

# 目錄

介紹	0
第一章 语言基础	1
1.1. Go语言创世纪	1.1
1.2. Hello, World 的革命	1.2
1.3. 数组、字符串和切片	1.3
1.4. 函数、方法和接口	1.4
1.5. 面向并发的内存模型	1.5
1.6. 常见的并发模式	1.6
1.7. 错误和异常	1.7
1.8. 配置开发环境	1.8
第二章 CGO编程	2
2.1. 你好, CGO!	2.1
2.2. 基础类型转换	2.2
2.3. CGO编程基础	2.3
2.4. CGO内存模型	2.4
2.5. C++类包装	2.5
2.6. 静态库和动态库	2.6
2.7. Go实现Python模块	2.7
2.8. 补充说明	2.8
第三章 汇编语言(TODO)	3
3.1. 汇编基础(TODO)	3.1
3.2. 控制流(TODO)	3.2
3.3. 包变量(TODO)	3.3
3.4. 叶子函数(TODO)	3.4
3.5. 动态栈(TODO)	3.5
3.6. 垃圾回收(TODO)	3.6
3.7. 例子: 图像降采样(TODO)	3.7
3.8. 补充说明(TODO)	3.8
第四章 移动平台(TODO)	4
第六章 Go和Web	5

6.1. Web开发简介	5.1
6.2. Router请求路由	5.2
6.3. Middleware中间件	5.3
6.4. Validator请求校验	5.4
6.5. Database和数据库打交道	5.5
附录	6
附录A: Go语言常见坑	6.1
附录B: 参考资料	6.2
附录C: 作者简介	6.3

# Go语言高级编程(Advanced Go Programming)

本书针对Go语言有一定经验，想深入了解Go语言各种高级用法的开发人员。



- 作者：柴树杉 (chai2010, chaishushan@gmail.com)
- 网址：<https://github.com/chai2010/advanced-go-programming-book>

在线阅读

- <https://www.gitbook.com/book/chai2010/advanced-go-programming-book/>
- [SUMMARY.md](#)

## 版权声明

Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License。



严禁任何商业行为使用或引用该文档的全部或部分内容！

欢迎大家提供建议！

# 第一章 语言基础

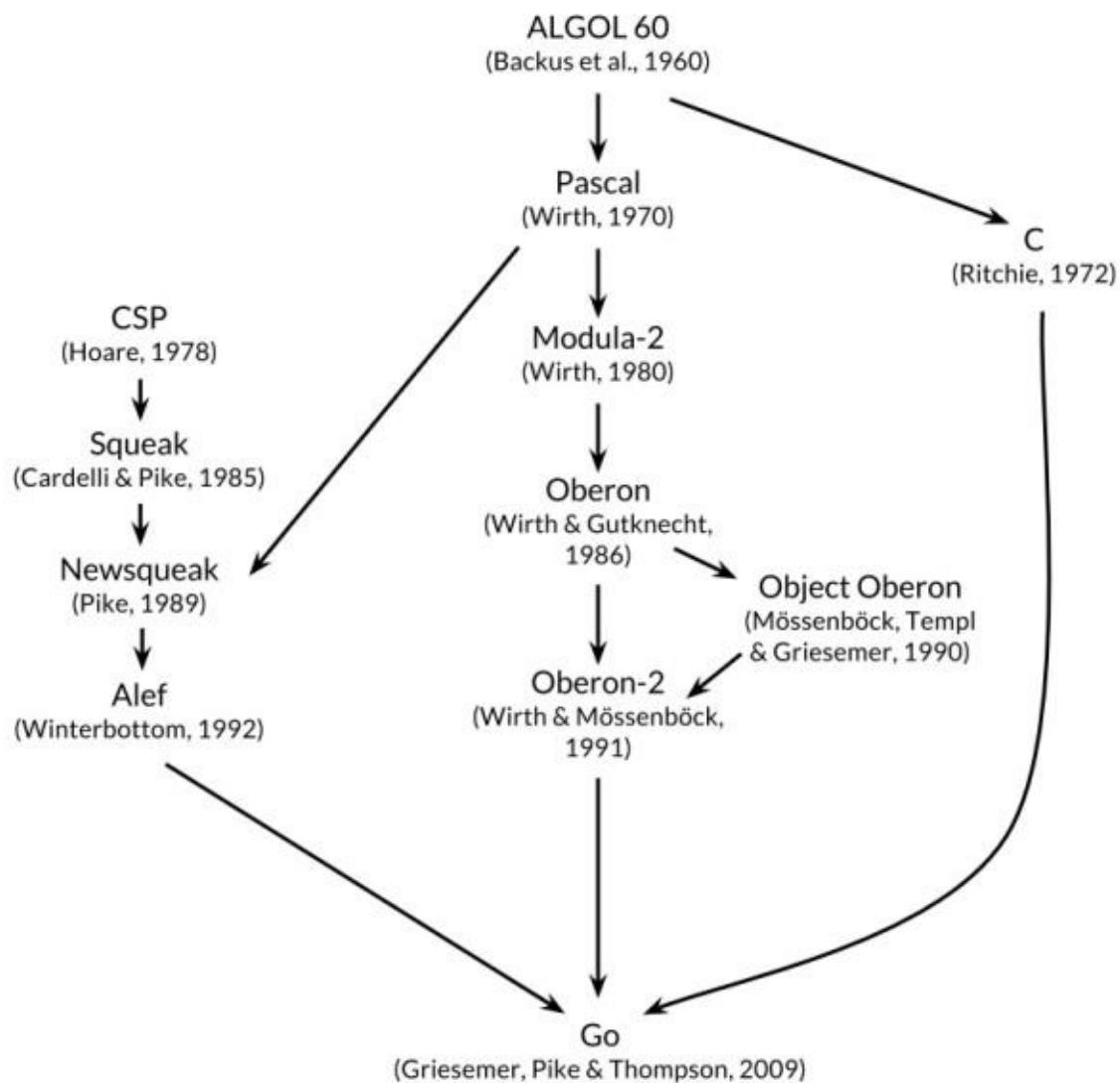
本章首先简要介绍Go语言的发展历史，并较详细地分析了“Hello World”程序在各个祖先语言中演化过程。然后，对以数组、字符串和切片为代表的基础结构，对以函数、方法和接口所体现的面向过程和鸭子对象的编程，以及Go语言特有的并发编程模型和错误处理哲学做了简单介绍。最后，针对macOS、Windows、Linux几个主流的开发平台，推荐了几个较友好的Go语言编辑器和集成开发环境，因为好的工具可以极大地提高我们的效率。

## 1.1. Go语言创世纪

Go语言最初由Google公司的Robert Griesemer、Ken Thompson和Rob Pike三个大牛于2007年开始设计发明，设计新语言的最初的洪荒之力来自于对超级复杂的C++11特性的吹捧报告的鄙视，最终的目标是设计网络和多核时代的C语言。到2008年中期，语言的大部分特性设计已经完成，并开始着手实现编译器和运行时，大约在这一年Russ Cox作为主力开发者加入。到了2010年，Go语言已经逐步趋于稳定，并在9月正式发布Go语言并开源了代码。



Go语言很多时候被描述为“类C语言”，或者是“21世纪的C语言”。从各种角度看，Go语言确实是从C语言继承了相似的表达式语法、控制流结构、基础数据类型、调用参数传值、指针等诸多编程思想，还有彻底继承和发扬了C语言简单直接的暴力编程哲学等。下面是《Go语言圣经》中给出的Go语言的基因图谱，我们可以从中看到有哪些编程语言对Go语言产生了影响。



首先看基因图谱的左边一支。可以明确看出Go语言的并发特性是由贝尔实验室的Hoare于1978年发布的CSP理论演化而来。其后，CSP并发模型在Squeak/NewSqueak和Alef等编程语言中逐步完善并走向实际应用，最终这些设计经验被消化并吸收到了Go语言中。业界比较熟悉的Erlang编程语言的并发编程模型也是CSP理论的另一种实现。

再看基因图谱的中间一支。中间一支主要包含了Go语言中面向对象和包特性的演化历程。Go语言中包和接口以及面向对象等特性则继承自Niklaus Wirth所设计的Pascal语言以及其后所衍生的相关编程语言。其中包的概念、包的导入和声明等语法主要来自于Modula-2编程语言，面向对象特性所提供的方法的声明语法等则来自于Oberon编程语言。最终Go语言演化出了自己特有的支持鸭子面向对象模型的隐式接口等诸多特性。

最后是基因图谱的右边一支，这是对C语言的致敬。Go语言是对C语言最彻底的一次扬弃，不仅仅是语法和C语言有着很多差异，最重要的是舍弃了C语言中灵活但是危险的指针运算。而且，Go语言还重新设计了C语言中部分不太合理运算符的优先级，并在很多细微的地方都做了必要的打磨和改变。当然，C语言中少即是多、简单直接的暴力编程哲学则被Go语言更彻底地发扬光大了（Go语言居然只有25个关键字，spec语言规范还不到50页）。

Go语言的其它的一些特性零散地来自于其他一些编程语言；比如*iota*语法是从APL语言借鉴，词法作用域与嵌套函数等特性来自于Scheme语言（和其他很多编程语言）。Go语言中也有很多自己发明创新的设计。比如Go语言的切片为轻量级动态数组提供了有效的随机存取的性能，这可能会让人联想到链表的底层的共享机制。还有Go语言新发明的*defer*语句（Ken发明）也是神来之笔。

## 来自贝尔实验室特有基因

作为Go语言标志性的并发编程特性则来自于贝尔实验室的Tony Hoare于1978年发表鲜为外界所知的关于并发研究的基础文献：顺序通信进程（communicating sequential processes，缩写为CSP）。在最初的CSP论文中，程序只是一组没有中间共享状态的平行运行的处理过程，它们之间使用管道进行通信和控制同步。Tony Hoare的CSP并发模型只是一个用于描述并发性基本概念的描述语言，它并不是一个可以编写可执行程序的通用编程语言。

CSP并发模型最经典的实际应用是来自爱立信发明的Erlang编程语言。不过在Erlang将CSP理论作为并发编程模型的同时，同样来自贝尔实验室的Rob Pike以及其同事也在不断尝试将CSP并发模型引入当时的新发明的编程语言中。他们第一次尝试引入CSP并发特性的编程语言叫Squeak（老鼠的叫声），是一个用于提供鼠标和键盘事件处理的编程语言，在这个语言中管道是静态创建的。然后是改进版的Newsqueak语言（新版老鼠的叫声），新提供了类似C语言语句和表达式的语法，还有类似Pascal语言的推导语法。Newsqueak是一个带垃圾回收的纯函数式语言，它再次针对键盘、鼠标和窗口事件管理。但是在Newsqueak语言中管道已经是动态创建的，管道属于第一类值、可以保存到变量中。然后是Alef编程语言（Alef也是C语言之父Ritchie比较喜爱的编程语言），Alef语言试图将Newsqueak语言改造为系统编程语言，但是因为缺少垃圾回收机制而导致并发编程很痛苦（这也是继承C语言手工管理内存的代价）。在Alef语言之后还有一个叫Limbo的编程语言（地狱的意思），这是一个运行在虚拟机中的脚本语言。Limbo语言是Go语言最接近的祖先，它和Go语言有着最接近的语法。到设计Go语言时，Rob Pike在CSP并发编程模型的实践道路上已经积累了几十年的经验，关于Go语言并发编程的特性完全是信手拈来，新编程语言的到来也是水到渠成了。

可以从Go语言库早期代码库日志可以看出最直接的演化历程（Git用`git log --before={2008-03-03} --reverse`命令查看）：

```
C:\go\go-tip>hg log -r 0:4
changeset: 0:f6182e5abf5e
user: Brian Kernighan <bwk>
date: Tue Jul 18 19:05:45 1972 -0500
summary: hello, world

changeset: 1:b66d0bf8da3e
user: Brian Kernighan <bwk>
date: Sun Jan 20 01:02:03 1974 -0400
summary: convert to C

changeset: 2:ac3363d7e788
user: Brian Kernighan <research!bwk>
date: Fri Apr 01 02:02:04 1988 -0500
summary: convert to Draft-Proposed ANSI C

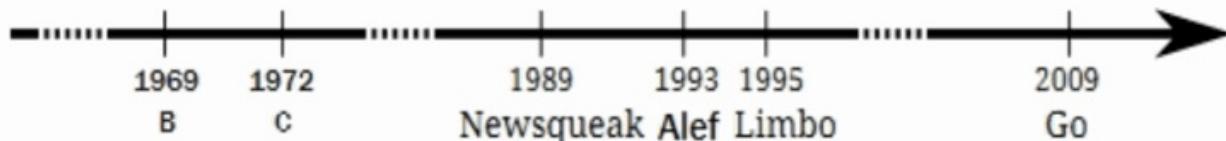
changeset: 3:172d32922e72
user: Brian Kernighan <bwk@research.att.com>
date: Fri Apr 01 02:03:04 1988 -0500
summary: last-minute fix: convert to ANSI C

changeset: 4:4e9a5b095532
user: Robert Griesemer <gri@golang.org>
date: Sun Mar 02 20:47:34 2008 -0800
summary: Go spec starting point.

C:\go\go-tip>
```

从早期提交日志中也可以看出，Go语言是从Ken Thompson发明的B语言、Dennis M. Ritchie发明的C语言逐步演化过来的，它首先是C语言家族的成员，因此很多人将Go语言称为21世纪的C语言。

下面是Go语言中来自贝尔实验室特有的并发编程基因的演化过程：



纵观整个贝尔实验室的编程语言的发展进程，从B语言、C语言、Newsqueak、Alef、Limbo语言一路走来，Go语言继承了来自贝尔实验室的半个世纪的软件设计基因，终于完成了C语言革新的使命。纵观这几年来的发展趋势，Go语言已经成为云计算、云存储时代最重要的基础编程语言。

## 你好，世界

按照惯例，介绍所有编程语言的第一个程序都是“Hello, World!”。虽然本教假设读者已经了解了Go语言，但是我们还是不想打破这个惯例（因为这个传统正是从Go语言的前辈C语言传承而来的）。不过，Go语言的这个程序输出的是中文“你好，世界!”。

```

package main

import "fmt"

func main() {
    fmt.Println("你好，世界!")
}

```

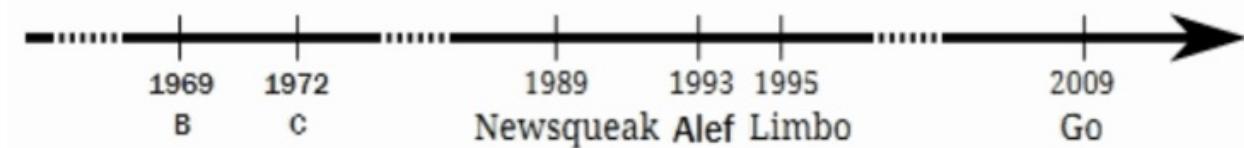
将以上代码保存到 `hello.go` 文件中。因为代码中有非ASCII的中文字符，我们需要将文件的编码显式指定为无BOM的UTF8编码格式（源文件采用UTF8编码是Go语言规范所要求的）。然后进入命令行并切换到 `hello.go` 文件所在的目录。目前我们可以将Go语言当作脚本语言，在命令行中直接输入 `go run hello.go` 来运行程序。如果一切正常的话。应该可以在命令行看到输出"你好，世界!"的结果。

现在，让我们简单介绍一下程序。所有的Go程序，都是由最基本的函数和变量构成，函数和变量被组织到一个个单独的Go源文件中，这些源文件再按照作者的意图组织成合适的 `package`，最终这些 `package` 再有机地组成一个完整的Go语言程序。其中，函数用于包含一系列的语句(指明要执行的操作序列)，以及执行操作时存放数据的变量。我们这个程序中函数的名字是 `main`。虽然Go语言中，函数的名字没有太多的限制，但是 `main` 包中的 `main` 函数默认是每一个可执行程序的入口。而 `package` 则用于包装和组织相关的函数、变量和常量。在使用一个 `package` 之前，我们需要使用 `import` 语句导入包。例如，我们这个程序中导入了 `fmt` 包 (`fmt` 是 `format` 单词的缩写，表示格式化相关的包)，然后我们才可以使用 `fmt` 包中的 `Println` 函数。

