高并发性能的软件中，比如你最熟悉的 Kubernetes。

通过这一讲加餐，你可以了解到 C 与 Go 这两种语言之间的相似性和区别，相信你一定能有所收获。

领资料

你好，我是 Tony Bai。

也许有同学对我比较熟悉，看过我在极客时间上的专栏 [《Tony Bai · Go 语言第一课》](#)，或者是关注了我的博客。那么，作为一个 Gopher，我怎么跑到这个 C 语言专栏做分享了呢？其实，在学习 Go 语言并成为一名 Go 程序员之前，我也曾是一名地地道道的 C 语言程序员。

大学毕业后，我就开始从事 C 语言后端服务开发工作，在电信增值领域摸爬滚打了十多年。不信的话，你可以去翻翻 [我的博客](#)，数一数我发的 C 语言相关文章是不是比关于 Go 的还多。一直到近几年，我才将工作中的主力语言从 C 切换到了 Go。不过这并不是 C 语言的问题，主要原因是我转换赛道了。我目前在智能网联汽车领域从事面向云原生平台的先行研发，而在云原生方面，新生代的 Go 语言有着更好的生态。

不过作为资深 C 程序员，C 语言已经在我身上打下了深深的烙印。虽然 Go 是我现在工作中的主力语言，但我仍然会每天阅读一些 C 开源项目的源码，每周还会写下数百行的 C 代码。在一些工作场景中，特别是在我参与先行研发一些车端中间件时，C 语言有着资源占用小、性能高的优势，这一点是 Go 目前还无法匹敌的。

正因为我有着 C 程序员和 Go 程序员的双重身份，接到这个加餐邀请时，我就想到了一个很适合聊的话题——在 Gopher（泛指 Go 程序员）与 C 语言之间“牵线搭桥”。在这门课的评论区里，我看到一些同学说，“正是因为学了 Go，所以我想学好 C”。如果你也对 Go 比较熟悉，那么恭喜你，这篇加餐简直是为你量身定制的：一个熟悉 Go 的程序员在学习 C 时需要注意的问题，还有可能会遇到的坑，我都替你总结好了。

当然，我知道还有一些对 Go 了解不多的同学，看到这里也别急着退出去。因为 C 和 Go 这两门语言的比较，本身就是一个很有意思的话题。今天的加餐，会涉及这两门语言的异同点，通过对 C 与 Go 语言特性的比较，你就能更好地理解“C 语言为什么设计成现在这样”。

C 语言是现代 IT 工业的根基

在比较 C 和 Go 之前，先说说我推荐 Gopher 学 C 的最重要原因吧：用一句话总结，C 语言在 IT 工业中的根基地位，是 Go 和其他语言目前都无法动摇的。

C 语言是由美国贝尔实验室的丹尼斯·里奇（Dennis Ritchie）以 Unix 发明人肯·汤普森（Ken Thompson）设计的 B 语言为基础而创建的高级编程语言。诞生于上个世纪（精确来说是 1972 年）的它，到今年（2022 年）已到了“知天命”的半百年纪。年纪大、设计久远一直是“C 语言过时论”兴起的根源，但如果你相信这一论断，那就大错特错了。下面，我来为你分析下个中缘由。

首先，我们说说 C 语言本身：**C 语言一直在演进，从未停下过脚步。**

领资料

虽然 C 语言之父丹尼斯·里奇不幸于 2011 年永远地离开了我们，但 C 语言早已成为 ANSI（美国国家标准学会）标准以及 ISO/IEC（国际标准化组织和国际电工委员会）标准，因此其演进也早已由标准委员会负责。我们来简单回顾一下 C 语言标准的演进过程：

