Go编程语言评估报告

1. Go简介

Go是由Google于2007年9月21日开始开发，2009年11月10日开放源码，2012年3月28日推出第一个正式版本的通用型编程语言。它为系统编程而7设计，是强类型化的语言，具有垃圾回收机制，并显式支持并发编程。Go程序由包构造，以此来提供高效的依赖管理功能。当前的编译器实现使用传统的“编译-链接”模型来生成可执行的二进制文件。

十年以来，主流的系统级编程语言并未出现过，但在这期间，计算环境已经发生了巨大的变化。以下是一些变化趋势：

- 相计算机的速度变得极快，但软件开发还不够快。
- 在今天，依赖管理已然成为了软件开发中当重要的部分，但传统的C家族语言以“头文件”的方式组织源码，这与清晰的依赖分析以及快速编译背道而驰。
- Java和C++等语言的类型系统比较笨重，人们的反抗越来越多，因此他们转向了Python和JavaScript之类的动态类型语言。
- 目前流行的系统语言对于像垃圾回收及并行计算等基本思想的支持并不算好。
- 多核计算机的出现产生了一些麻烦与混乱。
- 随着系统的庞大，分布式的要求越来越高，现有的语言开发分布式系统越来越笨重以及难以维护。

而Go则是一种新的语言，一种并发的、带垃圾回收的、快速编译的语言。它满足了以下特点：

- 它可以在一台计算机上用几秒钟的时间编译一个大型的Go程序。
- Go为软件构造提供了一种模型，它使依赖分析更加容易，且避免了大部分C风格include文件与库的开头。
- Go是静态类型的语言，它的类型系统没有层级。因此用户不需要在定义类型间的关系上花费时间，这样感觉起来比典型的面向对象语言更轻量级。
- Go完全是垃圾回收型的语言，并为并发执行与通信提供了基本的支持。
- 按照其设计，Go打算为多核机器上系统软件的构造提供一种方法。

Go试图成为结合解释型编程的轻松、动态类型语言的高效以及静态类型语言的安全的编译型语言。它也打算成为现代的，支持网络与多核计算的语言。要满足这些目标，需要解决一些语言上的问题：一个富有表达能力但轻量级的类型系统，并发与垃圾回收机制，严格的依赖规范等等。这些无法通过库或工具解决好，因此Go也就应运而生了。

2. C/C++的缺陷

a. 全局变量的初始化顺序

由于在C/C++中，全局变量的初始化顺序并不确定，因此依赖于全局变量初始化顺序的操作，可能会给程序带来不可预知的问题。

b. 变量默认不会被初始化

由于变量默认不会被初始化，因此如果在程序中忘记初始化某个变量，就有可能造成一些奇怪的细节性错误，以至于在Coding Standard中都为之专门加以强调。

c. 字符集的支持[1]

C/C++最先支持的字符集是ANSI。虽然在C99/C++98之后提供了对Unicode的支持，但在日常的编码工作中却要在ANSI与Unicode之间来回转换，相当地繁琐。

d. 复杂的继承模式

C++提供了单/多继承，而多继承则引入了大量的复杂性，比如“钻石型继承”等。细节请参阅《深度探索C++对象模型》。

e. 对并发的支持

在C++中，并发更多的是通过创建线程来启用，而线程间的通信则是通过加锁共享变量来实现，很容易死锁。虽然在C++11中添加了并发处理机制，但这给本来就十分复杂的类型系统又添加了更重的负担。

f. 不支持自动垃圾回收

关于这一点存在争议。内存泄漏是C/C++程序员经常遭遇到的问题，但随着垃圾回收算法的成熟，对于大多数开发者来说，自动回收带来的便利已经超过手工操作提高的效率。而在C++中虽然可使用智能指针来减少原生指针的使用，但不能杜绝它，因此这个问题仍然存在。

g. 落后的包管理机制

C/C++中采用.h/.c(pp)文件来组织代码，这种方式使编译时间变得过于漫长；C++中的模板更让这个问题雪上加霜。

h. C++编译器总会私自生成一些代码

比如: 构造函数/析构函数/new/delete等。如果是动态库，当include不同版本的头文件时，容易生成版本不兼容的代码。

i. 不加约束的指针使用是导致C/C++软件BUG的重要根源之一

3. Go的优势

---

[1] C/C++编译时的编码方式也不确定。main.cpp如果是UTF-8编码, 在gcc和VC下的行为也会不一样。——柴树杉

正如语言的设计者之一Rob Pike所说:

“*我们——Ken，Robert和我自己曾经是C++程序员，我们设计新的语言是为了解决那些我们在编写软件时遇到的问题。*”