而双引号包含的“你好，世界!”则是Go语言的字符串面值常量。和C语言中的字符串不同，Go语言中的字符串内容是不可变更的。在以字符串作为参数传递给 `fmt.Println` 函数时，字符串的内容并没有被复制——传递的仅仅是字符串的地址和长度（字符串的结构在 `reflect.StringHeader` 中定义）。在Go语言中，函数参数都是以复制的方式(不支持以引用的方式)传递（比较特殊的是，Go语言闭包函数对外部变量是以引用的方式使用）。

## 1.2. Hello, World 的革命

在创世纪章节中我们简单介绍了 Go 语言的演化基因族谱，对其中来自于贝尔实验室的特有并发编程基因做了重点介绍，最后引出了 Go 语言版的“Hello, World”程序。其实“Hello, World”程序是展示各种语言特性的最好的例子，是通向该语言的一个窗口。这一节我们将沿着各个编程语言演化的时间轴，简单回顾下“Hello, World”程序是如何逐步演化到目前的 Go 语言形式、最终完成它的革命使命的。



### B 语言 - Ken Thompson, 1972

首先是 B 语言，B 语言是 Go 语言之父贝尔实验室的 Ken Thompson 早年间开发的一种通用的程序设计语言，设计目的是为了用于辅助 UNIX 系统的开发。但是因为 B 语言缺乏灵活的类型系统导致使用比较困难。后来，Ken Thompson 的同事 Dennis Ritchie 以 B 语言为基础开发出了 C 语言，C 语言提供了丰富的类型，极大地增加了语言的表达能力。到目前为止它依然是世界上最常用的程序语言之一。而 B 语言自从被它取代之后，则就只存在于各种文献之中，成为了历史。

目前见到的 B 语言版本的“Hello World”，一般认为是来自于 Brian W. Kernighan 编写的 B 语言入门教程（Go 核心代码库中的第一个提交者名字正是 Brian W. Kernighan），程序如下：

```
main() {
    extrn a, b, c;
    putchar(a); putchar(b); putchar(c);
    putchar('!*n');
}
a 'hell';
b 'o, w';
c 'orld';
```

由于 B 语言缺乏灵活的数据类型，只能分别以 a/b/c 全局变量来定义要输出的内容，并且每个变量的长度必须对齐到了 4 个字节（有一种写汇编语言的感觉）。然后通过多次调用 `putchar` 函数输出字符，最后的 `'!*n'` 表示输出一个换行的意思。

总体来说，B 语言简单，功能也比较简陋。

# C语言 - Dennis Ritchie, 1974 ~ 1989

C语言是由Dennis Ritchie在B语言的基础上改进而来，它增加了丰富的数据类型，并最终实现了用它重写UNIX的伟大目标。C语言可以说是现代IT行业最重要的软件基石，目前主流的操作系统几乎全部是由C语言开发的，许多基础系统软件也是C语言开发的。C系家族的编程语言占据统治地位达几十年之久，半个多世纪以来依然充满活力。

在Brian W. Kernighan于1974年左右编写的C语言入门教程中，出现了第一个C语言版本的“Hello World”程序。这给后来大部分编程语言教程都以“Hello World”为第一个程序提供了惯例。第一个C语言版本的“Hello World”程序如下：

```
main()
{
    printf("hello, world");
}
```

关于这个程序，有几点需要说明的：首先是 `main` 函数因为没有明确返回值类型，默认返回 `int` 类型；其次 `printf` 函数默认不需要导入函数声明即可以使用；最后 `main` 没有明确返回语句，但默认返回0值。在这个程序出现时，C语言还远未标准化，我们看到的是上古时代的C语言语法：函数不用写返回值，函数参数也可以忽略，使用`printf`时不需要包含头文件等。

这个例子同样出现在了1978年出版的《C程序设计语言》第一版中，作者正是Brian W. Kernighan 和 Dennis M. Ritchie（简称K&R）。书中的“Hello World”末尾增加了一个换行输出：

```
main()
{
    printf("hello, world\n");
}
```

这个例子在字符串末尾增加了一个换行，C语言的 `\n` 换行比B语言的 `'\n'` 换行看起来要简洁了一些。

在K&R的教程面世10年之后的1988年，《C程序设计语言》第二版终于出版了。此时ANSI C语言的标准化草案已经初步完成，但正式版本的文档尚未发布。不过书中的“Hello World”程序根据新的规范增加了 `#include <stdio.h>` 头文件包含语句，用于包含 `printf` 函数的声明（新的C89标准中，仅仅是针对 `printf` 函数而言，依然可以不用声明函数而直接使用）。

```
#include <stdio.h>

main()
{
    printf("hello, world\n");
}
```

然后到了1989年，ANSI C语言第一个国际标准发布，一般被称为C89。C89是流行最广泛的一个C语言标准，目前依然被大量使用。《C程序设计语言》第二版的也再次印刷新版本，并针对新发布的C89规范建议，给 `main` 函数的参数增加了 `void` 输入参数说明，表示没有输入参数的意思。

```
#include <stdio.h>

main(void)
{
    printf("hello, world\n");
}
```

至此，C语言本身的进化基本完成。后面的C92/C99/C11都只是针对一些语言细节做了完善。因为各种历史因素，C89依然是使用最广泛的标准。

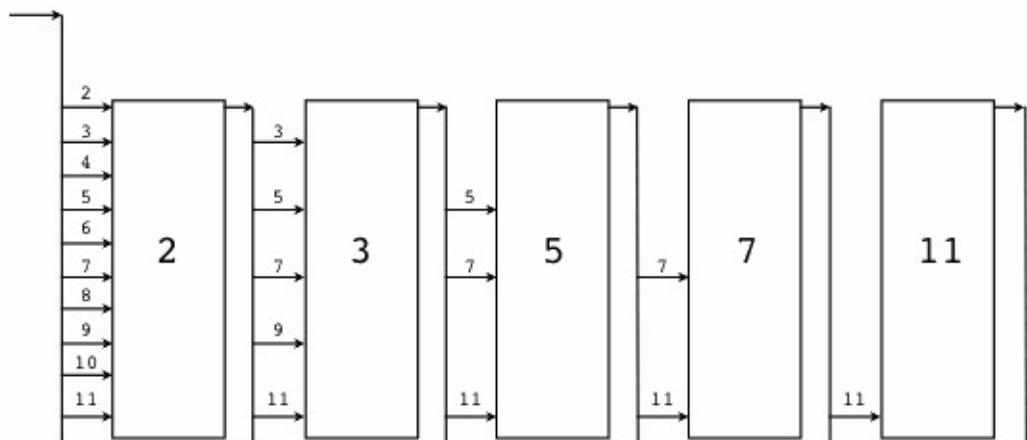
## Newsqueak - Rob Pike, 1989

Newsqueak是Rob Pike发明的老鼠语言的第二代，是他用于实践CSP并发编程模型的战场。Newsqueak是新的squeak语言的意思，其中squeak是老鼠吱吱吱的叫声，也可以看作是类似鼠标点击的声音。Squeak是一个提供鼠标和键盘事件处理的编程语言，Squeak语言的管道是静态创建的。改进版的Newsqueak语言则提供了类似C语言语句和表达式的语法和类似Pascal语言的推导语法。Newsqueak是一个带自动垃圾回收的纯函数式语言，它再次针对键盘、鼠标和窗口事件管理。但是在Newsqueak语言中管道是动态创建的，属于第一类值，因此可以保存到变量中。

Newsqueak类似脚本语言，内置了一个 `print` 函数，它的“Hello World”程序看不出什么特色：

```
print("Hello,", "World", "\n");
```

从上面的程序中，除了猜测 `print` 函数可以支持多个参数外，我们很难看到Newsqueak语言相关的特性。由于Newsqueak语言和Go语言相关的特性主要是并发和管道。因此，我们这里通过一个并发版本的“素数筛”算法来略窥Newsqueak语言的特性。“素数筛”的原理如图：



Newsqueak语言并发版本的“素数筛”程序如下：

```

// 向管道输出从2开始的自然数序列
counter := prog(c:chan of int) {
    i := 2;
    for(;;) {
        c <= i++;
    }
};

// 针对listen管道获取的数列，过滤掉是prime倍数的数
// 新的序列输出到send管道
filter := prog(prime:int, listen, send:chan of int) {
    i:int;
    for(;;) {
        if((i = <-listen)%prime) {
            send <= i;
        }
    }
};

// 主函数
// 每个管道第一个流出的数必然是素数
// 然后基于这个新的素数构建新的素数过滤器
sieve := prog() of chan of int {
    c := mk(chan of int);
    begin counter(c);
    prime := mk(chan of int);
    begin prog(){
        p:int;
        newc:chan of int;
        for(;;){
            prime <= p <= c;
            newc = mk();
            begin filter(p, c, newc);
            c = newc;
        }
    }();
    become prime;
};

// 启动素数筛
prime := sieve();

```

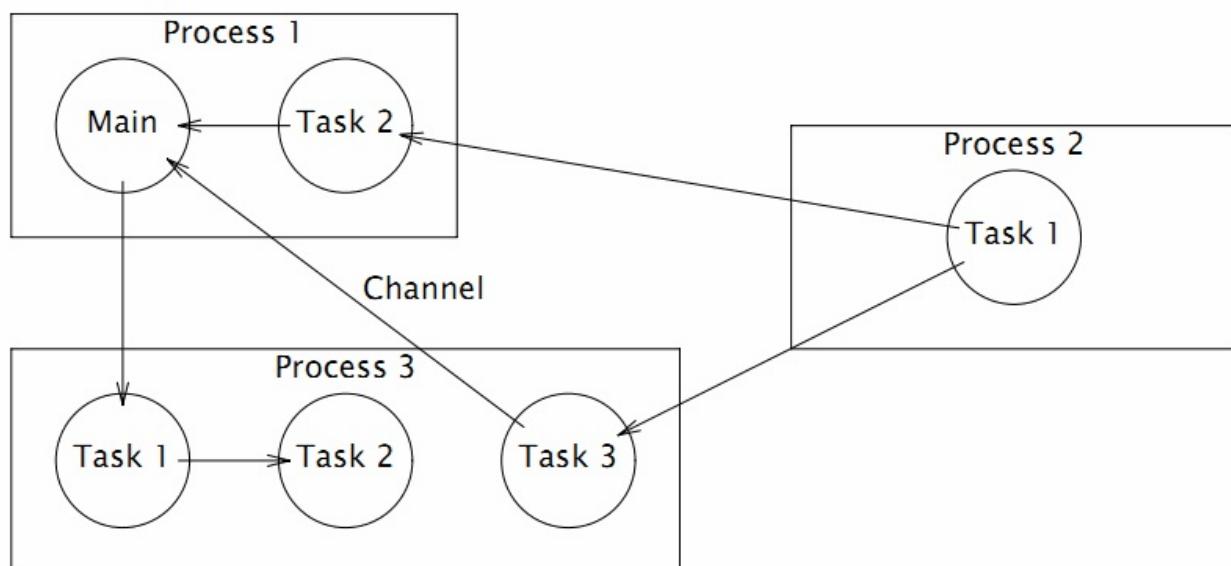
其中 `counter` 函数用于向管道输出原始的自然数序列，每个 `filter` 函数对象则对应每一个新的素数过滤管道，这些素数过滤管道根据当前的素数筛选将输入管道流入的数列筛选后重新输出到输出管道。`mk(chan of int)` 用于创建管道，类似Go语言的 `make(chan int)` 语句；`begin filter(p, c, newc)` 关键字启动素数筛选的并发体，类似Go语言的 `go filter(p, c, newc)` 语句；`become` 用于返回函数结果，类似 `return` 语句。

`Newspeak` 语言中并发体和管道的语法和 Go 语言已经比较接近了，后置的类型声明和 Go 语言的语法也很相似。

# Alef - Phil Winterbottom, 1993

在Go语言出现之前，Alef语言是作者心中比较完美的并发语言，Alef语法和运行时基本是无缝兼容C语言。Alef语言中的对线程和进程的并发体都提供了支持，其中 `proc receive(c)` 用于启动一个进程，`task receive(c)` 用于启动一个线程，它们之间通过管道 `c` 进行通讯。不过由于Alef缺乏内存自动回收机制，导致并发体的内存资源管理异常复杂。而且Alef语言只在Plan9系统中提供过短暂的支持，其它操作系统并没有实际可以运行的Alef开发环境。而且Alef语言只有《Alef语言规范》和《Alef编程向导》两个公开的文档，因此在贝尔实验室之外关于Alef语言的讨论并不多。

由于Alef语言同时支持进程和线程并发体，而且在并发体中可以再次启动更多的并发体，导致了Alef的并发状态会异常复杂。同时Alef没有自动垃圾回收机制（Alef因为保留的C语言灵活的指针特性，也导致了自动垃圾回收机制实现比较困难），各种资源充斥于不同的线程和进程之间，导致并发体的内存资源管理异常复杂。Alef语言全部继承了C语言的语法，可以认为是增强了并发语法的C语言。下图是Alef语言文档中展示的一个可能的并发体状态：



Alef语言并发版本的“Hello World”程序如下：

```
#include <alef.h>

void receive(chan(byte*) c) {
    byte *s;
    s = <- c;
    print("%s\n", s);
    terminate(nil);
}

void main(void) {
    chan(byte*) c;
    alloc c;
    proc receive(c);
    task receive(c);
    c <-= "hello proc or task";
    c <-= "hello proc or task";
    print("done\n");
    terminate(nil);
}
```

程序开头的 `#include <alef.h>` 语句用于包含Alef语言的运行时库。`receive` 是一个普通函数，程序中用作每个并发体的入口函数；`main` 函数中的 `alloc c` 语句先创建一个 `chan(byte*)` 类型的管道，类似Go语言的 `make(chan []byte)` 语句；然后分别启动以进程和线程的方式启动 `receive` 函数；启动并发体之后，`main` 函数向 `c` 管道发送了两个字符串数据；而进程和线程状态运行的 `receive` 函数会以不确定的顺序先后从管道收到数据后，然后分别打印字符串；最后每个并发体都通过调用 `terminate(nil)` 来结束自己。

Alef的语法和C语言基本保持一致，可以认为它是在C语言的语法基础上增加了并发编程相关的特性，可以看作是另一个维度的C++语言。

## Limbo - Sean Dorward, Phil Winterbottom, Rob Pike, 1995

**Limbo**（地狱）是用于开发运行在小型计算机上的分布式应用的编程语言，它支持模块化编程，编译期和运行时的强类型检查，进程内基于具有类型的通信管道，原子性垃圾收集和简单的抽象数据类型。Limbo被设计为：即便是在没有硬件内存保护的小型设备上，也能安全运行。Limbo语言主要运行在Inferno系统之上。

Limbo语言版本的“Hello World”程序如下：

```

implement Hello;

include "sys.m"; sys: Sys;
include "draw.m";

Hello: module
{
    init: fn(ctx: ref Draw->Context, args: list of string);
};

init(ctx: ref Draw->Context, args: list of string)
{
    sys = load Sys Sys->PATH;
    sys->print("hello, world\n");
}

```

从这个版本的“Hello World”程序中，我们已经可以发现很多Go语言特性的雏形。第一句 `implement Hello;` 基本对应Go语言的 `package Hello` 包声明语句。然后是 `include "sys.m"; sys: Sys;` 和 `include "draw.m";` 语句用于导入其它的模块，类似Go语言的 `import "sys"` 和 `import "draw"` 语句。然后Hello包模块还提供了模块初始化函数 `init`，并且函数的参数的类型也是后置的，不过Go语言的初始化函数是没有参数的。

## Go语言 - 2007~2009

贝尔实验室后来经历了多次动荡，包括Ken Thompson在内的Plan9项目原班人马最终加入了Google公司。在发明Limbo等前辈语言诞生10多年之后，在2007年底，Go语言三个最初的作者因为偶然的因素聚集到一起批斗C++（传说是C++语言的布道师在Google公司到处鼓吹的C++11各种牛逼特性彻底惹恼了他们），他们终于抽出了20%的自由时间创造了Go语言。最初的Go语言规范从2008年3月开始编写，最初的Go程序也是直接编译到C语言然后再二次编译为机器码。到了2008年5月，Google公司的领导们终于发现了Go语言的巨大潜力，从而开始全力支持这个项目（Google的创始人甚至还贡献了 `func` 关键字），让他们可以将全部工作时间投入到Go语言的设计和开发中。在Go语言规范初版完成之后，Go语言的编译器终于可以直接生成机器码了。

## hello.go - 2008年6月

```

package main

func main() int {
    print "hello, world\n";
    return 0;
}

```

这是初期Go语言程序正式开始测试的版本。其中内置的用于调试的 `print` 语句已经存在，不过是以命令的方式使用。入口 `main` 函数还和C语言中的 `main` 函数一样返回 `int` 类型的值，而且需要 `return` 显式地返回值。每个语句末尾的分号也还存在。