- 1989 年，ANSI 发布了首个 C 语言标准，被称为 C89，又称 ANSI C。次年，ISO 和 IEC 把 ANSI C89 标准定为 C 语言的国际标准（ISO/IEC 9899:1990），又称 C90，它也是 C 语言的第一个官方版本；
- 1999 年，ISO 和 IEC 发布了 [C99 标准 \(ISO/IEC 9899:1999\)](#)，它是 C 语言的第二个官方版本；
- 2011 年，ISO 和 IEC 发布了 [C11 标准 \(ISO/IEC 9899:2011\)](#)，它是 C 语言的第三个官方版本；
- 2018 年，ISO 和 IEC 发布了 [C18 标准 \(ISO/IEC 9899:2018\)](#)，它是 C 语言的第四个官方版本。

目前，ISO/IEC 标准化委员会正在致力于 C2x 标准的改进与制定，预计它会在 2023 年发布。

其次，时至今日，C 语言的流行度仍然非常高。

著名编程语言排行榜 TIOBE 的数据显示，各大编程语言年度平均排名的总位次，C 语言多年来高居第一，如下图（图片来自 [TIOBE](#)）所示：

Programming Language	2022	2017	2012	2007	2002	1997	1992	1987
C	1	2	2	2	2	1	1	1
Python	2	5	8	7	12	28	-	-
Java	3	1	1	1	1	14	-	-
C++	4	3	3	3	3	2	2	4
C#	5	4	4	8	14	-	-	-
Visual Basic	6	15	-	-	-	-	-	-
JavaScript	7	7	10	9	9	21	-	-
Assembly language	8	10	-	-	-	-	-	-
PHP	9	6	5	5	8	-	-	-
SQL	10	-	-	-	34	-	-	-

这说明，无论是在过去还是现在，C 语言都是一门被广泛应用的工业级编程语言。



最后，也是最重要的一点是：**C 语言是现代 IT 工业的根基**，我们说 C 永远不会退出 IT 行业舞台也不为过。

如今，无论是普通消费者端的 Windows、macOS、Android、苹果 iOS，还是服务器端的 Linux、Unix 等操作系统，亦或是各个工业嵌入式领域的操作系统，其内核实现语言都是 C 语言。互联网时代所使用的主流 Web 服务器，比如 Nginx、Apache，以及主流数据库，比如 MySQL、Oracle、PostgreSQL 等，也都是使用 C 语言开发的杰作。可以说，现代人类每天都在跟由 C 语言实现的系统亲密接触，并且已经离不开这些系统了。回到我们程序员的日常，Git、SVN 等我们时刻在用的源码版本控制软件也都是由 C 语言实现的。

可以说，C 语言在 IT 工业中的根基地位，不光 Go 语言替代不了，C++、Rust 等系统编程语言也无法动摇，而且不仅短期如此，长期来看也是如此。

总之，C 语言具有紧凑、高效、移植性好、对内存的精细控制等优秀特性，这使得我们在任何时候学习它都不会过时。不过，我在这里推荐 Gopher 去了解和系统学习 C 语言，其实还有另一个原因。我们继续往下看。

C 与 Go 的相通之处：Gopher 拥抱 C 语言的“先天优势”

众所周知，Go 是在 C 语言的基础上衍生而来的，二者之间有很多相通之处，因此 Gopher 在学习 C 语言时是有“先天优势”的。接下来，我们具体看看 C 和 Go 的相通之处有哪些。

简单且语法同源

Go 语言以简单著称，而作为 **Go 先祖** 的 C 语言，入门门槛同样不高：Go 有 25 个关键字，C 有 32 个关键字（C89 标准），简洁程度在伯仲之间。C 语言曾长期作为高校计算机编程教育的首选编程语言，这与 C 的简单也不无关系。

和 Go 不同的是，C 语言是一个**小内核、大外延**的编程语言，其简单主要体现在小内核上了。这个“小内核”包括 C 基本语法与其标准库，我们可以快速掌握它。但需要注意的是，与 Go 语言“开箱即用、内容丰富”的标准库不同，**C 标准库**非常小（在 C11 标准之前甚至连 thread 库都不包含），所以掌握“小内核”后，在 LeetCode 平台上刷题是没有任何问题的，但要写出某一领域的工业级生产程序，我们还有很多外延知识技能要学习，比如并发原语、操作系统的系统调用，以及进程间通信等。