这些问题中的大部分，就是在第2节中列举的内容。这一小节就是Go针对这些缺陷提出的解决方案。

a. Init

每个包都可以定义一个或多个init函数[2]（原型为 func init()），init函数在包初次被导入时调用，同一个包内的多个init函数的执行的顺序是不定的，而如果这个包又导入了其他的包，则级连调用，所有包import完成，所有init函数执行完后，则开始main的执行。

而对于全局变量，以一个简单的例子来说明：

```
// package p
var gInt int
…

// package a
import "p"
…

// package b
import "p"
…

// package main
import (
       "a"
       "b"
)
…
```

在package p中，我们定义了一个全局变量gInt，而p被package a,b所import，接着package main又按序import了a,b，即a在b前被import。a先import了p,所以此时gInt被初始化，这样就解决了C/C++中全局变量初始化顺序不一致的问题。

b. 默认自动初始化

Go引入了零值的概念，即每个对象被创建的时候，默认初始化为它相应类型的零值。例如，string为””，指针为nil，int为0等等，这样就保证了变量在使用时，不会因为忘记初始化而出现一些莫名其妙的问题。此外，由于零值的引入，也方便了代码的编写。比如说sync包的mutex类型，在引入零值后，就能以如下方式使用：

---

[2] 每个源文件也可包含多个init函数。多个init之间的调用顺序不确定。init函数本身不能被其它变量或函数引用（调用或取函数地址）。——柴树杉

```
var locker sync.Mutex
locker.Lock()
defer locker.Unlock()
…
```

而相应的C/C++代码，可能就要这样写了：

```
CRITICAL_SECTION locker
InitializeCriticalSection(&locker)
EnterCriticalSection(&locker)
…
LeaveCriticalSection(&locker)
DeleteCriticalSection(&locker)
```

忘记任何一步操作，都将造成死锁(dead lock)或者其他的问题。

c. UTF-8

Go语言原生支持UTF-8编码格式[3]。同时Go涉及到字符串的各种包，也直接为UTF-8提供了支持,比如：

```
str := "示例"
if str == "示例" {...}
```

d. 只支持组合不支持继承

OOP在Go中是通过组合而非继承来实现的，因为“继承”存在一些弊端，比如：“不适应变化”，“会继承到不适用的功能”。所以在编码实践中一般建议优先使用组合而非继承。在Go中则更进一步，直接去掉了继承，只支持组合。在定义struct时，采用匿名组合的方式，也更好地实现了C++中的“实现”继承，而在定义interface时，也可以实现接口继承。比如：

```
type A struct{}
func (a A) HelloA() {
      …
}

type B struct{}
func (b B) HelloB() {
      …
}

type C struct {
      A
B
}

c := &C{}
c.HelloA()
c.HelloB()
```

此时c就拥有了HelloA、HelloB两个方法，即我们很容易地实现了“实现继承”。

---

[3] 同时已经支持带BOM的UTF-8了。——柴树杉

e. Go程(goroutine)与信道(channel)

Go对并发的支持，采用的是CSP模型，即在代码编写的时候遵循“通过通信来共享内存，而非通过共享内存来通信”的原则。为此，Go提供了一种名为“Go程”的抽象。由于Go程是一种高于线程的抽象，因此它使用起来也就更加轻量方便。而当多个Go程需要通信的时候，信道就成为了它们之间的桥梁。例如：

```
func goroutine(pass chan bool) {
	fmt.Println("hello, i'm in the goroutine")
	pass <- true
}

func main() {
	pass := make(chan bool)
	go goroutine(pass)
	<-pass
	fmt.Println("passed")
}
```

代码中通过关键字chan来声明一个信道，在函数前加上关键字go来开启一个新的Go程。此Go程在执行完成后，会自动销毁。而在通信过程中，可通过<-操作符向信道中放入或从中取出数据。

f. 自动垃圾回收

与C#、Java等语言类似，为了将程序员从内存泄漏的泥沼中解救出来，Go提供了自动垃圾回收机制，同时不再区分对象是来自于栈(stack)还是堆(heap)。

g. 接口(interface)

除Go程外，Go语言的最大特色就是接口的设计，Go的接口与Java的接口，C++的虚基类是不同的，它是非侵入式的，即我们在定义一个struct的时候，不需要显式的说明它实现了哪一/几个interface，而只要某个struct定义了某个interface所声明的所有方法，则它就隐式的实现了那个interface，即所谓的Structural-Typing（关于Duck-Typing与Structural-Typing的区别，请参考minux.ma的相关注释）。