## hello.go - 2008年6月27日

```
package main

func main() {
    print "hello, world\n";
}
```

入口函数 `main` 已经去掉了返回值，程序默认通过隐式调用 `exit(0)` 来返回。Go语言朝着简单的方向逐步进化。

## hello.go - 2008年8月11日

```
package main

func main() {
    print("hello, world\n");
}
```

用于调试的内置的 `print` 由开始的命令改为普通的内置函数，使得语法更加简单一致。

## hello.go - 2008年10月24日

```
package main

import "fmt"

func main() {
    fmt.printf("hello, world\n");
}
```

作为C语言中招牌的 `printf` 格式化函数已经移植到了Go语言中，函数放在 `fmt` 包中（`fmt` 是格式化单词 `format` 的缩写）。不过 `printf` 函数名的开头字母依然是小写字母，采用大写字母表示导出的特性还没有出现。

## hello.go - 2009年1月15日

```
package main

import "fmt"

func main() {
    fmt.Println("hello, world\n");
}
```

Go语言开始采用是否大小写首字母来区分符号是否可以被导出。大写字母开头表示导出的公共符号，小写字母开头表示包内部的私有符号。国内用户需要注意的是，汉字中没有大小写字母的概念，因此以汉字开头的符号目前是无法导出的（针对问题中国用户已经给出相关建议，等Go2之后或许会调整对汉字的导出规则）。

## hello.go - 2009年12月11日

```
package main

import "fmt"

func main() {
    fmt.Println("hello, world\n")
}
```

Go语言终于移除了语句末尾的分号。这是Go语言在2009年11月10号正式开源之后第一个比较重要的语法改进。从1978年C语言教程第一版引入的分号分割的规则到现在，Go语言的作者们花了整整32年终于移除了语句末尾的分号。在这32年的演化的过程中必然充满了各种八卦故事，我想这一定是Go语言设计者深思熟虑的结果（现在Swift等新的语言也是默认忽略分号的，可见分号确实并不是那么的重要）。

## CGO版本

Go语言开源初期就支持通过CGO和C语言保持交互。CGO通过导入一个虚拟的 "c" 包来访问C语言中的函数。下面是CGO版本的“Hello World”程序：

```

package main

// #include <stdio.h>
// #include <stdlib.h>
import "C"
import "unsafe"

func main() {
    msg := C.CString("Hello, World!\n")
    defer C.free(unsafe.Pointer(msg))

    C.fputs(msg, C.stdout)
}

```

先通过 `ccstring` 函数将 Go 语言字符串转为 C 语言字符串，然后调用 C 语言的 `C.fputs` 函数向标准输出窗口打印转换后的 C 字符串。`defer` 延迟语句保证程序返回前通过 `C.free` 释放分配的 C 字符串。需要注意的是，CGO 不支持 C 语言中的可变参数函数（因为 Go 语言每次函数调用的栈帧大小是固定的，而且 Go 语言中可变参数语法只是切片的一个语法糖而已），因此在 Go 语言中是无法通过 CGO 访问 C 语言的 `printf` 等可变参数函数的。同时，CGO 只能访问 C 语言的函数、变量和简单的宏定义常量，CGO 并不支持访问 C++ 语言的符号（C++ 和 C 语言符号的名字修饰规则不同，CGO 采用 C 语言的名字修饰规则）。

其实 CGO 不仅仅用于在 Go 语言中调用 C 语言函数，还可以用于导出 Go 语言函数给 C 语言函数调用。在用 Go 语言编写生成 C 语言的静、动态库时，也可以用 CGO 导出对应的接口函数。正是 CGO 的存在，才保证了 Go 语言和 C 语言资源的双向互通，同时保证了 Go 语言可以继承 C 语言已有的庞大的软件资产。

## SWIG 版本

Go 语言开源初期除了支持通过 CGO 访问 C 语言资源外，还支持通过 SWIG 访问 C/C++ 接口。SWIG 是从 2010 年 10 月 04 日发布的 SWIG-2.0.1 版本开始正式支持 Go 语言的。可以将 SWIG 看作一个高级的 CGO 代码自动生成器，同时通过生成 C 语言桥接代码增加了对 C++ 类的支持。下面是 SWIG 版本的 "Hello World" 程序：

首先是创建一个 `hello.cc` 文件，里面有 `SayHello` 函数用于打印（这里的 `SayHello` 函数采用 C++ 的名字修饰规则）：

```

#include <iostream>

void SayHello() {
    std::cout << "Hello, World!" << std::endl;
}

```

然后创建一个 `hello.swigcxx` 文件，以SWIG语法导出上面的C++函数 `SayHello`：

```
%module main

%inline %{
extern void SayHello();
%}
```

然后在Go语言中直接访问 `SayHello` 函数（首字母自动转为大写字母）：

```
package main

import (
    hello "."
)

func main() {
    hello.SayHello()
}
```

需要将上述3个文件放到同一个目录中，并且 `hello.swigcxx` 和Go文件对应同一个包。系统除了需要安装Go语言环境外，还需要安装对应版本的SWIG工具。最后运行 `go build` 就可以构建了。

注：在Windows系统下，路径最长为260个字符。这个程序生成的中间cgo文件可能导致某些文件的绝对路径长度超出Windows系统限制，可能导致程序构建失败。这是由于 `go build` 调用 `swig` 和 `cgo` 等命令生成中间文件时生成的不合适的超长文件名导致（作者提交[ISSUE3358](#)，`Go1.8`已经修复）。

## Go汇编语言版本

Go语言底层使用了自己独有的跨操作系统汇编语言，该汇编语言是从Plan9系统的汇编语言演化而来。Go汇编语言并不能独立使用，它是属于Go语言的一个组成部分，必须以Go语言包的方式被组织。下面是Go汇编语言版本的“Hello World”程序：

先创建一个 `main.go` 文件，以Go语言的语法声明包和声明汇编语言对应的函数签名，函数签名不能有函数体：

```
package main

func main()
```

然后创建 `main_amd64.s` 文件，对应Go汇编语言实现AMD64架构的 `main` 函数：

```

#include "textflag.h"
#include "funcdata.h"

// "Hello World!\n"
DATA text<>+0(SB)/8,$"Hello Wo"
DATA text<>+8(SB)/8,$"rld!\n"
GLOBAL text<>(SB), NOPTR, $16

// func main()
TEXT ·main(SB), $16-0
NO_LOCAL_POINTERS
MOVQ $text<>+0(SB), AX
MOVQ AX, (SP)
MOVQ $16, 8(SP)
CALL runtime·printstring(SB)
RET

```

代码中 `#include "textflag.h"` 语句包含运行时库定义的头文件，里面有 `NOPTR / NO_LOCAL_POINTERS` 等基本的宏的定义。`DATA` 汇编指令用于定义数据，每个数据的宽度必须是`1/2/4/8`，然后 `GLOBL` 汇编命令在当前文件内导出 `text` 变量符号。`TEXT ·main(SB), $16-0` 用于定义 `main` 函数，其中 `$16-0` 表示 `main` 函数的帧大小是16个字节（对应 `string` 头的大小，用于给 `runtime·printstring` 函数传递参数），`0` 表示 `main` 函数没有参数和返回值。`main` 函数内部通过调用运行时内部的 `runtime·printstring(SB)` 函数来打印字符串。

Go汇编语言虽然针对每种CPU架构（主要有386/AMD64/ARM／ARM64等）有对应的指令和寄存器，但是汇编语言的基本语法和函数调用规范是一致的，不同操作系统之间用法是一致的。在Go语言标准库中，`runtime` 运行时库、`math` 数学库和`crypto` 密码相关的函数很多是采用汇编语言实现的。其中 `runtime` 运行时库中采用部分汇编语言并不完全是为了性能，而是运行时的某些特性功能（比如`goroutine`上下文的切换等）无法用纯Go实现，因此需要汇编代码实现某些辅助功能。对于普通用户而言，Go汇编语言的最大价值在于性能的优化，对于性能比较关键的地方，可以尝试用Go汇编语言实现终极优化。

## 你好，世界！ - V2.0

在经过半个世纪的涅槃重生之后，Go语言不仅仅打印出了Unicode版本的“Hello, World”，而且可以方便地向全球用户提供打印服务。下面版本通过 `http` 服务向每个访问的客户端打印中文的“你好，世界！”和当前的时间信息。

```
package main

import (
    "fmt"
    "log"
    "net/http"
    "time"
)

func main() {
    fmt.Println("Please visit http://127.0.0.1:12345/")
    http.HandleFunc("/", func(w http.ResponseWriter, req *http.Request) {
        s := fmt.Sprintf("你好, 世界! -- Time: %s", time.Now().String())
        fmt.Fprintf(w, "%v\n", s)
        log.Printf("%v\n", s)
    })
    if err := http.ListenAndServe(":12345", nil); err != nil {
        log.Fatal("ListenAndServe: ", err)
    }
}
```

我们通过Go语言标准库自带的 `net/http` 包构造了一个独立运行的http服务。其中 `http.HandleFunc("/", ...)` 针对 / 根路径请求注册了响应处理函数。在响应处理函数中，我们依然使用 `fmt.Fprintf` 格式化输出函数实现了通过http协议向请求的客户端打印格式化的字符串，同时通过标准库的日志包在服务器端也打印相关字符串。最后通过 `http.ListenAndServe` 函数调用来启动http服务。

至此，Go语言终于完成了从单机单核时代的C语言到21世纪互联网时代多核环境的通用编程语言的蜕变。

## 1.3. 数组、字符串和切片

在主流的编程语言中数组及其相关数据结构是使用得最为频繁的，只有在它(们)不能满足时才会考虑链表、hash表（hash表可以看作是数组和链表的混合体）和更复杂的自定义数据结构。

Go语言中数组、字符串和切片三者是密切相关的数据结构。这三种数据类型，在底层原始数据有着相同的内存结构，在上层，因为语法的限制而有着不同的行为表现。首先，Go语言的数组是一种值类型，虽然数组的元素可以被修改，但是数组本身的赋值和函数传参都是以整体复制的方式处理的。Go语言字符串底层数据也是对应的字节数组，但是字符串的只读属性禁止了在程序中对底层字节数组的元素的修改。字符串赋值只是复制了数据地址和对应的长度，而不会导致底层数据的复制。切片的行为更为灵活，切片的结构和字符串结构类似，但是解除了只读限制。切片的底层数据虽然也是对应数据类型的数组，但是每个切片还有独立的长度和容量信息，切片赋值和函数传参数时也是将切片头信息部分按传值方式处理。因为切片头含有底层数据的指针，所以它的赋值也不会导致底层数据的复制。其实Go语言的赋值和函数传参规则很简单，除了闭包函数以引用的方式对外部变量访问之外，其它赋值和函数传参数都是以传值的方式处理。要理解数组、字符串和切片三种不同的处理方式的原因需要详细了解它们的底层数据结构。

### 数组

数组是一个由固定长度的特定类型元素组成的序列，一个数组可以由零个或多个元素组成。数组的长度是数组类型的组成部分。因为数组的长度是数组类型的一个部分，不同长度或不同类型的数据组成的数组都是不同的类型，因此在Go语言中很少直接使用数组（不同长度的数组因为类型不同无法直接赋值）。和数组对应的类型是切片，切片是可以动态增长和收缩的序列，切片的功能也更加灵活，但是要理解切片的工作原理还是要先理解数组。

我们先看看数组有哪些定义方式：

```
var a [3]int           // 定义一个长度为3的int类型数组，元素全部为0
var b = [...]int{1, 2, 3} // 定义一个长度为3的int类型数组，元素为 1, 2, 3
var c = [...]int{2: 3, 1: 2} // 定义一个长度为3的int类型数组，元素为 0, 2, 3
var d = [...]int{1, 2, 4: 5, 6} // 定义一个长度为6的int类型数组，元素为 1, 2, 0, 0, 5, 6
```

第一种方式是定义一个数组变量的最基本的方式，数组的长度明确指定，数组中的每个元素都以零值初始化。

第二种方式定义数组，可以在定义的时候顺序指定全部元素的初始化值，数组的长度根据初始化元素的数目自动计算。

第三种方式是以索引的方式来初始化数组的元素，因此元素的初始化值出现顺序比较随意。这种初始化方式和 `map[int]Type` 类型的初始化语法类似。数组的长度以出现的最大的索引为准，没有明确初始化的元素依然用 0 值初始化。

第四种方式是混合了第二种和第三种的初始化方式，前面两个元素采用顺序初始化，第三第四个元素零值初始化，第五个元素通过索引初始化，最后一个元素跟在前面的第五个元素之后采用顺序初始化。

数组的内存结构比较简单。比如下面是一个 `[4]int{2,3,5,7}` 数组值对应的内存结构：

```
primes := [4]int{2,3,5,7}
[ 2 | 3 | 5 | 7 ] [4]int
```

Go 语言中数组是值语义。一个数组变量即表示整个数组，它并不是隐式的指向第一个元素的指针（比如 C 语言的数组），而是一个完整的值。当一个数组变量被赋值或者被传递的时候，实际上会复制整个数组。如果数组较大的话，数组的赋值也会有较大的开销。为了避免复制数组带来的开销，可以传递一个指向数组的指针，但是数组指针并不是数组。

```
var a = [...]int{1, 2, 3} // a 是一个数组
var b = &a                // b 是指向数组的指针

fmt.Println(a[0], a[1])    // 打印数组的前2个元素
fmt.Println(b[0], b[1])    // 通过数组指针访问数组元素的方式和数组类似

for i, v := range b {      // 通过数组指针迭代数组的元素
    fmt.Println(i, v)
}
```

其中 `b` 是指向 `a` 数组的指针，但是通过 `b` 访问数组中元素的写法和 `a` 类似的。还可以通过 `for range` 来迭代数组指针指向的数组元素。其实数组指针类型除了类型和数组不同之外，通过数组指针操作数组的方式和通过数组本身的操作类似，而且数组指针赋值时只会拷贝一个指针。但是数组指针类型依然不够灵活，因为数组的长度是数组类型的组成部分，指向不同长度数组的数组指针类型也是完全不同的。

可以将数组看作一个特殊的结构体，结构的字段名对应数组的索引，同时结构体成员的数目是固定的。内置函数 `len` 可以用于计算数组的长度，`cap` 函数可以用于计算数组的容量。不过对于数组类型来说，`len` 和 `cap` 函数返回的结果始终是一样的，都是对应数组类型的长度。

我们可以用 `for` 循环来迭代数组。下面常见的几种方式都可以用来遍历数组：

```
for i := range a {
    fmt.Printf("b[%d]: %d\n", i, b[i])
}
for i, v := range b {
    fmt.Printf("b[%d]: %d\n", i, v)
}
for i := 0; i < len(c); i++ {
    fmt.Printf("b[%d]: %d\n", i, b[i])
}
```

用 `for range` 方式迭代的性能可能会更好一些，因为这种迭代可以保证不会出现数组越界的情形，每轮迭代对数组元素的访问时可以省去对下标越界的判断。

用 `for range` 方式迭代，还可以忽略迭代时的下标：

```
var times [5][0]int
for range times {
    fmt.Println("hello")
}
```

其中 `times` 对应一个 `[5][0]int` 类型的数组，虽然第一维数组有长度，但是数组的元素 `[0]int` 大小是 0，因此整个数组占用的内存大小依然是 0。没有付出额外的内存代价，我们就通过 `for range` 方式实现了 `times` 次快速迭代。

数组不仅仅可以用于数值类型，还可以定义字符串数组、结构体数组、函数数组、接口数组、管道数组等等：

```
// 字符串数组
var s1 = [2]string{"hello", "world"}
var s2 = [...]string{"你好", "世界"}
var s3 = [...]string{1: "世界", 0: "你好", }

// 结构体数组
var line1 [2]image.Point
var line2 = [...]image.Point{image.Point{X: 0, Y: 0}, image.Point{X: 1, Y: 1}}
var line3 = [...]image.Point{{0, 0}, {1, 1}}

// 图像解码器数组
var decoder1 [2]func(io.Reader) (image.Image, error)
var decoder2 = [...]func(io.Reader) (image.Image, error){
    png.Decode,
    jpeg.Decode,
}

// 接口数组
var unknown1 [2]interface{}
var unknown2 = [...]interface{}{123, "你好"}

// 管道数组
var chanList = [2]chan int{}
```