C 语言的这种简单很容易获得 Gopher 们的认同感。当年 Go 语言之父们在设计 Go 语言时，也是主要借鉴了 C 语言的语法。当然，这与他们深厚的 C 语言背景不无关系：肯·汤普森（Ken Thompson）是 Unix 之父，与丹尼斯·里奇共同设计了 C 语言；罗博·派克（Rob Pike）是贝尔实验室的资深研究员，参与了 Unix 系统的演进、Plan9 操作系统的开发，还是 UTF-8 编码的发明人；罗伯特·格瑞史莫（Robert Griesemer）也是用 C 语言手写 Java 虚拟机的大神级人物。

Go 的第一版编译器就是由肯·汤普森（Ken Thompson）用 C 语言实现的。并且，Go 语言的早期版本中，C 代码的比例还不小。以 Go 语言发布的第一个版本，[Go 1.0 版本](#)为例，我们通过 [loccount 工具](#)对其进行分析，会得到下面的结果：

复制代码

```
1 $loccount .
2 all          SLOC=460992  (100.00%) LLOC=193045  in 2746 files
3 Go           SLOC=256321  (55.60%) LLOC=109763  in 1983 files
4 C            SLOC=148001  (32.10%) LLOC=73458   in 368 files
5 HTML         SLOC=25080   (5.44%)  LLOC=0         in 57 files
6 asm          SLOC=10109   (2.19%)  LLOC=0         in 133 files
7 ... ..
```

这里我们看到，在 1.0 版本中，C 语言代码行数占据了 32.10% 的份额，这一份额直至 Go 1.5 版本实现自举后，才下降为不到 1%。

我当初对 Go “一见钟情”，其中一个主要原因就是 Go 与 C 语言的**语法同源**。相对应地，相信这种同源的语法也会让 Gopher 们喜欢上 C 语言。

静态编译且基础范式相同

除了语法同源，C 语言与 Go 语言的另一个相同点是，它们都是静态编译型语言。这意味着它们都有如下的语法特性：

- 变量与函数都要先声明后才能使用；
- 所有分配的内存块都要有对应的类型信息，并且在确定其类型信息后才能操作；
- 源码需要先编译链接后才能运行。

相似的编程逻辑与构建过程，让学习 C 语言的 Gopher 可以做到无缝衔接。



除此之外，Go 和 C 的基础编程范式都是命令式编程（imperative programming），即面向算法过程，由程序员通过编程告诉计算机应采取的动作。然后，计算机按程序指令执行一系列流程，生成特定的结果，就像菜谱指定了厨师做蛋糕时应遵循的一系列步骤一样。

从 Go 看 C，没有面向对象，没有函数式编程，没有泛型（Go 1.18 已加入），满眼都是类型与函数，可以说是相当亲切了。

错误处理机制如出一辙

对于后端编程语言来说，错误处理机制十分重要。如果两种语言的错误处理机制不同，那么这两种语言的代码整体语法风格很可能大不相同。

在 C 语言中，我们通常用一个类型为整型的函数返回值作为错误状态标识，函数调用者基于值比较的方式，对这一代表错误状态的返回值进行检视。通常，当这个返回值为 0 时，代表函数调用成功；当这个返回值为其他值时，代表函数调用出现错误。函数调用者需根据该返回值所代表的错误状态，来决定后续执行哪条错误处理路径上的代码。

C 语言这种简单的**基于错误值比较**的错误处理机制，让每个开发人员必须显式地去关注和处理每个错误。经过显式错误处理的代码会更为健壮，也会让开发人员对这些代码更有信心。另外，这些错误就是普通的值，我们不需要额外的语言机制去处理它们，只需利用已有的语言机制，像处理其他普通类型值那样去处理错误就可以了。这让代码更容易调试，我们也更容易针对每个错误处理的决策分支进行测试覆盖。