假设我要定义一个叫Shape的interface，它有Circle、Square、Triangle等实现类。

在java等语言中，我们是先在大脑中从多个实现中抽象出一个interface，即:

在定义Shape的时候，我们会先从实现类中得出共性。比如它们都可以计算面积，都可以被绘制出来，即Shape拥有Area与Show方法。在定义出了Shape过后，再定义Circle、Square、Triangle等实现类，这些类都显式的从Shape派生，即我们先实现了接口再实现了“实现”。在实现“实现”的过程中，如果发现定义的接口不合适，因为“实现”显式地指定了它派生自哪个基类，所以此时我们需要重构：

```
public interface Shape {
       public float Area();
       public void Show();
}
public class Circle : implements Shape {
       public float Area() { return …}
       public void Show() {…}
}
// (同理Square和Triangle)
```

而在Go中，由于interface是隐式的，非侵入式的，我们就可以先实现Circle、Square、Triangle等子类。在实现这些“实现类”的过程中，由于知识的增加，我们可以更好地了解哪些方法应该放到interface中，即在抽象的过程中完成了重构。

```
type Circle struct {}
func (c Circle) Area() float32 {}
func (c Circle) Show() {}
// (同理Square和Triangle)
type Shape interface {
       Area() float32
       Show()
}
```

这样Circle、Square和Triangle就实现了Shape。

对于一个模块来说，只有模块的使用者才能最清楚地知道，它需要使用由其它被使用模块提供的哪些方法，即interface应该由使用者定义。而被使用者在实现时，并不知道它会被哪些模块使用，所以它只需要实现自己就好了，不需要去关心接口的粒度是多细才合适这一类的琐碎问题。interface是由使用方按需定义，而不用事前规划。

Go的interface与Java等的interface相比优势在于：

1. 按需定义，最小化重构的代价。
2. 先实现后抽象，搭配结构嵌入，在编写大型软件的时候，我们的模块可以组织得耦合度更低。

h. Go命令

在Unix/Linux下为了编译程序的方便，都可能需要编写makefile或者各种高级的自动构建工具（Windows也存在类似的工具，只不过被各种强大的IDE给隐藏在背后了）。而Rob Pike等人当初发明Go的动机之一就是：“Google的大型的C++程序的编译时间过长”。所以为了达到：“编译Go程序时，作为程序员除开编写代码外，不需要编写任何配置文件或类似额外的东西。”这个目标，引入了Go命令族。通过Go命令族，你可以很容易从实现的在线repository上获得开源代码，编译并执行代码，测试代码等功能。这与C/C++的处理方式相比，前进了一大步。

i. 自动类型推导

Go虽然是一门编译型语言，但是在编写代码的时候，却可以给你提供动态语言的灵活性。在定义一个变量的时候，你可以省略类型，而让编译器自动为之推导类型，这样减少了程序员的输入字数。比如:

```
i := 0 ⇔ var i int
s := "hello world" ⇔ var s string = "hello world"
```

j. 强制编码风格规范

在C/C++中，大家为大括号的位置采用K&R还是ANSI，是使用tab还是whitespace，whitespace是2个字符还是4个字符等琐碎的问题而争论不休。每个公司内部都定义了自己的Coding Standard来强制约束。而随着互联网的蓬勃发展，开源项目的越发增多，这些小问题却影响了大家的工作效率。而有一条编程准则是“less is more”。为了一致性，Go提供了专门的格式化命令go fmt，用以统一大家的编码风格。

作为程序员，你在编写代码的时候，可以按你喜欢的风格编写。编写完成后，执行一下go fmt命令，就可以将你的代码统一成Go的标准风格。这样你在接触到陌生的Go代码时，减少了因为编码风格差异带来的陌生感，强调了一致性。

k. 自带单元测试及性能测试工具

C/C++虽未提供官方的单元测试与性能测试工具，但有大量第三方的相关工具。而由于每个人接触的，喜欢的工具可能不一样，就造成了在交流时的负担。有鉴于此，Go提供了官方测试工具go test，你可以很方便地编写出单元测试用例。比如这样就完成了一个单元测试的编写：

```
package test

// example.go
func Add(a, b int) int {
      return a + b
}
…
// example_test.go
func TestAdd(t *testing.T) {
      //定义一个表格，以展示 table-driven 测试
      table := []struct {
            a, b, result int
      }{
            {1, 0, 1},
            {1, 2, 3},
            {-1, -2, 0},
      }

      for _, row := range table {
            if row.result != Add(row.a, row.b) {
                  t.Fatalf("failed")
            }
      }
}
```