我们还可以定义一个空的数组：

```
var d [0]int      // 定义一个长度为0的数组
var e = [0]int{}  // 定义一个长度为0的数组
var f = [...]int{} // 定义一个长度为0的数组
```

长度为0的数组在内存中并不占用空间。空数组虽然很少直接使用，但是可以用于强调某种特有类型的操作时避免分配额外的内存空间，比如用于管道的同步操作：

```
c1 := make(chan [0]int)
go func() {
    fmt.Println("c1")
    c1 <- [0]int{}
}()
<-c1
```

在这里，我们并不关心管道中传输数据的真实类型，其中管道接收和发送操作只是用于消息的同步。对于这种场景，我们用空数组来作为管道类型可以减少管道元素赋值时的开销。当然一般更倾向于用无类型的匿名结构体代替：

```
c2 := make(chan struct{})
go func() {
    fmt.Println("c2")
    c2 <- struct{}{} // struct{}{}部分是类型, {}表示对应的结构体值
}()
<-c2
```

我们可以用 `fmt.Printf` 函数提供的 `%T` 或 `%#v` 谓词语法来打印数组的类型和详细信息：

```
fmt.Printf("b: %T\n", b) // b: [3]int
fmt.Printf("b: %#v\n", b) // b: [3]int{1, 2, 3}
```

在Go语言中，数组类型是切片和字符串等结构的基础。以上数组的很多操作都可以直接用于字符串或切片中。

## 字符串

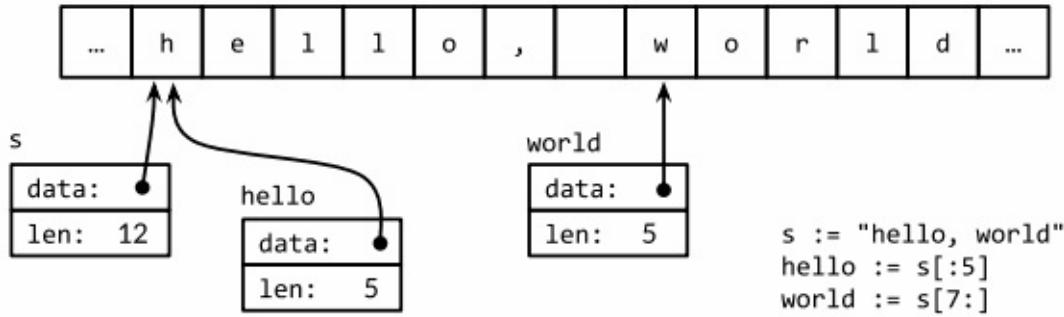
一个字符串是一个不可改变的字节序列，字符串通常是用来包含人类可读的文本数据。和数组不同的是，字符串的元素不可修改，是一个只读的字节数组。每个字符串的长度虽然也是固定的，但是字符串的长度并不是字符串类型的一部分。由于Go语言的源代码要求是UTF8编码，导致Go源代码中出现的字符串面值常量一般也是UTF8编码的。源代码中的文本字符串通常被解释为采用UTF8编码的Unicode码点（rune）序列。因为字节序列对应的是只读的字节序列，因此字符串可以包含任意的数据，包括byte值0。我们也可以用字符串表示GBK等非UTF8编码的数据，不过这种时候将字符串看作是一个只读的二进制数组更准确，因为 `for range` 等语法并不能支持非UTF8编码的字符串的遍历。

Go语言字符串的底层结构在 `reflect.StringHeader` 中定义：

```
type StringHeader struct {
    Data uintptr
    Len   int
}
```

字符串结构由两个信息组成：第一个是字符串指向的底层字节数组，第二个是字符串的字节的长度。字符串其实是一个结构体，因此字符串的赋值操作也就是 `reflect.StringHeader` 结构体的复制过程，并不会涉及底层字节数组的复制。在前面数组一节提到的 `[2]string` 字符串数组对应的底层结构和 `[2]reflect.StringHeader` 对应的底层结构是一样的，可以将字符串数组看作一个结构体数组。

我们可以看看字符串“Hello, world”本身对应的内存结构：



分析可以发现，“Hello, world”字符串底层数据和以下数组是完全一致的：

```
var data = [...]byte{'h', 'e', 'l', 'l', 'o', ',', ' ', 'w', 'o', 'r', 'l', 'd'}
```

字符串虽然不是切片，但是支持切片操作，不同位置的切片底层也访问的同一块内存数据（因为字符串是只读的，相同的字符串面值常量通常是对同一个字符串常量）：

```

s := "hello, world"
hello := s[:5]
world := s[7:]

s1 := "hello, world"[:5]
s2 := "hello, world"[7:]

```

和数组一样，内置的 `len` 和 `cap` 函数返回相同的结果，都对应字符串的长度。也可以通过 `reflect.StringHeader` 结构访问字符串的长度（这里只是为了演示字符串的结构，并不是推荐的做法）：

```

fmt.Println("len(s):", (*reflect.StringHeader)(unsafe.Pointer(&s)).Len) // 12
fmt.Println("len(s1):", (*reflect.StringHeader)(unsafe.Pointer(&s1)).Len) // 5
fmt.Println("len(s2):", (*reflect.StringHeader)(unsafe.Pointer(&s2)).Len) // 5

```

根据Go语言规范，Go语言的源文件都是采用UTF8编码。因此，Go源文件中出现的字符串面值常量一般也是UTF8编码的（对于转义字符，则没有这个限制）。提到Go字符串时，我们一般都会假设字符串对应的是一个合法的UTF8编码的字符序列。可以用内置的 `print` 调试函数或 `fmt.Print` 函数直接打印，也可以用 `for range` 循环直接遍历UTF8解码后的Unicode码点值。

下面的“Hello, 世界”字符串中包含了中文字符，可以通过打印转型为字节类型来查看字符底层对应的数据：

```
fmt.Printf("%#v\n", []byte("Hello, 世界"))
```

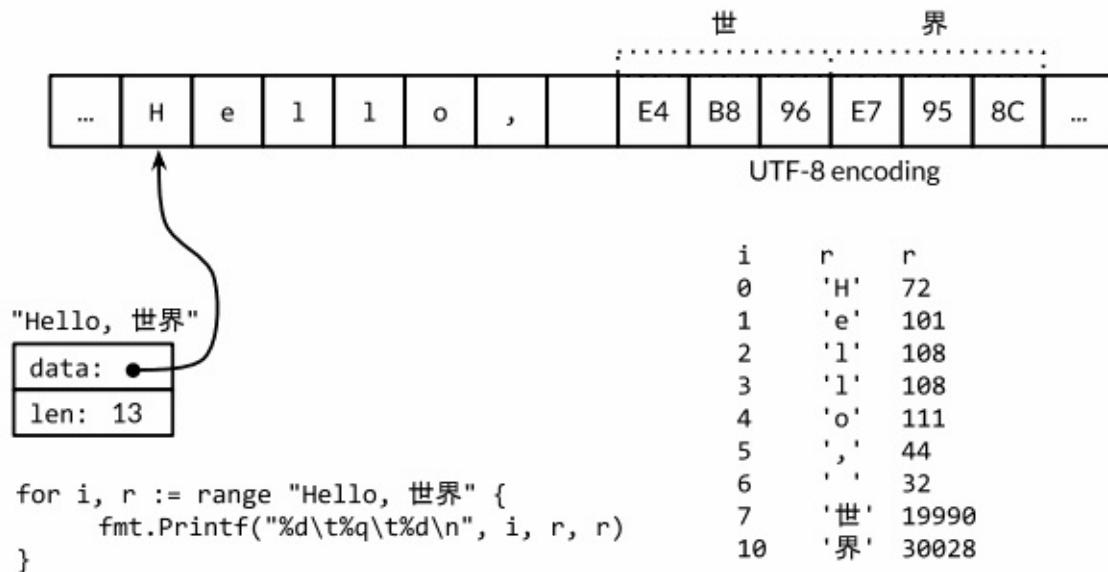
输出的结果是：

```
[]byte{0x48, 0x65, 0x6c, 0x6c, 0x6f, 0x2c, 0x20, 0xe4, 0xb8, 0x96, 0xe7, 0x95, 0x8c}
```

分析可以发现 `0xe4, 0xb8, 0x96` 对应中文“世”，`0xe7, 0x95, 0x8c` 对应中文“界”。我们也可以在字符串面值中直指定UTF8编码后的值（源文件中全部是ASCII码，可以避免出现多字节的字符）。

```
fmt.Println("\xe4\xb8\x96") // 打印: 世
fmt.Println("\xe7\x95\x8c") // 打印: 界
```

下图展示了“Hello, 世界”字符串的内存结构布局：



Go语言的字符串中可以存放任意的二进制字节序列，而且即使是UTF8字符序列也可能会遇到坏的编码。如果遇到一个错误的UTF8编码输入，将生成一个特别的Unicode字符‘\uFFFD’，这个字符在不同的软件中的显示效果可能不太一样，在印刷中这个符号通常是一个黑色六角形或钻石形状，里面包含一个白色的问号‘?’。

下面的字符串中，我们故意损坏了第一字符的第二和第三字节，因此第一字符将会打印为“?”，第二和第三字节则被忽略，后面的“abc”依然可以正常解码打印（错误编码不会向前扩散是UTF8编码的优秀特性之一）。

```
fmt.Println("\xe4\x00\x00\xe7\x95\x8cabc") // ?界abc
```

不过在 `for range` 迭代这个含有损坏的UTF8字符串时，第一字符的第二和第三字节依然会被单独迭代到，不过此时迭代的值是损坏后的0：

```

for i, c := range "\xe4\x00\x00\xe7\x95\x8cabc" {
    fmt.Println(i, c)
}
// 0 65533 // \uFFFD, 对应 ？
// 1 0      // 空字符
// 2 0      // 空字符
// 3 30028 // 界
// 6 97     // a
// 7 98     // b
// 8 99     // c

```

如果不想解码UTF8字符串，想直接遍历原始的字节码，可以将字符串强制转为 `[]byte` 字节序列后再行遍历（这里的转换一般不会产生运行时开销）：

```

for i, c := range []byte("世界abc") {
    fmt.Println(i, c)
}

```

或者是采用传统的下标方式遍历字符串的字节数组：

```

const s = "\xe4\x00\x00\x00\xe7\x95\x8cabc"
for i := 0; i < len(s); i++ {
    fmt.Printf("%d %x\n", i, s[i])
}

```

Go语言除了 `for range` 语法对UTF8字符串提供了特殊支持外，还对字符串和 `[]rune` 类型的相互转换提供了特殊的支持。

```

fmt.Printf("%#v\n", []rune("Hello, 世界"))      // []int32{19990, 30028}
fmt.Printf("%#v\n", string([]rune{'世', '界'})) // 世界

```

从上面代码的输出结果来看，我们可以发现 `[]rune` 其实是 `[]int32` 类型，这里的 `rune` 只是 `int32` 类型的别名，并不是重新定义的类型。`rune` 用于表示每个Unicode码点，目前只使用了21个bit位。

字符串相关的强制类型转换主要涉及到 `[]byte` 和 `[]rune` 两种类型。每个转换都可能隐含重新分配内存的代价，最坏的情况下它们的运算时间复杂度都是  $O(n)$ 。不过字符串和 `[]rune` 的转换要更为特殊一些，因为一般这种强制类型转换要求两个类型的底层内存结构要尽量一致，显然它们底层对应的 `[]byte` 和 `[]int32` 类型是完全不同的内部布局，因此这种转换可能隐含重新分配内存的操作。

下面分别用伪代码简单模拟Go语言对字符串内置的一些操作，这样对每个操作的处理的时间复杂度和空间复杂度都会有较明确的认识。

**for range** 对字符串的迭代模拟实现

```
func forOnString(s string, forBody func(i int, r rune)) {
    for i := 0; len(s) > 0; {
        r, size := utf8.DecodeRuneInString(str)
        forBody(i, r)
        s = s[size:]
        i += size
    }
}
```

`for range` 迭代字符串时，每次解码一个Unicode字符，然后进入 `for` 循环体，遇到崩坏的编码并不会导致迭代停止。

**[]byte(s)** 转换模拟实现

```
func str2bytes(s string) []byte {
    p := make([]byte, len(s))
    for i := 0; i < len(s); i++ {
        c := s[i]
        p[i] = c
    }
    return p
}
```

模拟实现中新创建了一个切片，然后将字符串的数组逐一复制到了切片中，这是为了保证字符串只读的语义。当然，在将字符串转为 `[]byte` 时，如果转换后的变量并没有被修改的情形，编译器可能会直接返回原始的字符串对应的底层数据。

**string(bytes)** 转换模拟实现

```
func bytes2str(s []byte) (p string) {
    data := make([]byte, len(s))
    for i, c := s {
        data[i] = c
    }

    hdr := (*reflect.StringHeader)(unsafe.Pointer(&p))
    hdr.Data = uintptr(unsafe.Pointer(&data[0]))
    hdr.Len = len(s)

    return p
}
```

因为 Go 语言的字符串是只读的，无法直接同构构造底层字节数组生成字符串。在模拟实现中通过 `unsafe` 包获取了字符串的底层数据结构，然后将切片的数据逐一复制到了字符串中，这同样是为了保证字符串只读的语义不会收切片的影响。如果转换后的字符串在生命周期中原

始的 `[]byte` 的变量并不会发生变化，编译器可能会直接基于 `[]byte` 底层的数据构建字符串。

### `[]rune(s)` 转换模拟实现

```
func str2runes(s []byte) []rune {
    var p []int32
    for len(s) > 0 {
        r, size := utf8.DecodeRuneInString(s)
        p = append(p, r)
        s = s[size:]
    }
    return []rune(p)
}
```

因为底层内存结构的差异，字符串到 `[]rune` 的转换必然会导致重新分配 `[]rune` 内存空间，然后依次解码并复制对应的 Unicode 码点值。这种强制转换并不存在前面提到的字符串和字节切片转化时的优化情况。

### `string(runes)` 转换模拟实现

```
func runes2string(s []int32) string {
    var p []byte
    buf := make([]byte, 3)
    for _, r := range s {
        n := utf8.EncodeRune(buf, r)
        p = append(p, buf[:n]...)
    }
    return string(p)
}
```

同样因为底层内存结构的差异，`[]rune` 到字符串的转换也必然会导致重新构造字符串。这种强制转换并不存在前面提到的优化情况。

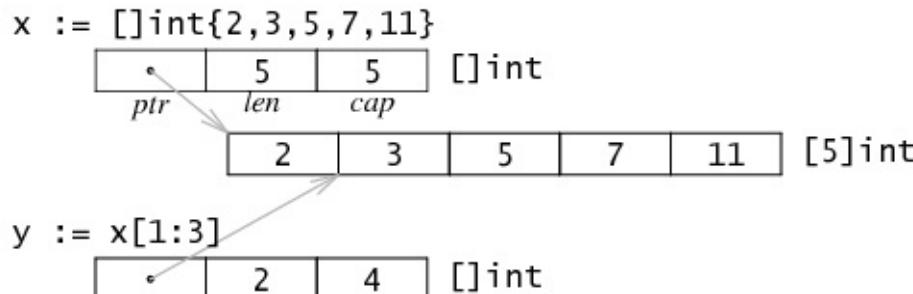
## 切片 (slice)

简单地说，切片就是一种简化版的动态数组。因为动态数组的长度是不固定，切片的长度自然也就不能是类型的组成部分了。数组虽然有适用它们的地方，但是数组的类型和操作都不够灵活，因此在 Go 代码中数组使用的并不多。而切片则使用得相当广泛，理解切片的原理和用法是一个 Go 程序员的必备技能。

我们先看看切片的结构定义，`reflect.SliceHeader`：

```
type SliceHeader struct {
    Data uintptr
    Len  int
    Cap  int
}
```

可以看出切片的开头部分和Go字符串是一样的，但是切片多了一个 `cap` 成员表示切片指向的内存空间的最大容量（对应元素的个数，而不是字节数）。下图是 `x := []int{2,3,5,7,11}` 和 `y := x[1:3]` 两个切片对应的内存结构。



让我们看看切片有哪些定义方式：

```
var (
    a []int          // nil切片, 和 nil 相等, 一般用来表示一个不存在的切片
    b = []int{}      // 空切片, 和 nil 不相等, 一般用来表示一个空的集合
    c = []int{1, 2, 3} // 有3个元素的切片, len和cap都为3
    d = c[:2]        // 有2个元素的切片, len为2, cap为3
    e = c[0:2:cap(c)] // 有2个元素的切片, len为2, cap为3
    f = c[:0]        // 有0个元素的切片, len为0, cap为3
    g = make([]int, 3) // 有3个元素的切片, len和cap都为3
    h = make([]int, 2, 3) // 有2个元素的切片, len为2, cap为3
    i = make([]int, 0, 3) // 有0个元素的切片, len为0, cap为3
)
```