C 语言错误处理机制的这种简单与显式，跟 Go 语言的设计哲学十分契合，于是 Go 语言设计者决定继承这种错误处理机制。因此，当 Gopher 们来到 C 语言的世界时，无需对自己的错误处理思维做出很大的改变，就可以很容易地适应 C 语言的风格。

知己知彼，来看看 C 与 Go 的差异

虽说 Gopher 学习 C 语言有“先天优势”，但是不经过脚踏实地的学习与实践就想掌握和精通 C 语言，也是不可能的。而且，C 和 Go 还是有很大差异的，Gopher 们只有清楚这些差异，做到“知己知彼”，才能在学习过程中分清轻重，有的放矢。俗话说，“磨刀不误砍柴功”，下面我们就一起看看 C 与 Go 有哪些不同。

设计哲学



在人类自然语言学界，有一个很著名的假说——“[🔗 萨丕尔 - 沃夫假说](#)”。这个假说的内容是这样的：**语言影响或决定人类的思维方式**。对我来说，**编程语言也不仅仅是一门工具，它还影响着程序员的思维方式**。每次开始学习一门新的编程语言时，我都会先了解这门编程语言的设计哲学。

每种编程语言都有自己的设计哲学，即便这门语言的设计者没有将其显式地总结出来，它也真真切切地存在，并影响着这门语言的后续演进，以及这门语言程序员的思维方式。我在 [🔗 《Tony Bai · Go 语言第一课》](#) 专栏里，将 Go 语言的设计哲学总结成了 5 点，分别是**简单、显式、组合、并发和面向工程**。

那么 C 语言的设计哲学又是什么呢？从表面上看，简单紧凑、性能至上、极致资源、全面移植，这些都可以作为 C 的设计哲学，但我倾向于一种更有人文气息的说法：**满足和相信程序员**。

在这样的设计哲学下，一方面，C 语言提供了几乎所有可以帮助程序员表达自己意图的语法手段，比如宏、指针与指针运算、位操作、`pragma` 指示符、`goto` 语句，以及跳转能力更为强大的 `longjmp` 等；另一方面，C 语言对程序员的行为并没有做特别严格的限定与约束，C 程序员可以利用语言提供的这些语法手段，进行天马行空的发挥：访问硬件、利用指针访问内存中的任一字节、操控任意字节中的每个位（bit）等。总之，C 语言假定程序员知道他们在做什么，并选择相信程序员。

C 语言给了程序员足够的自由，可以说，在 C 语言世界，你几乎可以“为所欲为”。但这种哲学也是有代价的，那就是你可能会犯一些莫名其妙的错误，比如悬挂指针，而这些错误很少或不可能在其他语言中出现。

这里再用一个比喻来更为形象地表达下：从 Go 世界到 C 世界，就好比在动物园中饲养已久的动物被放归到野生自然保护区，有了更多自由，但周围也暗藏着很多未曾遇到过的危险。因此，学习 C 语言的 Gopher 们要有足够的心理准备。

内存管理

接下来我们来看 C 与 Go 在内存管理方面的不同。我把这一点放在第二位，是因为这两种语言在内存管理上有很大的差异，而且这一差异会给程序员的日常编码带来巨大影响。

领资料



我们知道，Go 是带有垃圾回收机制（俗称 GC）的静态编程语言。使用 Go 编程时，内存申请与释放，在栈上还是在堆上分配，以及新内存块的清零等等，这一切都是自动的，且对程序员透明。

但在 C 语言中，上面说的这些都是程序员的责任。手工内存管理在带来灵活性的同时，也带来了极大的风险，其中最常见的就是内存泄露（memory leak）与悬挂指针（dangling pointer）问题。

内存泄露主要指的是**程序员手工在堆上分配的内存使用后被没有释放（free），进而导致的堆内存持续增加**。而悬挂指针的意思是**指针指向了非法的内存地址**，未初始化的指针、指针所指对象已经被释放等，都是导致悬挂指针的主要原因。针对悬挂指针进行解引用（dereference）操作将会导致运行时错误，从而导致程序异常退出的严重后果。