同理性能测试。
编写完成后执行go test就可完成测试。

l. 云平台的支持

最近几年云计算发展得如火如荼，Go被称为“21世纪的C语言”，当然它也不能忽视这一块的需求。现在有大量的云计算平台支持Go语言开发，比如由官方维护的GAE，第三方的AWS等。

m. 简化的指针

这一条，可能不算优势，在C/C++中指针运算的功能非常强大，但是带来的危害也很突出，所以在Go中指针取消了运算功能，只保留了“引用/解引用”功能 。

n. 简单的语法，入门快速，对于新成员很容易上手

Go本质上是一个C家族的语言，所以如果有C家族语言的经验，很容易上手。

4. Go的劣势[4]

a. 调度器的不完善

b. 原生库太少/弱

c. 32bit上的-内存泄漏

关于这一点，Go的贡献者minux.ma在Golang-China讨论组上有详细解释:

“目前Go使用的GC是个保守的GC，换句通俗的话说就是宁可少释放垃圾，也不可误释放还在用的内存；这一点反映在设计上就是会从堆栈、全局变量开始，把所有可能是指针的uintptr全部当作指针，遍历，找到所有还能访问到的内存中的对象，然后把剩下的释放。

那么如何判断一个uintptr可能是指针呢？大家知道Go的内存分配是参考的tcmalloc，并做了一些改动。原先tcmalloc是使用类似页表的树形结构保存已经从操作系统中获得的内存页面，Go使用了另外一个办法。由于Go需要维护每个内存字的一些状态（比如是否包含指针？是否有finalizer？是否是结构体的开始？还有上面提到的是否还能访问到的状态），综合在一起是每个字需要4bit信息；于是Go就先找一片区域（arena），以不可访问的权限从操作系统那里申请过来（mmap

的prot参数是PROT_NONE），然后根据每一个uintptr对应4位申请一片RW的内存（bitmap）与前面的arena对应；这样已知heap上内存的地址想获得对应的bitmap地址就很简单了，不需要像tcmalloc似的查找，直接简单的右移和加法就能获得；同时呢，操

[4] 可以归纳为: 性能/生态/Bug/工具等。——柴树杉

作系统的demand paging会自动处理还没有使用到的bitmap。

这里大家就明白了为啥Go用了那么大的虚拟内存（arena）并且知道为啥经常在内存不足的时候panic说申请到的内存不在范围了（因为内存不在bitmap所能映射的范围里，当然多个bitmap是可以解决这个问题的，不过目前还不支持）；回到开始的那个问题，既然arena有个地址范围，判断一个uintptr是否可能是指针就是判断是否在这个范围里了。

这样的问题就来了。如果我有一个int32，他的内容恰巧在那个范围里，更碰巧的是如果把它当作指针，它恰巧指向一个大的数据结构，那么GC只能认为那个数据结构还在使用中。这样就造成了泄露。这个问题在32位/64位平台上都是存在的。但是在32位上问题更严重些，主要是32位表示的地址空间有768MB是Arena，也就是说一个均匀分布的uintptr是指针的概率是768/4096，这个远比64位系统的16GiB/(2^64B)的概率大得多。

Go 1.1不出意外的话会使用记录每个heap上分配的对象的类型的方式来几乎完整地解决这个问题；说几乎完整是因为，堆栈上的数据还是没有类型的，所以这里面还是有前面说的问题的，只不过会影响会小很多了。”

d. 无强大IDE支持

e. 最大可用内存16G限制

因为今年3月28日Go才推出Go 1，所以目前Go还存在不足。a、c、e这几个缺陷在2013年初的Go 1.1中会得到解决，而b、d则需要等时间的积累才能完善。

5. Go的争议

a. 错误处理机制

在错误处理上，Go不像C++、Java等提供了异常机制，而是采取检查返回值的方案，这是目前Go最大争议所在。

反对者的理由：

1. 每一步，都得做检查繁琐，原始。
2. 返回的error类型可以通过 _ 给忽略掉。
3. 返回的官方error接口只有一个Error()方法来输出字符串，无法用来判断复杂的错误，比我定义一个方法OpenJsonFile(name string) (jFile JFile, err Error), 这个方法可能引法的错误有两种1.文件没找到，2,文件解析错误，这时我希望返回的错误中带有两个信息，1，错误码，2错误的提示， 错误码，用于程序中的判断，错误提示用于快速了解这个错误，现在官方的error接口只有错误提示，而没有错误码。