和数组一样，内置的 `len` 函数返回切片中有效元素的长度，内置的 `cap` 函数返回切片容量大小，容量必须大于或等于切片的长度。也可以通过 `reflect.SliceHeader` 结构访问切片的信息（只是为了说明切片的结构，并不是推荐的做法）。切片可以和 `nil` 进行比较，只有当切片底层数据指针为空时切片本身为 `nil`，这时候切片的长度和容量信息将是无效的。如果有切片的底层数据指针为空，但是长度和容量不为0的情况，那么说明切片本身已经被损坏了（比如直接通过 `reflect.SliceHeader` 或 `unsafe` 包对切片作了不正确的修改）。

遍历切片的方式和遍历数组的方式类似：

```

for i := range a {
    fmt.Printf("b[%d]: %d\n", i, a[i])
}
for i, v := range b {
    fmt.Printf("b[%d]: %d\n", i, v)
}
for i := 0; i < len(c); i++ {
    fmt.Printf("b[%d]: %d\n", i, c[i])
}

```

其实除了遍历之外，只要是切片的底层数据指针、长度和容量没有发生变化的话，对切片的遍历、元素的读取和修改都和数组是一样的。在对切片本身赋值或参数传递时，和数组指针的操作方式类似，只是复制切片头信息（`reflect.SliceHeader`），并不会复制底层的数据。对于类型，和数组的最大不同是，切片的类型和长度信息无关，只要是相同类型元素构成的切片均对应相同的切片类型。

如前所述，切片是一种简化版的动态数组，这是切片类型的灵魂。除了构造切片和遍历切片之外，添加切片元素、删除切片元素都是切片处理中经常遇到的问题。

### 添加切片元素

内置的泛型函数 `append` 可以在切片的尾部追加 `N` 个元素：

```

var a []int
a = append(a, 1)           // 追加1个元素
a = append(a, 1, 2, 3)      // 追加多个元素，手写解包方式
a = append(a, []int{1,2,3}...) // 追加一个切片，切片需要解包

```

不过要注意的是，在容量不足的情况下，`append` 的操作会导致重新分配内存，从而导致巨大的内存分配和复制数据代价。即使容量足够，依然需要用 `append` 函数的返回值来更新切片本身，因为新切片的长度已经发生了变化。

除了在切片的尾部追加，我们还可以在切片的开头添加元素：

```

var a = []int{1,2,3}
a = append([]int{0}, a...)      // 在开头添加1个元素
a = append([]int{-3,-2,-1}, a...) // 在开头添加1个切片

```

在开头一般都会导致内存的重新分配，而且会导致已有的元素全部复制1次。因此，从切片的开头添加元素的性能一般要比从尾部追加元素的性能差很多。

由于 `append` 函数返回新的切片，也就是它支持链式操作。我们可以将多个 `append` 操作组合起来，实现在切片中间插入元素：

```

var a []int
a = append(a[:i], append([]int{x}, a[i:]...)...) // 在第i个位置插入x
a = append(a[:i], append([]int{1,2,3}, a[i:]...)...) // 在第i个位置插入切片

```

每个添加操作中的第二个 `append` 调用都会创建一个临时切片，并将 `a[i:]` 的内容复制到新创建的切片中，然后将临时创建的切片再追加到 `a[:i]`。

可以用 `copy` 和 `append` 组合可以避免创建中间的临时切片，同样是完成添加元素的操作：

```

a = append(a, 0) // 切片扩展1个空间
copy(a[i+1:], a[i:]) // a[i:]向后移动1个位置
a[i] = x // 设置新添加的元素

```

第一句 `append` 用于扩展切片的长度，为要插入的元素留出空间。第二句 `copy` 操作将要插入位置开始之后的元素向后挪动一个位置。第三句真实地将新添加的元素赋值到对应的位置。操作语句虽然冗长了一点，但是相比前面的方法，可以减少中间创建的临时切片。

用 `copy` 和 `append` 组合也可以实现在中间位置插入多个元素(也就是插入一个切片)：

```

a = append(a, x...) // 为x切片扩展足够的空间
copy(a[i+len(x):], a[i:]) // a[i:]向后移动len(x)个位置
copy(a[i:], x) // 复制新添加的切片

```

稍显不足的是，在第一句扩展切片容量的时候，扩展空间部分的元素复制是没有必要的。并没专门有内置的函数用于扩展切片的容量，`append` 本质是用于追加元素而不是扩展容量，扩展切片容量只是 `append` 的一个副作用。

### 删除切片元素

根据要删除元素的位置有三种类型：从开头位置删除，从中间位置删除，从尾部删除。其中删除切片尾部的元素最快：

```

a = []int{1, 2, 3}
a = a[:len(a)-1] // 删除尾部1个元素
a = a[:len(a)-N] // 删除尾部N个元素

```

删除开头的元素需要对剩余的元素进行一次整体挪动，可以用 `append` 原地完成（所谓原地完成是指在原有的切片数据对应的内存区间内完成，不会导致内存空间结构的变化）：

```

a = []int{1, 2, 3}
a = append(a[:0], a[1:]...) // 删除开头1个元素
a = append(a[:0], a[N:]...) // 删除开头N个元素

```

也可以用 `copy` 完成删除开头的元素：

```
a = []int{1, 2, 3}
a = a[:copy(a, a[1:])] // 删除开头1个元素
a = a[:copy(a, a[N:])] // 删除开头N个元素
```

对于删除中间的元素，需要对剩余的元素进行一次整体挪动，同样可以用 `append` 或 `copy` 原地完成：

```
a = []int{1, 2, 3, ...}

a = append(a[:i], a[i+1:]...) // 删除中间1个元素
a = append(a[:i], a[i+N:]...) // 删除中间N个元素

a = a[:copy(a[i:], a[i+1:])] // 删除中间1个元素
a = a[:copy(a[i:], a[i+N:])] // 删除中间N个元素
```

删除开头的元素和删除尾部的元素都可以认为是删除中间元素操作的特殊情况。

### 切片内存技巧

在本节开头的数组部分我们提到过有类似 `[0]int` 的空数组，空数组一般很少用到。但是对于切片来说，`len` 为 `0` 但是 `cap` 容量不为 `0` 的切片则是非常有用的特性。当然，如果 `len` 和 `cap` 都为 `0` 的话，则变成一个真正的空切片，虽然它并不是一个 `nil` 值的切片。在判断一个切片是否为空时，一般通过 `len` 获取切片的长度来判断，一般很少将切片和 `nil` 值做直接的比较。

比如下面的 `TrimSpace` 函数用于删除 `[]byte` 中的空格。函数实现利用了 `0` 长切片的特性，实现高效而且简洁。

```
func TrimSpace(s []byte) []byte {
    b := s[:0]
    for _, x := range s {
        if x != ' ' {
            b = append(b, x)
        }
    }
    return b
}
```

其实类似的根据过滤条件原地删除切片元素的算法都可以采用类似的方式处理（因为是删除操作不会出现内存不足的情形）：

```
func Filter(s []byte, fn func(x byte) bool) []byte {
    b := s[:0]
    for _, x := range s {
        if !fn(x) {
            b = append(b, x)
        }
    }
    return b
}
```

切片高效操作的要点是要降低内存分配的次数，尽量保证 `append` 操作不会超出 `cap` 的容量，降低触发内存分配的次数和每次分配内存大小。

### 避免切片内存泄漏

如前面所说，切片操作并不会复制底层的数据。底层的数组会被保存在内存中，直到它不再被引用。但是有时候可能会因为一个小小的内存引用而导致底层整个数组处于被使用的状态，这会延迟自动内存回收器对底层数组的回收。

例如，`FindPhoneNumber` 函数加载整个文件到内存，然后搜索第一个出现的电话号码，最后结果以切片方式返回。

```
func FindPhoneNumber(filename string) []byte {
    b, _ := ioutil.ReadFile(filename)
    return regexp.MustCompile("[0-9]+").Find(b)
}
```

这段代码返回的 `[]byte` 指向保存整个文件的数组。因为切片引用了整个原始数组，导致自动垃圾回收器不能及时释放底层数组的空间。一个小的需求可能导致需要长时间保存整个文件数据。这虽然这并不是传统意义上的内存泄漏，但是可能会拖慢系统的整体性能。

要修复这个问题，可以将感兴趣的数据复制到一个新的切片中（数据的传值是Go语言编程的一个哲学，虽然传值有一定的代价，但是换取好处是切断了对原始数据的依赖）：

```
func FindPhoneNumber(filename string) []byte {
    b, _ := ioutil.ReadFile(filename)
    b = regexp.MustCompile("[0-9]+").Find(b)
    return append([]byte{}, b...)
}
```

类似的问题，在删除切片元素时可能会遇到。假设切片里存放的是指针对象，那么下面删除末尾的元素后，被删除的元素依然被切片底层数组引用，从而导致不能及时被自动垃圾回收器回收（这要依赖回收器的实现方式）：

```
var a []*int{ ... }
a = a[:len(a)-1] // 本删除的最后一个元素依然被引用，可能导致GC操作被阻碍
```

保险的方式是先将需要自动内存回收的元素设置为 `nil`，保证自动回收器可以发现需要回收的对象，然后再进行切片的删除操作：

```
var a []*int{ ... }
a[len(a)-1] = nil // GC回收最后一个元素内存
a = a[:len(a)-1] // 从切片删除最后一个元素
```

当然，如果切片存在的周期很短的话，可以不用刻意处理这个问题。因为如果切片本身已经可以被GC回收的话，切片对应的每个元素自然也就是可以被回收的了。

### 切片类型强制转换

为了安全，当两个切片类型 `[]T` 和 `[]Y` 的底层原始切片类型不同时，Go语言是无法直接转换类型的。不过安全都是有一定代价的，有时候这种转换是有它的价值的——可以简化编码或者是提升代码的性能。比如在64位系统上，需要对一个 `[]float64` 切片进行高速排序，我们可以将它强制转为 `[]int` 整数切片，然后以整数的方式进行排序（因为 `float64` 遵循 IEEE754浮点数标准特性，当浮点数有序时对应的整数也必然是有序的）。

下面的代码通过两种方法将 `[]float64` 类型的切片转换为 `[]int` 类型的切片：

```
// +build amd64 arm64

import "sort"

var a = []float64{4, 2, 5, 7, 2, 1, 88, 1}

func SortFloat64FastV1(a []float64) {
    // 强制类型转换
    var b []int = ((*[1<<20]int)(unsafe.Pointer(&a[0])))[:len(a):cap(a)]

    // 以int方式给float64排序
    sort.Ints(b)
}

func SortFloat64FastV2(a []float64) {
    // 通过 reflect.SliceHeader 更新切片头部信息实现转换
    var c []int
    aHdr := (*reflect.SliceHeader)(unsafe.Pointer(&a))
    cHdr := (*reflect.SliceHeader)(unsafe.Pointer(&c))
    *cHdr = *aHdr

    // 以int方式给float64排序
    sort.Ints(c)
}
```

第一种强制转换是先将切片数据的开始地址转换为一个较大的数组的指针，然后对数组指针对应的数组重新做切片操作。中间需要 `unsafe.Pointer` 来连接两个不同类型的指针传递。需要注意的是，Go语言实现中非0大小数组的长度不得超过2GB，因此需要针对数组元素的类型大小计算数组的最大长度范围（`[]uint8` 最大2GB，`[]uint16` 最大1GB，以此类推，但是`[]struct{}` 数组的长度可以超过2GB）。

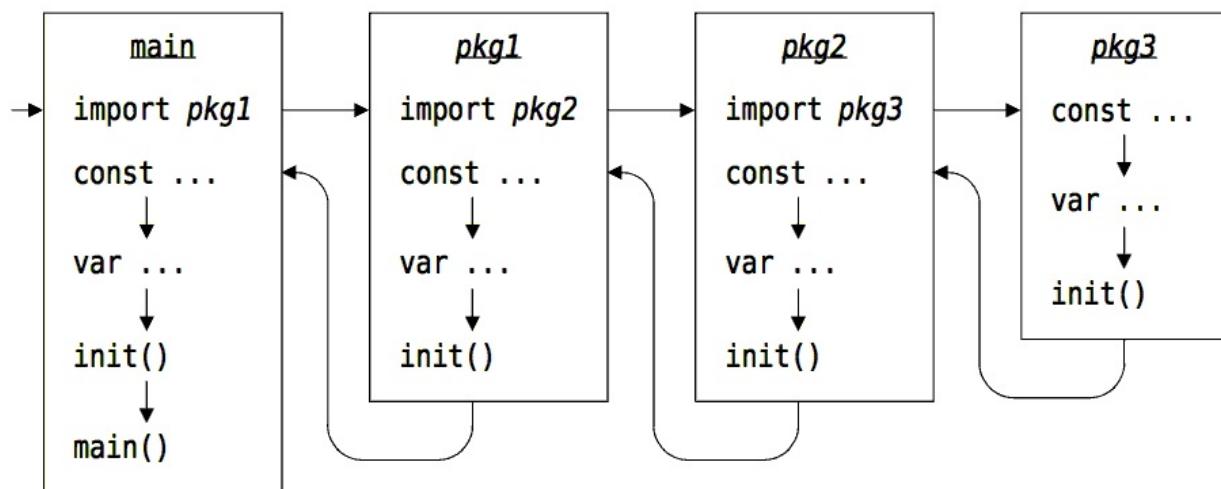
第二种转换操作是分别取到两个不同类型的切片头信息指针，任何类型的切片头部信息底层都是对应 `reflect.SliceHeader` 结构，然后通过更新结构体方式来更新切片信息，从而实现 `a` 对应的 `[]float64` 切片到 `c` 对应的 `[]int` 类型切片的转换。

通过基准测试，我们可以发现用 `sort.Ints` 对转换后的 `[]int` 排序的性能要比用 `sort.Float64s` 排序的性能好一点。不过需要注意的是，这个方法可行的前提是要保证 `[]float64` 中没有NaN和Inf等非规范的浮点数（因为浮点数中NaN不可排序，正0和负0相等，但是整数中没有这类情形）。

## 1.4. 函数、方法和接口

函数对应操作序列，是程序的基本组成元素。Go语言中的函数有具名和匿名之分：具名函数一般对应于包级的函数，是匿名函数的一种特例，当匿名函数引用了外部作用域中的变量时就成了闭包函数，闭包函数是函数式编程语言的核心。方法是绑定到一个具体类型的特殊函数，Go语言中的方法是依托于类型的，必须在编译时静态绑定。接口定义了方法的集合，这些方法依托于运行时的接口对象，因此接口对应的方法是在运行时动态绑定的。Go语言通过隐式接口机制实现了鸭子面向对象模型。

Go语言程序的初始化和执行总是从 `main.main` 函数开始的。但是如果 `main` 包导入了其它的包，则会按照顺序将它们包含进 `main` 包里（这里的导入顺序依赖具体实现，一般可能是以文件名或包路径名的字符串顺序导入）。如果某个包被多次导入的话，在执行的时候只会导入一次。当一个包被导入时，如果它还导入了其它的包，则先将其它的包包含进来，然后创建和初始化这个包的常量和变量，再调用包里的 `init` 函数，如果一个包有多个 `init` 函数的话，调用顺序未定义（实现可能是以文件名的顺序调用），同一个文件内的多个 `init` 则是以出现的顺序依次调用（`init` 不是普通函数，可以定义有多个，所有也不能被其它函数调用）。最后，当 `main` 包的所有包级常量、变量被创建和初始化完成，并且 `init` 函数被执行后，才会进入 `main.main` 函数，程序开始正常执行。下图是Go程序函数启动顺序的示意图：



要注意的是，在 `main.main` 函数执行之前所有代码都运行在同一个goroutine，也就是程序的主系统线程中。因此，如果某个 `init` 函数内部用 `go` 关键字启动了新的goroutine的话，新的goroutine只有在进入 `main.main` 函数之后才可能被执行到。

### 函数

在Go语言中，函数是第一类对象，我们可以将函数保持到变量中。函数主要有具名和匿名之分，包级函数一般都是具名函数，具名函数是匿名函数的一种特例。当然，Go语言中每个类型还可以有自己的方法，方法其实也是函数的一种。

```
// 具名函数
func Add(a, b int) int {
    return a+b
}

// 匿名函数
var Add = func(a, b int) int {
    return a+b
}
```