Go 语言带有 GC，而 C 语言不带 GC，这都是由各自语言设计哲学所决定的。GC 是不符合 C 语言的设计哲学的，因为一旦有了 GC，程序员就远离了机器，程序员直面机器的需求就无法得到满足了。并且，一旦有了 GC，无论是在性能上还是在资源占用上，都不可能做到极致了。

在 C 中，手工管理内存到底是一种什么感觉呢？作为一名有着十多年 C 开发经验的资深 C 程序员，我只能告诉你：**与内存斗，其乐无穷！**这是在带 GC 的编程语言中无法体会到的。

语法形式

虽然 C 语言是 Go 的先祖，并且 Go 也继承了很多 C 语言的语法元素，但在变量 / 函数声明、行尾分号、代码块是否用括号括起、标识符作用域，以及控制语句语义等方面，二者仍有较大差异。因此，对 Go 已经很熟悉的程序员在初学 C 时，受之前编码习惯的影响，往往会踩一些“坑”。基于此，我总结了 Gopher 学习 C 语言时需要特别注意的几点，接下来我们具体看看。

第一，注意声明变量时类型与变量名的顺序。

前面说过，Go 与 C 都是静态编译型语言，这就要求我们在使用任何变量之前，需要先声明这个变量。但 Go 采用的变量声明语法颇似 Pascal 语言，即**变量名在前，变量类型在后**，这与 C 语言恰好相反，如下所示：




```

2 Go:
3
4 var a, b int
5 var p, q *int
6
7 vs.
8
9 C:
10 int a, b;
    int *p, *q;

```

此外，Go 支持短变量声明，并且由于短变量声明更短小，无需显式提供变量类型，Go 编译器会根据赋值操作符后面的初始化表达式的结果，自动为变量赋予适当类型。因此，它成为了 Gopher 们喜爱和重度使用的语法。但短声明在 C 中却不是合法的语法元素：

 复制代码

```

1 int main() {
2     a := 5; // error: expected expression
3     printf("a = %d\n", a);
4 }

```

不过，和上面的变量类型与变量名声明的顺序问题一样，C 编译器会发现并告知我们这个问题，并不会给程序带来实质性的伤害。

第二，注意函数声明无需关键字前缀。

无论是 C 语言还是 Go 语言，函数都是基本功能逻辑单元，我们也可以说 **C 程序就是一组函数的集合**。实际上，我们日常的 C 代码编写大多集中在实现某个函数上。

和变量一样，函数在两种语言中都需要先声明才能使用。Go 语言使用 `func` 关键字作为**函数声明的前缀**，并且函数返回值列表放在函数声明的最后。但在 C 语言中，函数声明无需任何关键字作为前缀，函数只支持单一返回值，并且返回值类型放在函数名的前面，如下所示：

 领资料

 复制代码

```

1 Go:
2 func Add(a, b int) int {
3     return a+b
4 }
5
6 vs.

```

```
7 C:
8 int Add(int a, int b) {
9     return a+b;
10 }
11
```

第三，记得加上代码行结尾的分号。

我们日常编写 Go 代码时，**极少手写分号**。这是因为，Go 设计者当初为了简化代码编写，提高代码可读性，选择了**由编译器在词法分析阶段自动在适当位置插入分号的技术路线**。如果你是一个被 Go 编译器惯坏了的 Gopher，来到 C 语言的世界后，一定不要忘记代码行尾的分号。比如上面例子中的 C 语言 Add 函数实现，在 return 语句后面记得要手动加上分号。

第四，补上“省略”的括号。

同样是出于简化代码、增加可读性的考虑，Go 设计者最初就取消掉了条件分支语句（if）、选择分支语句（switch）和循环控制语句（for）中条件表达式外围的小括号：