支持的理由：

1. 在离错误发生最近的地方，可以用最佳的方式处理错误。
2. 异常在crash后抛出的stack信息，对于别有用心者，会泄漏关键信息；而对于最终用户，他将看不明白究竟发生了什么情况。而使用错误机制能让你有机会将stack信息替换为更有意义的信息，这样就能提高安全性和用户友好性。

3. 异常也可以默认处理。

b. new与变量初始化

在Go中，new与delete和在C++中的含义是不一样的。delete用以删除一个map项，而new用以获得一个指向某种类型对象的指针，而因为Go支持类似如下的语法：

```
type T struct {
        …
}
obj := &T{} ⇔ obj = new(T)
```

同时Go提供另一个关键字make用以创建内建的对象，所以&T{}这种语法与make合起来，就基本可以替代new（但目前new(int)这类基本类型指针的创建，则无法用&T{}的写法），因此new看起来有点冗余了，这与Go的简单原则有点不一致。

c. For…range不能用于自定义类型

为了遍历的方便，Go提供了for-range语法，但是这种构造只能用于built-in类型，如slice、map和chan；而对于非built-in类型，即使官方包container中的相关数据结构也不行，这降低了for-range的易用性。而目前在不支持泛型的前提下，要实现一个很友好的for-range看起来还是很不容易的。

d. 不支持动态链接

目前Go只支持静态链接（但gccgo支持动态链接，Go 1.1可能会支持部分动态链接），这又是另一个引起争论的地方。争论双方的论据就是动态链接/静态链接的优、缺点，在此不再赘述。

e. 无泛型

现代的大多数编程语言都提供了对泛型的支持，而在Go 1中则没有提供对泛型的支持。按官方团队成员Russ Cox的说法，支持泛型要么降低编译效率，要么降低程序员效率，要么降低运行效率。而这三个恰好与Go的快速、高效、易编写的目标是相冲突的。同时Go提供的interface{}可以降低对泛型的期望和需求，因此是否需要泛型也成了争论的焦点。

f. 首字母大写表示可见性

Go中只支持包级别的可见性，即无论变量、结构、方法、还是函数等，如果以大写字母开头，则它的可见性是公共的，在其它包中可加以引用；如果以小写字母开头，则其可见性为其所在的包。由于Go支持UTF-8，而对于像中文这种没有大小写分别的字符在需要导出时，就会出现问题。关于这个问题，支持者的理由是：既然语言本身支持UTF-8，那么在变量命名上就应该是一致的；不支持者的理由是，中国人用中文命名，日本人用日语命名…而且非要用类似中文这类符号编写的话，可以在中文符号前加一个英文符号.比如:

```
var 不可导出 int = 0
var E可导出 int = 0
```

6. 替代方案

a. Cgo

在前边的劣势部分有讲过，Go缺乏原生包，而现在世面上已经有大量的C实现的高质量的第三方库，比如OpenAL、OpenCL、OpenGL等。为了解决这个问题，Go引入一个叫做cgo的命令，通过遵守简单的约定，就可以将一个C库wrapper成一个Go包，这也是为何在短短几年Go拥有了大量高质量包的原因。cgo相关示例在此不再展示。

b. B/S

因为到目前为止，Go尚未提供GUI相关的支持。同时在云计算时代，越来越多的程序采用了B/S结构，而Go对Web编程提供了最完善的支持，所以如果程序需要提供界面，无论是本地程序，还是服务器程序，在当下建议使用B/S架构来替代。

7. 参考资料及推荐

a. 本文内容多来自于以下三个Google Groups:

1. Golang-Dev（Go语言开发讨论组）
2. Golang-Nuts（Go语言使用讨论组）
3. Golang-China（Go语言使用中文讨论组）

b. 推荐书籍

1. 《学习Go语言》
   由Mikespook翻译的一本英文学习资料。
2. 《Go语言编程》
   由许式伟主笔的国内第2本关于Go语言的中文书籍。
3. 《Go Web编程》
   一本由Asta谢主笔，我参与审阅的开源中文书籍。如果在阅读过程中有任何疑问，请加入QQ群：259316004讨论。

c. 推荐框架

1. Golanger Web框架
   一个由leetaifook与borderj开发的开源Web框架。如果您在阅读/使用过程中有任何疑问，请加入QQ群：29994666讨论。

d. 推荐站点

1. Golang-Chinese
   一个主要由我维护的Google+ Page，包含一些我认为值得分享的有关Go语言的资料。如果哪位愿意共同维护，请留言告之，Thanks。
2. Go-zh
   Go语言官方网站的中文翻译版，除pkg文档外，其它文档基本翻译完成。如果哪位愿意共同翻译维护，请联系Oling Cat。

注：

BSD 3-CLAUSE LINCENSE