Go语言中的函数可以有多个输入参数和多个返回值，输入参数和返回值都是以传值的方式和被调用者交换数据。在语法上，函数还支持可变数量的参数，可变数量的参数必须是最后出现的参数，可变数量的参数其实是一个切片类型的参数。

```
// 多个输入参数和多个返回值
func Swap(a, b int) (int, int) {
    return b, a
}

// 可变数量的参数
// more 对应 []int 切片类型
func Sum(a int, more ...int) int {
    for _, v := range more {
        a += v
    }
    return a
}
```

当可变参数是一个空接口类型时，调用者是否解包可变参数会导致不同的结果：

```
func main() {
    var a = []interface{}{123, "abc"}

    Print(a...) // 123 abc
    Print(a)    // [123 abc]
}

func Print(a ...interface{}) {
    fmt.Println(a...)
}
```

第一个 `Print` 调用时传入的参数是 `a...`，等价于直接调用 `Print(123, "abc")`。第二个 `Print` 调用传入的是为解包的 `a`，等价于直接调用 `Print([]interface{}{123, "abc"})`。

不仅函数的输入参数可以有名字，也可以给函数的返回值命名：

```
func Find(m map[int]int, key int) (value int, ok bool) {
    return m[key]
}
```

如果返回值命名了，可以通过名字来修改返回值，也可以通过 `defer` 语句在 `return` 语句之后修改返回值：

```
func Inc() (v int) {
    defer func(){ v++ }()
    return 42
}
```

其中 `defer` 语句延迟执行了一个匿名函数，因为这个匿名函数捕获了外部函数的局部变量 `v`，这种函数我们一般叫闭包。闭包对捕获的外部变量并不是传值方式访问，而是以引用的方式访问。

闭包的这种引用方式访问外部变量的行为可能会导致一些隐含的问题：

```
func main() {
    for i := 0; i < 3; i++ {
        defer func(){ println(i) }()
    }
}
// Output:
// 3
// 3
// 3
```

因为是闭包，在 `for` 迭代语句中，每个 `defer` 语句延迟执行的函数引用的都是同一个 `i` 迭代变量，在循环结束后这个变量的值为3，因此最终输出的都是3。

修复的思路是在每轮迭代中为每个 `defer` 函数生成独有的变量。可以用下面两种方式：

```

func main() {
    for i := 0; i < 3; i++ {
        i := i // 定义一个循环体内局部变量i
        defer func(){ println(i) } ()
    }
}

func main() {
    for i := 0; i < 3; i++ {
        // 通过函数函数传入i
        // defer 语句会马上对调用参数求值
        defer func(i int){ println(i) } (i)
    }
}

```

第一种方法是在循环体内部再定义一个局部变量，这样每次迭代 `defer` 语句的闭包函数捕获的都是不同的变量，这些变量的值对应迭代时的值。第二种方式是将迭代变量通过闭包函数的参数传入，`defer` 语句会马上对调用参数求值。两种方式都是可以工作的。不过一般来说，在 `for` 循环内部执行 `defer` 语句并不是一个好的习惯，此处仅为示例，不建议使用。

Go语言中，如果以切片为参数调用函数时，有时候会给人一种参数采用了传引用的方式的假象：因为在被调用函数内部可以修改传入的切片的元素。其实，任何可以通过函数参数修改调用参数的情形，都是因为函数参数中显式或隐式传入了指针参数。函数参数传值的规范更准确说是只针对数据结构中固定的部分传值，例如字符串或切片对应结构体中的指针和字符串长度结构体传值，但是并不包含指针间接指向的内容。将切片类型的参数替换为类似 `reflect.SliceHeader` 结构体就很好理解切片传值的含义了：

```

func twice(x []int) {
    for i := range x {
        x[i] *= 2
    }
}

type IntSliceHeader struct {
    Data []int
    Len  int
    Cap  int
}

func twice(x IntSliceHeader) {
    for i := 0; i < x.Len; i++ {
        x.Data[i] *= 2
    }
}

```

因为切片中的底层数组部分是通过隐式指针传递(指针本身依然是传值的，但是指针指向的却是同一份的数据)，所以被调用函数是可以通过指针修改掉调用参数切片中的数据。除了数据之外，切片结构还包含了切片长度和切片容量信息，这2个信息也是传值的。如果被调用函数中修改了 `Len` 或 `cap` 信息的话，就无法反映到调用参数的切片中，这时候我们一般会通过返回修改后的切片来更新之前的切片。这也是为何内置的 `append` 必须要返回一个切片的原因。

Go语言中，函数还可以直接或间接地调用自己，也就是支持递归调用。Go语言函数的递归调用深度逻辑上没有限制，函数调用的栈是不会出现溢出错误的，因为Go语言运行时会根据需要动态地调整函数栈的大小。每个goroutine刚启动时只会分配很小的栈（4或8KB，具体依赖实现），根据需要动态调整栈的大小，栈最大可以达到GB级（依赖具体实现）。在Go1.4以前，Go的动力栈采用的是分段式的动力栈，通俗地说就是采用一个链表来实现动态栈，每个链表的节点内存位置不会发生变化。但是链表实现的动力栈对某些导致跨越链表不同节点的热点调用的性能影响较大，因为相邻的链表节点它们在内存位置一般不是相邻的，这会增加CPU高速缓存命中失败的几率。为了解决热点调用的CPU缓存命中率问题，Go1.4之后改用连续的动力栈实现，也就是采用一个类似动态数组的结构来表示栈。不过连续动力栈也带来了新的问题：当连续栈动态增长时，需要将之前的数据移动到新的内存空间，这会导致之前栈中全部变量的地址发生变化。虽然Go语言运行时会自动更新引用了地址变化的栈变量的指针，但最重要的一点是要明白Go语言中指针不再是固定不变的了（因此不能随意将指针保持到数值变量中，Go语言的地址也不能随意保存到不在GC控制的环境中，因此使用CGO时不能在C语言中长期持有Go语言对象的地址）。

因为，Go语言函数的栈不会溢出，所以普通Go程序员已经很少需要关心栈的运行机制的。在Go语言规范中甚至故意没有讲到栈和堆的概念。我们无法知道函数参数或局部变量到底是保存在栈中还是堆中，我们只需要知道它们能够正常工作就可以了。看看下面这个例子：

```
func f(x int) *int {
    return &x
}

func g() int {
    x = new(int)
    return *x
}
```

第一个函数直接返回了函数参数变量的地址——这似乎是不可以的，因为如果参数变量在栈上的话，函数返回之后栈变量就失效了，返回的地址自然也应该失效了。但是Go语言的编译器和运行时比我们聪明的多，它会保证指针指向的变量在合适的地方。第二个函数，内部虽然调用 `new` 函数创建了 `*int` 类型的指针对象，但是依然不知道它具体保存在哪里。对于有C/C++编程经验的程序员需要强调的是：不用关心Go语言中函数栈和堆的问题，编译器和运行时会帮我们搞定；同样不要假设变量在内存中的位置是固定不变的，指针随时可能会变化，特别是在你不期望它变化的时候。

# 方法

方法一般是面向对象编程(OOP)的一个特性，在C++语言中方法对应一个类对象的成员函数，是关联到具体对象上的虚表中的。但是Go语言的方法却是关联到类型的，这样可以在编译阶段完成方法的静态绑定。一个面向对象的程序会用方法来表达其属性和对应的操作，这样使用这个对象的用户就不需要直接去操作对象，而是借助方法来做这些事情。面向对象编程(OOP)进入主流开发领域一般认为是从C++开始的，C++就是在兼容C语言的基础之上支持了class等面向对象的特性。然后Java编程则号称是纯粹的面向对象语言，因为Java中函数是不能独立存在的，每个函数都必然是属于某个类的。

面向对象编程更多的只是一种思想，很多号称支持面向对象编程的语言只是将经常用到的特性内置到语言中了而已。Go语言的祖先C语言虽然不是一个支持面向对象的语言，但是C语言的标准库中的File相关的函数也用到了的面向对象编程的思想。下面我们实现一组C语言风格的File函数：

```
// 文件对象
type File struct {
    fd int
}

// 打开文件
func OpenFile(name string) (f *File, err error) {
    // ...
}

// 关闭文件
func CloseFile(f *File) error {
    // ...
}

// 读文件数据
func ReadFile(f *File, offset int64, data []byte) int {
    // ...
}
```

其中 `OpenFile` 类似构造函数用于打开文件对象，`CloseFile` 类似析构函数用于关闭文件对象，`ReadFile` 则类似普通的成员函数，这三个函数都是普通的函数。`CloseFile` 和 `ReadFile` 作为普通函数，需要占用包级空间中的名字资源。不过 `closeFile` 和 `ReadFile` 函数只是针对 `File` 类型对象的操作，这时候我们更希望这类函数和操作对象的类型紧密绑定在一起。

Go语言中的做法是，将 `closeFile` 和 `ReadFile` 函数的第一个参数移动到函数名的开头：

```
// 关闭文件
func (f *File) CloseFile() error {
    // ...
}

// 读文件数据
func (f *File) ReadFile(int64 offset, data []byte) int {
    // ...
}
```

这样的话，`CloseFile` 和 `ReadFile` 函数就成了 `File` 类型独有的方法了（而不是 `File` 对象方法）。它们也不再占用包级空间中的名字资源，同时 `File` 类型已经明确了它们操作对象，因此方法名字一般简化为 `Close` 和 `Read`：

```
// 关闭文件
func (f *File) Close() error {
    // ...
}

// 读文件数据
func (f *File) Read(int64 offset, data []byte) int {
    // ...
}
```

将第一个函数参数移动到函数前面，从代码角度看虽然只是一个小的改动，但是从编程哲学角度来看，Go语言已经是进入面向对象语言的行列了。我们可以给任何自定义类型添加一个或多个方法。每种类型对应的方法必须和类型的定义在同一个包中，因此是无法给 `int` 这类内置类型添加方法的（因为方法的定义和类型的定义不在一个包中）。对于给定的类型，每个方法的名字必须是唯一的，同时方法和函数一样也不支持重载。

方法是由函数演变而来，只是将函数的第一个对象参数移动到了函数名前面了而已。因此我们依然可以按照原始的过程式思维来使用方法。通过叫方法表达式的特性可以将方法还原为普通类型的函数：

```
// 不依赖具体的文件对象
// func CloseFile(f *File) error
var CloseFile = (*File).Close

// 不依赖具体的文件对象
// func ReadFile(f *File, int64 offset, data []byte) int
var ReadFile = (*File).Read

// 文件处理
f, _ := OpenFile("foo.dat")
ReadFile(f, 0, data)
CloseFile(f)
```

在有些场景更关心一组相似的操作：比如 `Read` 读取一些数组，然后调用 `Close` 关闭。此时的环境中，用户并不关心操作对象的类型，只要能满足通用的 `Read` 和 `Close` 行为就可以了。不过在方法表达式中，因为得到的 `ReadFile` 和 `CloseFile` 函数参数中含有 `File` 这个特有的类型参数，这使得 `File` 相关的方法无法和其它不是 `File` 类型但是有着相同 `Read` 和 `Close` 方法的对象无缝适配。这种小困难难不倒我们 Go 语言码农，我们可以通过结合闭包特性来消除方法表达式中第一个参数类型的差异：

```
// 先打开文件对象
f, _ := OpenFile("foo.dat")

// 绑定到了 f 对象
// func Close() error
var Close = func Close() error {
    return (*File).Close(f)
}

// 绑定到了 f 对象
// func Read(int64 offset, data []byte) int
var Read = func Read(int64 offset, data []byte) int {
    return (*File).Read(f, offset, data)
}

// 文件处理
Read(0, data)
Close()
```

这刚好是方法值也要解决的问题。我们用方法值特性可以简化实现：

```
// 先打开文件对象
f, _ := OpenFile("foo.dat")

// 方法值：绑定到了 f 对象
// func Close() error
var Close = f.Close

// 方法值：绑定到了 f 对象
// func Read(int64 offset, data []byte) int
var Read = f.Read

// 文件处理
Read(0, data)
Close()
```

Go 语言不仅支持传统面向对象中的继承特性，而是以自己特有的组合方式支持了方法的继承。Go 语言中，通过在结构体内部置匿名的成员来实现继承：

```

import "image/color"

type Point struct{ X, Y float64 }

type ColoredPoint struct {
    Point
    Color color.RGBA
}

```

虽然我们可以将 `ColoredPoint` 定义为一个有三个字段的扁平结构的结构体，但是我们这里将 `Point` 嵌入到 `ColoredPoint` 来提供 `X` 和 `Y` 这两个字段。

```

var cp ColoredPoint
cp.X = 1
fmt.Println(cp.Point.X) // "1"
cp.Point.Y = 2
fmt.Println(cp.Y)       // "2"

```

通过嵌入匿名的成员，我们不仅可以继承匿名成员的内部成员，而且可以继承匿名成员类型所对应的方法。我们一般会将 `Point` 看作基类，把 `ColoredPoint` 看作是它的继承类或子类。不过这种方式继承的方法并不能实现 C++ 中虚函数的多态特性。所有继承来的方法的接收者参数依然是那个匿名成员本身，而不是当前的变量。

```

type Cache struct {
    m map[string]string
    sync.Mutex
}

func (p *Cache) Lookup(key string) string {
    p.Lock()
    defer p.Unlock()

    return p.m[key]
}

```

`Cache` 结构体类型通过嵌入一个匿名的 `sync.Mutex` 来继承它的 `Lock` 和 `Unlock` 方法。但是在调用 `p.Lock()` 和 `p.Unlock()` 时，`p` 并不是 `Lock` 和 `Unlock` 方法的真正接收者，而是会将它们展开为 `p.Mutex.Lock()` 和 `p.Mutex.Unlock()` 调用。这种展开是编译期完成的，并没有运行时代价。

在传统的面向对象语言(eg.C++或Java)的继承中，子类的方法是在运行时动态绑定到对象的，因此基类实现的某些方法看到的 `this` 可能不是基类型对应的对象，这个特性会导致基类方法运行的不确定性。而在 Go 语言通过嵌入匿名的成员来“继承”的基类方法，`this` 就是实现该方法的类型的对象，Go 语言中方法是编译时静态绑定的。如果需要虚函数的多态特性，我们需要借助 Go 语言接口来实现。

# 接口

Go语言之父Rob Pike曾说过一句名言：那些试图避免白痴行为的语言最终自己变成了白痴语言（Languages that try to disallow idiocy become themselves idiotic）。一般静态编程语言都有着严格的类型系统，这使得编译器可以深入检查程序员没有作出什么出格的举动。但是，过于严格的类型系统却会使得编程太过繁琐，让程序员把大好的青春都浪费在了和编译器的斗争中。Go语言试图让程序员能在安全和灵活的编程之间取得一个平衡。它在提供严格的数据类型检查的同时，通过接口类型实现了对鸭子类型的支持，使得安全动态的编程变得相对容易。

Go的接口类型是对其它类型行为的抽象和概括；因为接口类型不会和特定的实现细节绑定在一起，通过这种抽象的方式我们可以让对象更加灵活和更具有适应能力。很多面向对象的语言都有相似的接口概念，但Go语言中接口类型的独特之处在于它是满足隐式实现的鸭子类型。所谓鸭子类型说的是：只要走起路来像鸭子、叫起来也像鸭子，那么就可以把它当作鸭子。Go语言中的面向对象就是如此，如果一个对象只要看起来像是某种接口类型的实现，那么它就可以作为该接口类型使用。这种设计可以让你创建一个新的接口类型满足已经存在的具体类型却不用去破坏这些类型原有的定义；当我们使用的类型来自于不受我们控制的包时这种设计尤其灵活有用。Go语言的接口类型是延迟绑定，可以实现类似虚函数的多态功能。

接口在Go语言中无处不在，在“Hello world”的例子中，`fmt.Printf` 函数的设计就是完全基于接口的，它的真正功能由 `fmt.Fprintf` 函数完成。用于表示错误的 `error` 类型更是内置的接口类型。在C语言中，`printf` 只能将几种有限的基础数据类型打印到文件对象中。但是Go语言灵活接口特性，`fmt.Fprintf` 却可以向任何自定义的输出流对象打印，可以打印到文件或标准输出、也可以打印到网络、甚至可以打印到一个压缩文件；同时，打印的数据也不仅仅局限于语言内置的基础类型，任意隐式满足 `fmt.Stringer` 接口的对象都可以打印，不满足 `fmt.Stringer` 接口的依然可以通过反射的技术打印。`fmt.Fprintf` 函数的签名如下：

```
func Fprintf(w io.Writer, format string, args ...interface{}) (int, error)
```