 复制代码

```
1 // Go代码
2 func f() int {
3     return 5
4 }
5 func main() {
6     a := 1
7     if a == 1 { // 无需小括号包裹条件表达式
8         fmt.Println(a)
9     }
10
11     switch b := f(); b { // 无需小括号包裹条件表达式
12     case 4:
13         fmt.Println("b = 4")
14     case 5:
15         fmt.Println("b = 5")
16     default:
17         fmt.Println("b = n/a")
18     }
19
20     for i := 1; i < 10; i++ { // 无需小括号包裹循环语句的循环表达式
21         a += i
22     }
23     fmt.Println(a)
24 }
```

领资料

这一点恰恰与 C 语言“背道而驰”。因此，我们在使用 C 语言编写代码时，务必要想着补上这些括号：

 复制代码

```
1 // C代码
2 int f() {
3     return 5;
4 }
5
6 int main() {
7     int a = 1;
8     if (a == 1) { // 需用小括号包裹条件表达式
9         printf("%d\n", a);
10    }
11
12    int b = f();
13    switch (b) { // 需用小括号包裹条件表达式
14    case 4:
15        printf("b = 4\n");
16        break;
17    case 5:
18        printf("b = 5\n");
19        break;
20    default:
21        printf("b = n/a\n");
22    }
23
24    int i = 0;
25    for (i = 1; i < 10; i++) { // 需用小括号包裹循环语句的循环表达式
26        a += i;
27    }
28    printf("%d\n", a);
29 }
```

第五，留意 C 与 Go 导出符号的不同机制。

C 语言通过头文件来声明对外可见的符号，所以我们不用管符号是不是首字母大写的。但在 Go 中，只有首字母大写的包级变量、常量、类型、函数、方法才是可导出的，即对外部包可见。反之，首字母小写的则为包私有的，仅在包内使用。Gopher 一旦习惯了这样的规则，在切换到 C 语言时，就会产生“心理后遗症”：遇到在其他头文件中定义的首字母小写的函数时，总以为不能直接使用。

第六，记得在 switch case 语句中添加 break。

 领资料



C 语言与 Go 语言在选择分支语句的语义方面有所不同：C 语言的 `case` 语句中，如果没有显式加入 `break` 语句，那么代码将向下自动掉落执行。而 Go 在最初设计时就重新规定了 `switch case` 的语义，默认不自动掉落（`fallthrough`），除非开发者显式使用 `fallthrough` 关键字。

适应了 Go 的 `switch case` 语句的语义后再回来写 C 代码，就会存在潜在的“风险”。我们来看一个例子：

 复制代码

```
1 // C代码：
2 int main() {
3     int a = 1;
4     switch(a) {
5         case 1:printf("a = 1\n");
6         case 2:printf("a = 2\n");
7         case 3:printf("a = 3\n");
8         default:printf("a = ?\n");
9     }
10 }
```

这段代码是按 Go 语义编写的 `switch case`，编译运行后得到的结果如下：

 复制代码

```
1 a = 1
2 a = 2
3 a = 3
4 a = ?
```

这显然不符合我们输出“`a = 1`”的预期。对于初学 C 的 Gopher 而言，这个问题影响还是蛮大的，因为这样编写的代码在 C 编译器眼中是完全合法的，但所代表的语义却完全不是开发人员想要的。这样的程序一旦流入到生产环境，其缺陷可能会引发生产故障。

一些 C lint 工具可以检测出这样的问题，因此对于写 C 代码的 Gopher，我建议提交代码前使用 lint 工具对代码做一下检查。

 领资料

构建机制

Go 与 C 都是静态编译型语言，它们的源码需要经过编译器和链接器处理，这个过程称为**构建 (build)**，构建后得到的可执行文件才是最终交付给用户的成果物。

和 Go 语言略有不同的是，C 语言的构建还有一个预处理（pre-processing）阶段，预处理环节的输出才是 C 编译器的真正输入。C 语言中的宏就是在预处理阶段展开的。不过，Go 没有预处理阶段。