其中 `io.Writer` 用于输出的接口，`error` 是内置的错误接口，它们的定义如下：

```
type io.Writer interface {
    Write(p []byte) (n int, err error)
}

type error interface {
    Error() string
}
```

我们可以通过定制自己的输出对象，将每个字符转为大写字符后输出：

```

type UpperWriter struct {
    io.Writer
}

func (p *UpperWriter) Write(data []byte) (n int, err error) {
    return p.Writer(bytes.ToUpper(data))
}

func main() {
    fmt.Fprintln(&UpperWriter{os.Stdout}, "hello, world")
}

```

当然，我们也可以定义自己的打印格式来实现将每个字符转为大写字符后输出的效果。对于每个要打印的对象，如果满足了 `fmt.Stringer` 接口，则默认使用对象的 `String` 方法返回的结果打印：

```

type UpperString string

func (s UpperString) String() string {
    return strings.ToUpper(s)
}

type fmt.Stringer interface {
    String() string
}

func main() {
    fmt.Fprintln(os.Stdout, UpperString("hello, world"))
}

```

Go语言中，对于基础类型（非接口类型）不支持隐式的转换，我们无法将一个 `int` 类型的值直接赋值给 `int64` 类型的变量，也无法将 `int` 类型的值赋值给底层是 `int` 类型的新定义命名类型的变量。Go语言对基础类型的类型一致性要求可谓是非常的严格，但是Go语言对于接口类型的转换则非常的灵活。对象和接口之间的转换、接口和接口之间的转换都可能是隐式的转换。可以看下面的例子：

```

var (
    a io.ReadCloser = (*os.File)(f) // 隐式转换, *os.File 类型满足了 io.ReadCloser 接口
    b io.Reader     = a             // 隐式转换, io.ReadCloser 满足了 io.Reader 接口
    c io.Closer     = a             // 隐式转换, io.ReadCloser 满足了 io.Closer 接口
    d io.Reader     = c.(io.Reader) // 显式转换, io.Closer 并不显式满足 io.Reader 接口
)

```

有时候对象和接口之间太灵活了，导致我们需要人为地限制这种无意之间的适配。常见的做法是定义一个含特殊方法来区分接口。比如 `runtime` 包中的 `Error` 接口就定义了一个特有的 `RuntimeError` 方法，用于避免其它类型无意中适配了该接口：

```

type runtime.Error interface {
    error

    // RuntimeError is a no-op function but
    // serves to distinguish types that are run time
    // errors from ordinary errors: a type is a
    // run time error if it has a RuntimeError method.
    RuntimeError()
}

```

在protobuf中，`Message` 接口也采用了类似的方法，也定义了一个特有的 `ProtoMessage`，用于避免其它类型无意中适配了该接口：

```

type proto.Message interface {
    Reset()
    String() string
    ProtoMessage()
}

```

不过这种做法只是君子协定，如果有人刻意伪造一个 `proto.Message` 接口也是很容易的。再严格一点的做法是给接口定义一个私有方法。只有满足了这个私有方法的对象才可能满足这个接口，而私有方法的名字是包含包的绝对路径名的，因此只能在包内部实现这个私有方法才能满足这个接口。测试包中的 `testing.PB` 接口就是采用类似的技术：

```

type testing.TB interface {
    Error(args ...interface{})
    Errorf(format string, args ...interface{})
    ...

    // A private method to prevent users implementing the
    // interface and so future additions to it will not
    // violate Go 1 compatibility.
    private()
}

```

不过这种通过私有方法禁止外部对象实现接口的做法也是有代价的：首先是这个接口只能包内部使用，外部包正常情况下是无法直接创建满足该接口对象的；其次，这种防护措施也不是绝对的，恶意的用户依然可以绕过这种保护机制。

在前面的方法一节中我们讲到，通过在结构体中嵌入匿名类型成员，可以继承匿名类型的方法。其实这个被嵌入的匿名成员不一定是普通类型，也可以是接口类型。我们可以通过嵌入匿名的 `testing.PB` 接口来伪造私有的 `private` 方法，因为接口方法是延迟绑定，编译时 `private` 方法是否真的存在并不重要。

```

package main

import (
    "fmt"
    "testing"
)

type TB struct {
    testing.TB
}

func (p *TB) Fatal(args ...interface{}) {
    fmt.Println("TB.Fatal disabled!")
}

func main() {
    var tb testing.TB = new(TB)
    tb.Fatal("Hello, playground")
}

```

我们在自己的 `PB` 结构体类型中重新实现了 `Fatal` 方法，然后通过将对象隐式转换为 `testing.TB` 接口类型（因为内嵌了匿名的 `testing.TB` 对象，因此是满足 `testing.TB` 接口的），然后通过 `testing.TB` 接口来调用我们自己的 `Fatal` 方法。

这种通过嵌入匿名接口或嵌入匿名指针对象来实现继承的做法其实是一种纯虚继承，我们继承的只是接口指定的规范，真正的实现在运行的时候才被注入。比如，我们可以模拟实现一个`grpc`的插件：

```

type grpcPlugin struct {
    *generator.Generator
}

func (p *grpcPlugin) Name() string { return "grpc" }

func (p *grpcPlugin) Init(g *generator.Generator) {
    p.Generator = g
}

func (p *grpcPlugin) GenerateImports(file *generator.FileDescriptor) {
    if len(file.Service) == 0 {
        return
    }

    p.P(`import "google.golang.org/grpc"`)
    // ...
}

```

构造的 `grpcPlugin` 类型对象必须满足 `generate.Plugin` 接口  
(在 "github.com/golang/protobuf/protoc-gen-go/generator" 包中) :

```
type Plugin interface {
    // Name identifies the plugin.
    Name() string
    // Init is called once after data structures are built but before
    // code generation begins.
    Init(g *Generator)
    // Generate produces the code generated by the plugin for this file,
    // except for the imports, by calling the generator's methods P, In, and Out.
    Generate(file *FileDescriptor)
    // GenerateImports produces the import declarations for this file.
    // It is called after Generate.
    GenerateImports(file *FileDescriptor)
}
```

`generate.Plugin` 接口对应的 `grpcPlugin` 类型的 `GenerateImports` 方法中使用的 `p.P(...)` 函数却是通过 `Init` 函数注入的 `generator.Generator` 对象实现。这里的 `generator.Generator` 对应一个具体类型，但是如果 `generator.Generator` 是接口类型的话我们甚至可以传入直接的实现。

Go 语言通过几种简单特性的组合，就轻易就实现了鸭子面向对象和虚拟继承等高级特性，真的是不可思议。

## 1.5. 面向并发的内存模型

在早期，CPU都是以单核的形式顺序执行机器指令。Go语言的祖先C语言正是这种顺序编程语言的代表。顺序编程语言中的顺序是指：所有的指令都是以串行的方式执行，在相同的时刻有且仅有一个CPU在顺序执行程序的指令。

随着处理器技术的发展，单核时代以提升处理器频率来提高运行效率的方式遇到了瓶颈，目前各种主流的CPU频率基本被锁定在了3GHZ附近。单核CPU的发展的停滞，给多核CPU的发展带来了机遇。相应地，编程语言也开始逐步向并行化的方向发展。Go语言正是在多核和网络化的时代背景下诞生的原生支持并发的编程语言。

常见的并行编程有多种模型，主要有线程、消息传递等。从理论上来看，多线程和基于消息的并发编程是等价的。由于多线程并发模型可以自然对应到多核的处理器，主流的操作系统因此也都提供了系统级的多线程支持，同时从概念上讲多线程似乎也更直观，因此多线程编程模型逐步被吸纳到主流的编程语言特性或语言扩展库中。而主流编程语言对基于消息的并发编程模型支持则相比较少，Erlang语言是支持基于消息传递并发编程模型的代表者，它的并发体之间不共享内存。Go语言是基于消息并发模型的集大成者，它将基于CSP模型的并发编程内置到了语言中，通过一个go关键字就可以轻易地启动一个Goroutine，与Erlang不同的是Go语言的Goroutine之间是共享内存的。

## Goroutine和系统线程

Goroutine是Go语言特有的并发体，是一种轻量级的线程，由go关键字启动。在真实的Go语言的实现中，goroutine和系统线程也不是等价的。尽管两者的区别实际上只是一个量的区别，但正是这个量变引发了Go语言并发编程质的飞跃。

首先，每个系统级线程都会有一个固定大小的栈（一般默认可能是2MB），这个栈主要用来保存函数递归调用时参数和局部变量。固定了栈的大小这导致了两个问题：一是对于很多只需要很小的栈空间的线程来说是一个巨大的浪费，二是对于少数需要巨大栈空间的线程来说又面临栈溢出的风险。针对这两个问题的解决方案是：要么降低固定的栈大小，提升空间的利用率；要么增大栈的深度以允许更深的函数递归调用，但这两者是没法同时兼得的。相反，一个Goroutine会以一个很小的栈启动（可能是2KB或4KB），当遇到深度递归导致当前栈空间不足时，Goroutine会根据需要动态地伸缩栈的大小（主流实现中栈的最大值可达到1GB）。因为启动的代价很小，所以我们可以轻易地启动成千上万个Goroutine。

Go的运行时还包含了其自己的调度器，这个调度器使用了一些技术手段，可以在n个操作系统线程上多工调度m个Goroutine。Go调度器的工作和内核的调度是相似的，但是这个调度器只关注单独的Go程序中的Goroutine。Goroutine采用的是半抢占式的协作调度，只有在当前

Goroutine发生阻塞时才会导致调度；同时发生在用户态，调度器会根据具体函数只保存必要的寄存器，切换的代价要比系统线程低得多。运行时有一个 `runtime.GOMAXPROCS` 变量，用于控制当前运行正常非阻塞Goroutine的系统线程数目。

在Go语言中启动一个Goroutine不仅和调用函数一样简单，而且Goroutine之间调度代价也很低，这些因素极大地促进了并发编程的流行和发展。

## 原子操作

所谓的原子操作就是并发编程中“最小的且不可并行化”的操作。通常，有多个并发体对一个共享资源的操作是原子操作的话，同一时刻最多只能有一个并发体对该资源进行操作。从线程角度看，在当前线程修改共享资源期间，其它的线程是不能访问该资源的。原子操作对于多线程并发编程模型来说，不会发生有别于单线程的意外情况，共享资源的完整性可以得到保证。

一般情况下，原子操作都是通过“互斥”访问来保证访问的，通常由特殊的CPU指令提供保护。当然，如果仅仅是想模拟下粗粒度的原子操作，我们可以借助于 `sync.Mutex` 来实现：

```
import (
    "sync"
)

var total struct {
    sync.Mutex
    value int
}

func worker(wg *sync.WaitGroup) {
    defer wg.Done()

    for i := 0; i <= 100; i++ {
        total.Lock()
        total.value += i
        total.Unlock()
    }
}

func main() {
    go worker(&wg)
    go worker(&wg)
    wg.Wait()

    fmt.Println(total.value)
}
```

在 `worker` 的循环中，为了保证 `total.value += i` 的原子性，我们通过 `sync.Mutex` 加锁和解锁来保证该语句在同一时刻只被一个线程访问。对于多线程模型的程序而言，进出临界区前后进行加锁和解锁都是必须的。如果没有锁的保护，`total` 的最终值将由于多线程之间的竞争而可能会不正确。

用互斥锁来保护一个数值型的共享资源，麻烦且效率低下。标准库的 `sync/atomic` 包对原子操作提供了丰富的支持。我们可以重新实现上面的例子：

```
import (
    "sync"
    "sync/atomic"
)

var total uint64

func worker(wg *sync.WaitGroup) {
    defer wg.Done()

    var i uint64
    for i = 0; i <= 100; i++ {
        atomic.AddUint64(&total, i)
    }
}

func main() {
    var wg sync.WaitGroup
    wg.Add(2)

    go worker(&wg)
    go worker(&wg)
    wg.Wait()
}
```

`atomic.AddUint64` 函数调用保证了 `total` 的读取、更新和保存是一个原子操作，因此在多线程中访问也是安全的。

原子操作配合互斥锁可以实现非常高效的单件模式。互斥锁的代价比普通整数的原子读写高很多，在性能敏感的地方可以增加一个数字型的标志位，通过原子检测标志位状态通过降低互斥锁的次数来提高性能。

```

type singleton struct {}

var (
    instance    *singleton
    initialized uint32
    mu          sync.Mutex
)

func Instance() *singleton {
    if atomic.LoadUint32(&initialized) == 1 {
        return instance
    }

    mu.Lock()
    defer mu.Unlock()

    if instance == nil {
        defer atomic.StoreUint32(&initialized, 1)
        instance = &singleton{}
    }
    return instance
}

```

我们可以将通用的代码提取出来，就成了标准库中 `sync.Once` 的实现：

```

type Once struct {
    m    Mutex
    done uint32
}

func (o *Once) Do(f func()) {
    if atomic.LoadUint32(&o.done) == 1 {
        return
    }

    o.m.Lock()
    defer o.m.Unlock()

    if o.done == 0 {
        defer atomic.StoreUint32(&o.done, 1)
        f()
    }
}

```

基于 `sync.Once` 重新实现单件模式：

```

var (
    instance *singleton
    once     sync.Once
)

func Instance() *singleton {
    once.Do(func() {
        instance = &singleton{}
    })
    return instance
}

```

`sync/atomic` 包对基本的数值类型及复杂对象的读写都提供了原子操作的支持。`atomic.Value` 原子对象提供了 `Load` 和 `Store` 两个原子方法，分别用于加载和保存数据，返回值和参数都是 `interface{}` 类型，因此可以用于任意的自定义复杂类型。

```

var config atomic.Value // 保存当前配置信息

// 初始化配置信息
config.Store(loadConfig())

// 启动一个后台线程，加载更新后的配置信息
go func() {
    for {
        time.Sleep(time.Second)
        config.Store(loadConfig())
    }
}()

// 用于处理请求的工作者线程始终采用最新的配置信息
for i := 0; i < 10; i++ {
    go func() {
        for r := range requests() {
            c := config.Load()
            // ...
        }
    }()
}

```

这是一个简化的生产者、消费者模型：后台线程生成最新的配置信息；前台多个工作者线程获取最新的配置信息。所有线程共享配置信息资源。

## 顺序一致性内存模型

如果只是想简单地在线程之间进行数据同步的话，原子操作已经为编程人员提供了一些同步保障。不过这种保障有一个前提：顺序一致性的内存模型。要了解顺序一致性，我们先看看一个简单的例子：

```
var a string
var done bool

func setup() {
    a = "hello, world"
    done = true
}

func main() {
    go setup()
    for !done {} 
    print(a)
}
```

我们创建了 `setup` 线程，用于对字符串 `a` 的初始化工作，初始化完成之后设置 `done` 标志为 `true`。`main` 函数所在的主线程中，通过 `for !done {}` 检测 `done` 变为 `true` 时，认为字符串初始化工作完成，然后进行字符串的打印工作。

但是Go语言并不保证在 `main` 函数中观测到的对 `done` 的写入操作发生在对字符串 `a` 的写入的操作之后，因此程序很可能打印一个空字符串。更糟糕的是，因为两个线程之间没有同步事件，`setup` 线程对 `done` 的写入操作甚至无法被 `main` 线程看到，`main` 函数有可能陷入死循环中。

在Go语言中，同一个Goroutine线程内部，顺序一致性内存模型是得到保证的。但是不同的Goroutine之间，并不满足顺序一致性内存模型，需要通过明确定义的同步事件来作为同步的参考。如果两个事件不可排序，那么就说这两个事件是并发的。为了最大化并行，Go语言的编译器和处理器在不影响上述规定的前提下可能会对执行语句重新排序（CPU也会对一些指令进行乱序执行）。

因此，如果在一个Goroutine中顺序执行 `a = 1; b = 2;` 两个语句，虽然在当前的Goroutine中可以认为 `a = 1;` 语句先于 `b = 2;` 语句执行，但是在另一个Goroutine中 `b = 2;` 语句可能会先于 `a = 1;` 语句执行，甚至在另一个Goroutine中无法看到它们的变化（可能始终在寄存器中）。也就是说在另一个Goroutine看来，`a = 1; b = 2;` 两个语句的执行顺序是不确定的。如果一个并发程序无法确定事件的偏序关系，那么程序的运行结果往往会有不确定的结果。比如下面这个程序：

```
func main() {
    go println("你好, 世界")
}
```