C 语言的编译单元是一个 C 源文件（.c），每个编译单元在编译过程中会对应生成一个目标文件（.o/.obj），最后链接器将这些目标文件链接在一起，形成可执行文件。

而 Go 则是以一个包（package）为编译单元的，每个包内的源文件生成一个.o 文件，一个包的所有.o 文件聚合（archive）成一个.a 文件，链接器将这些目标文件链接在一起形成可执行文件。

Go 语言提供了统一的 Go 命令行工具链，且 Go 编译器原生支持增量构建，源码构建过程不需要 Gopher 手工做什么配置。但在 C 语言的世界中，用于构建 C 程序的工具有很多，主流的包括 gcc/clang，以及微软平台的 C 编译器。这些编译器原生不支持增量构建，为了提升工程级构建的效率，避免每次都进行全量构建，我们通常会使用第三方的构建管理工具，比如 make（Makefile）或 CMake。考虑移植性时，我们还会使用到 configure 文件，用于在目标机器上收集和设置编译器所需的环境信息。

依赖管理

我在前面提过，C 语言仅提供了一个“小内核”。像依赖管理这类的事情，C 语言本身并没有提供跟 Go 中的 Go Module 类似的，统一且相对完善的解决方案。在 C 语言的世界中，我们依然要靠外部工具（比如 CMake）来管理第三方的依赖。

C 语言的第三方依赖通常以静态库（.a）或动态共享库（.so）的形式存在。如果你的应用要使用静态链接，那就必须在系统中为 C 编译器提供第三方依赖的静态库文件。但在实际工作中，完全采用静态链接有时是会遇到麻烦的。这是因为，很多操作系统在默认安装时是不带开发包的，也就是说，像 libc、libpthread 这样的系统库只提供了动态共享库版本（如 /lib 下提供了 libc 的共享库 libc.so.6），其静态库版本是需要自行下载、编译和安装的（如 libc 的静态库 libc.a 在安装后是放在 /usr/lib 下面的）。所以**多数情况下，我们是将静态、动态两种链接方式混合在一起使用的**，比如像 libc 这样的系统库多采用动态链接。

动态共享库通常是有版本的，并且按照一定规则安装到系统中。举个例子，一个名为 libfoo 的动态共享库，在安装的目录下文件集合通常是这样：

领资料




```
1 2022-03-10 12:28 libfoo.so -> libfoo.so.0.0.0*
2 2022-03-10 12:28 libfoo.so.0 -> libfoo.so.0.0.0*
3 2022-03-10 12:28 libfoo.so.0.0.0*
```

按惯例，每个动态共享库都有多个名字属性，包括 **real name**、**soname** 和 **linker name**。下面我们分别看下。

- **real name**: 实际包含共享库代码的那个文件的名称 (如上面例子中的 `libfoo.so.0.0.0`)。动态共享库的真实版本信息就在 **real name** 中，显然 **real name** 中的版本号符合 [语义版本规范](#)，即 `major.minor.patch`。当两个版本的 `major` 号一致，说明是向后兼容的两个版本；
- **soname**: **shared object name** 的缩写，也是这三个名字中最重要的一个。无论是在编译阶段还是在运行阶段，系统链接器都是通过动态共享库的 **soname** (如上面例子中的 `libfoo.so.0`) 来唯一识别共享库的。我们看到的 **soname** 实际上是仅包含 `major` 号的共享库名字；
- **linker name**: 编译阶段提供给编译器的名称 (如上面例子中的 `libfoo.so`)。如果你构建的共享库的 **real name** 跟上面例子中 `libfoo.so.0.0.0` 类似，带有版本号，那么你在编译器命令中直接使用 `-L path -lfoo` 是无法让链接器找到对应的共享库文件的，除非你为 `libfoo.so.0.0.0` 提供了一个 **linker name** (如 `libfoo.so`，一个指向 `libfoo.so.0.0.0` 的符号链接)。linker name 一般在共享库安装时手工创建。

动态共享库有了这三个名称属性，依赖管理就有了依据。但由于在链接的时候使用的是 **linker name**，而 **linker name** 并不带有版本号，真实版本与主机环境有关，因此要实现 C 应用的可重现构建还是比较难。在实践中，我们通常会使用专门的构建主机，项目组将该主机上的依赖管理起来，进而保证每次构建所使用的依赖版本是可控的。同时，应用部署的目标主机上的依赖版本也应该得到管理，避免运行时出现动态共享库版本不匹配的问题。

代码风格

Go 语言是历史上首次实现了代码风格全社区统一的编程语言。它基本上消除了开发人员在代码风格上的无休止的、始终无法达成一致的争论，以及不同代码风格带来的阅读、维护他人代码时的低效。`gofmt` 工具格式化出来的代码风格已经成为 Go 开发者的一种共识，融入到 Go 语言的开发文化当中了。所以，如果你让某个 Go 开发者说说 `gofmt` 后的代码风格是什么样的，多数 Go 开发者可能说不出，因为代码会被 `gofmt` 自动变成那种风格，大家已经不再关心风格了。



而在 C 语言的世界，代码风格仍存争议。但经过多年的演进，以及像 Go 这样新兴语言的不断“教育”，C 社区也在尝试进行这方面的改进，涌现出了像 [🔗 clang-format](#) 这样的工具。目前，虽然还没有在全社区达成一致的代码风格（由于历史原因，这很难做到），但已经可以减少很多不必要的争论。

对于正在学习 C 语言，并进行 C 编码实践的 Gopher，我的建议是：**不要拘泥于使用什么代码风格，先用 clang-format，并确定一套风格模板就好。**

小结

作为一名对 Go 跟随和研究了近十年的程序员，我深刻体会到，Go 的简单性、性能和生产力使它成为了创建面向用户的应用程序和服务的理想语言。快速的迭代让团队能够快速地作出反应，以满足用户不断变化的需求，让团队可以将更多精力集中在保持灵活性上。

但 Go 也有缺点，比如缺少对内存以及一些低级操作的精确控制，而 C 语言恰好可以弥补这个缺陷。C 语言提供的更精细的控制允许更多的精确性，使得 C 成为低级操作的理想语言。这些低级操作不太可能发生变化，并且 C 相比 Go 还提高了性能。所以，如果你是一个有性能与低级操作需求的 Gopher，就有充分的理由来学习 C 语言。

C 的优势体现在最接近底层机器的地方，而 Go 的优势在离用户较近的地方能得到最大发挥。当然，这并不是说两者都不能在对方的空间里工作，但这样做会增加“摩擦”。当你的需求从追求灵活性转变为注重效率时，用 C 重写库或服务的理由就更充分了。

总之，虽然 Go 和 C 的设计有很大的不同，但它们也有很多相似性，具备发挥兼容优势的基础。并且，当我们同时使用这二者时，就可以既有很大的灵活性，又有很好的性能，可以说是相得益彰！

写在最后


今天的加餐中，我主要是基于 C 与 Go 的比较来讲解的，对于 Go 语言的特性并没有作详细展开。如果你还想进一步了解 Go 语言的设计哲学、语法特性、程序设计相关知识，欢迎来学习我在极客时间上的专栏 [🔗 《Tony Bai · Go 语言第一课》](#)。在这门课里，我会用我十年 Gopher 的经验，带给你一条系统、完整的 Go 语言入门路径。

感谢你看到这里，如果今天的内容让你有所收获，欢迎把它分享给你的朋友。



分享给需要的人，Ta订阅超级会员，你最高得 50 元

Ta单独购买本课程，你将得 20 元

 生成海报并分享

 赞 1  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 大咖助阵 | 海纳：C 语言是如何编译执行的？（三）

下一篇 期末考试 | 来赴一场满分之约吧！

更多课程推荐

操作系统实战 45 讲

从 0 到 1, 实现自己的操作系统

彭东
网名 LMOS
Intel 傲腾项目关键开发者



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

领资料

精选留言

 写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。