根据Go语言规范，`main` 函数退出时程序结束，不会等待任何后台线程。因为Goroutine的执行和`main` 函数的返回事件是并发的，谁都有可能先发生，所以什么时候打印，能否打印都是未知的。

用前面的原子操作并不能解决问题，因为我们无法确定两个原子操作之间的顺序。解决问题的办法就是通过同步原语来给两个事件明确排序：

```
func main() {
    done := make(chan int)

    go func(){
        println("你好, 世界")
        done <- 1
    }()

    <-done
}
```

当`<-done` 执行时，必然要求`done <- 1` 也已经执行。根据同一个Goroutine依然满足顺序一致性规则，我们可以判断当`done <- 1` 执行时，`println("你好, 世界")` 语句必然已经执行完成了。因此，现在的程序确保可以正常打印结果。

当然，通过`sync.Mutex` 互斥量也是可以实现同步的：

```
func main() {
    var mu sync.Mutex

    mu.Lock()
    go func(){
        println("你好, 世界")
        mu.Unlock()
    }()

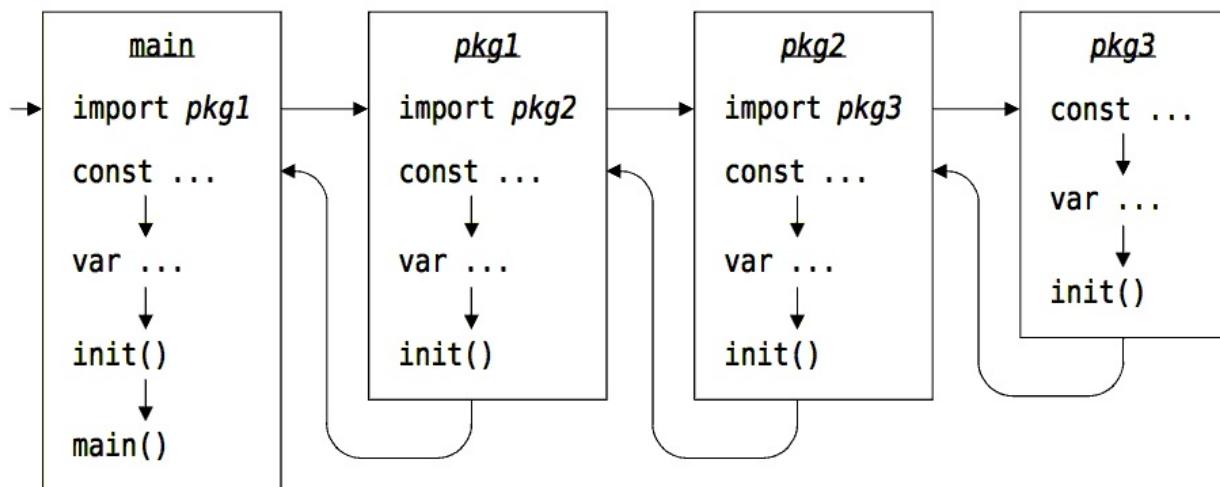
    mu.Lock()
}
```

可以确定后台线程的`mu.Unlock()` 必然在`println("你好, 世界")` 完成后发生（同一个线程满足顺序一致性），`main` 函数的第二个`mu.Lock()` 必然在后台线程的`mu.Unlock()` 之后发生（`sync.Mutex` 保证），此时后台线程的打印工作已经顺利完成了。

## 初始化顺序

前面函数章节中我们已经简单介绍过程序的初始化顺序，这是属于Go语言面向并发的内存模型的基础规范。

Go程序的初始化和执行总是从 `main.main` 函数开始的。但是如果 `main` 包里导入了其它的包，则会按照顺序将它们包含进 `main` 包里（这里的导入顺序依赖具体实现，一般可能是以文件名或包路径名的字符串顺序导入）。如果某个包被多次导入的话，在执行的时候只会导入一次。当一个包被导入时，如果它还导入了其它的包，则先将其它的包包含进来，然后创建和初始化这个包的常量和变量。然后就是调用包里的 `init` 函数，如果一个包有多个 `init` 函数的话，实现可能是以文件名的顺序调用，同一个文件内的多个 `init` 则是以出现的顺序依次调用（`init` 不是普通函数，可以定义有多个，所以不能被其它函数调用）。最终，在 `main` 包的所有包常量、包变量被创建和初始化，并且 `init` 函数被执行后，才会进入 `main.main` 函数，程序开始正常执行。下图是Go程序函数启动顺序的示意图：



要注意的是，在 `main.main` 函数执行之前所有代码都运行在同一个goroutine中，也是运行在程序的主系统线程中。如果某个 `init` 函数内部用 `go` 关键字启动了新的goroutine的话，新的goroutine只有在进入 `main.main` 函数之后才可能被执行到。

因为所有的 `init` 函数和 `main` 函数都是在主线程完成，它们也是满足顺序一致性模型的。

## Goroutine的创建

`go` 语句会在当前Goroutine对应函数开始执行前启动新的Goroutine. 例如:

```

var a string

func f() {
    print(a)
}

func hello() {
    a = "hello, world"
    go f()
}

```

执行 `go f()` 语句创建Goroutine和 `hello` 函数是在同一个Goroutine中执行，根据语句的书写顺序可以确定Goroutine的创建发生在 `hello` 函数返回之前，但是新创建Goroutine对应的 `f()` 的执行事件和 `hello` 函数返回的事件则是不可排序的，也就是并发的。调用 `hello` 可能在将来的某一时刻打印 `"hello, world"`，也很可能是在 `hello` 函数执行完成后才打印。

## 基于**Channel**的通信

**Channel**通信是在Goroutine之间进行同步的主要方法。在无缓存的**Channel**上的每一次发送操作都有与其对应的接收操作相配对，发送和接收操作通常发生在不同的Goroutine上（在同一个Goroutine上执行2个操作很容易导致死锁）。无缓存的**Channel**上的发送操作总在对应的接收操作完成前发生。

```
var done = make(chan bool)
var msg string

func aGoroutine() {
    msg = "你好，世界"
    done <- true
}

func main() {
    go aGoroutine()
    <-done
    println(msg)
}
```

可保证打印出“hello, world”。该程序首先对 `msg` 进行写入，然后在 `done` 管道上发送同步信号，随后从 `done` 接收对应的同步信号，最后执行 `println` 函数。

若在关闭信道后继续从中接收数据，接收者就会收到该信道返回的零值。因此在这个例子中，用 `close(c)` 关闭管道代替 `done <- false` 依然能保证该程序产生相同的行为。

```
var done = make(chan bool)
var msg string

func aGoroutine() {
    msg = "你好，世界"
    close(done)
}

func main() {
    go aGoroutine()
    <-done
    println(msg)
}
```

对于从无缓冲信道进行的接收，发生在对该信道进行的发送完成之前。

基于上面这个规则可知，交换两个Goroutine中的接收和发送操作也是可以的（但是很危险）：

```
var done = make(chan bool)
var msg string

func aGoroutine() {
    msg = "hello, world"
    <-done
}

func main() {
    go aGoroutine()
    done <- true
    println(msg)
}
```

也可保证打印出“hello, world”。后台线程首先对 `msg` 进行写入，然后从 `done` 中接收信号，随后 `main` 线程向 `done` 发送对应的信号，最后执行 `println` 函数完成。但是，若该信道为带缓冲的（例如，`done = make(chan bool, 1)`），`main` 线程的 `done <- true` 接收操作将不会被后台线程的 `<-done` 接收操作阻塞，该程序将无法保证打印出“hello, world”。

对于带缓冲的Channel，对于**Channel**的第 `k` 个接收完成操作发生在第 `k+c` 个发送操作完成之前，其中 `c` 是**Channel**的缓存大小。如果将 `c` 设置为0自然就对应无缓存的**Channel**，也即使第K个接收完成在第K个发送完成之前。因为无缓存的**Channel**只能同步发1个，也就简化为前面无缓存**Channel**的规则：对于从无缓冲信道进行的接收，发生在对该信道进行的发送完成之前。

我们可以根据控制**Channel**的缓存大小来控制并发执行的Goroutine的最大数目，例如：

```
var limit = make(chan int, 3)

func main() {
    for _, w := range work {
        go func() {
            limit <- 1
            w()
            <-limit
        }()
    }
    select{}
}
```

最后一句 `select{}` 是一个空的管道选择语句，该语句会导致 `main` 线程阻塞，从而避免程序过早退出。还有 `for{}<-make(chan int)` 等诸多方法可以达到类似的效果。因为 `main` 线程被阻塞了，如果需要程序正常退出的话可以通过调用 `os.Exit(0)` 实现。

## 不靠谱的同步

前面我们已经分析过，下面代码无法保证正常打印结果。实际的运行效果也是大概率不能正常输出结果。

```
func main() {
    go println("你好, 世界")
}
```

刚接触Go语言的话，可能希望通过加入一个随机的休眠时间来保证正常的输出：

```
func main() {
    go println("hello, world")
    time.Sleep(time.Second)
}
```

因为主线程休眠了1秒钟，因此这个程序大概率是可以正常输出结果的。因此，很多人会觉得这个程序已经没有问题了。但是这个程序是不稳健的，依然有失败的可能性。我们先假设程序是可以稳定输出结果的。因为Go线程的启动是非阻塞的，`main`线程显式休眠了1秒钟退出导致程序结束，我们可以近似地认为程序总共执行了1秒多时间。现在假设`println`函数内部实现休眠的时间大于`main`线程休眠的时间的话，就会导致矛盾：后台线程既然先于`main`线程完成打印，那么执行时间肯定是小于`main`线程执行时间的。当然这是不可能的。

严谨的并发程序的正确性不应该是依赖于CPU的执行速度和休眠时间等不靠谱的因素的。严谨的并发也应该是可以静态推导出结果的：根据线程内顺序一致性，结合Channel或sync同步事件的可排序性来推导，最终完成各个线程各段代码的偏序关系排序。如果两个事件无法根据此规则来排序，那么它们就是并发的，也就是执行先后顺序不可靠的。

解决同步问题的思路是相同的：使用显式的同步。

## 1.6. 常见的并发模式

Go语言最吸引人的地方是它内建的并发支持。Go语言并发体系的理论是C.A.R Hoare在1978年提出的CSP（Communicating Sequential Process，通讯顺序进程）。CSP有着精确的数学模型，并实际应用在了Hoare实际参与设计的T9000通用计算机上。从NewSqueak、Alef、Limbo到现在的Go语言，对于对CSP有着20多年实战经验的Rob Pike来说，他更关注的是将CSP应用在通用编程语言上的潜力。作为Go并发编程核心的CSP理论的核心概念只有一个：同步通信。关于同步通信的话题我们在前面一节已经讲过，本节我们将简单介绍下Go语言中常见的并发模式。

首先要明确一个概念：并发不是并行。并发更关注的是程序的设计层面，并发的程序完全是可以顺序执行的，只有在真正的多核CPU上才可能真正地同时运行。并行更关注的是程序的运行层面，并行一般是简单的大量重复，例如GPU中对图像处理都会有大量的并行运算。Go语言从一开始设计，就围绕着如何能在编程语言的层级，为更好的编写并发程序设计一个简洁安全高效的抽象模型，让程序员专注于分解问题和组合方案，而且不用被被线程管理和信号互斥这些繁琐的操作分散精力。

在并发编程中，对共享资源的正确访问需要精确的控制，在目前的绝大多数语言中，都是通过加锁等线程同步方案来解决这一困难问题，而Go语言却另辟蹊径，它将共享的值通过信道传递(实际上多个独立执行的线程很少主动共享资源)。在任意给定的时刻，最好只有一个Goroutine能够拥有该资源。数据竞争从设计层面上就被杜绝了。为了提倡这种思考方式，Go语言将其并发编程哲学化为一句口号：

Do not communicate by sharing memory; instead, share memory by communicating.

不要通过共享内存来通信，而应通过通信来共享内存。

这是更高层次的并发编程哲学(通过管道来传值是Go语言推荐的做法)。虽然像引用计数这类简单的并发问题通过原子操作或互斥锁就能很好地实现，但是通过信道来控制访问能够让你写出更简洁正确的程序。

### 并发版本的Hello world

我们先以在一个新的Goroutine中输出“Hello world”，`main` 等待后台线程输出工作完成之后退出，这样一个简单的并发程序作为热身。

并发编程的核心概念是同步通信，但是同步的方式却有多种。我们先以大家熟悉的互斥量`sync.Mutex`来实现同步通信。根据文档，我们不能直接对一个未加锁状态的`sync.Mutex`进行解锁，这会导致运行时异常。下面这种方式并不能保证正常工作：

```

func main() {
    var mu sync.Mutex

    go func(){
        fmt.Println("你好，世界")
        mu.Lock()
    }()

    mu.Unlock()
}

```

因为 `mu.Lock()` 和 `mu.Unlock()` 并不在同一个 Goroutine 中，所以也就不满足顺序一致性内存模型。同时它们也没有其它的同步事件可以参考，这两个事件不可排序也就是可以并发的。因为可能是并发的事件，所以 `main` 函数中的 `mu.Unlock()` 很有可能先发生，而这个时刻 `mu` 互斥对象还处于未加锁的状态，从而会导致运行时异常。

下面是修复后的代码：

```

func main() {
    var mu sync.Mutex

    mu.Lock()
    go func(){
        fmt.Println("你好，世界")
        mu.Unlock()
    }()

    mu.Lock()
}

```

修复的方式是在 `main` 函数所在线程中执行两次 `mu.Lock()`，当第二次加锁时会因为锁已经被占用（不是递归锁）而阻塞，`main` 函数的阻塞状态驱动后台线程继续向前执行。当后台线程执行到 `mu.Unlock()` 时解锁，此时打印工作已经完成了，解锁会导致 `main` 函数中的第二个 `mu.Lock()` 阻塞状态取消，此时后台线程和主线程再没有其它的同步事件参考，它们退出的事件将是并发的：在 `main` 函数退出导致程序退出时，后台线程可能已经退出了，也可能没有退出。虽然无法确定两个线程退出的时间，但是打印工作是可以正确完成的。

使用 `sync.Mutex` 互斥锁同步是比较低级的做法。我们现在改用无缓存的管道来实现同步：

```

func main() {
    done := make(chan int)

    go func(){
        fmt.Println("你好，世界")
        <-done
    }()

    done <- 1
}

```

根据Go语言内存模型规范，对于从无缓冲信道进行的接收，发生在对该信道进行的发送完成之前。因此，后台线程 `<-done` 接收操作完成之后，`main` 线程的 `done <- 1` 发生操作才可能完成（从而退出`main`、退出程序），而此时打印工作已经完成了。

上面的代码虽然可以正确同步，但是对管道的缓存大小太敏感：如果管道有缓存的话，就无法保证能`main`退出之前后台线程能正常打印了。更好的做法是将管道的发送和接收方向调换一下，这样可以避免同步事件受管道缓存大小的影响：

```

func main() {
    done := make(chan int, 1) // 带缓存的管道

    go func(){
        fmt.Println("你好，世界")
        done <- 1
    }()

    <-done
}

```

对于带缓冲的Channel，对于Channel的第K个接收完成操作发生在第K+C个发送操作完成之前，其中C是Channel的缓存大小。虽然管道是带缓存的，`main` 线程接收完成是在后台线程发送开始但还未完成的时刻，此时打印工作也是已经完成的。

基于带缓存的管道，我们可以很容易将打印线程扩展到N个。下面的例子是开启10个后台线程分别打印：

```

func main() {
    done := make(chan int, 10) // 带 10 个缓存

    // 开N个后台打印线程
    for i := 0; i < cap(done); i++ {
        go func(){
            fmt.Println("你好， 世界")
            done <- 1
        }()
    }

    // 等待N个后台线程完成
    for i := 0; i < cap(done); i++ {
        <-done
    }
}

```

对于这种要等待N个线程完成后再进行下一步的同步操作有一个简单的方法，就是使用 `sync.WaitGroup` 来等待一组事件：

```

func main() {
    var wg sync.WaitGroup

    // 开N个后台打印线程
    for i := 0; i < 10; i++ {
        wg.Add(1)

        go func() {
            fmt.Println("你好， 世界")
            wg.Done()
        }()
    }

    // 等待N个后台线程完成
    wg.Wait()
}

```

其中 `wg.Add(1)` 用于增加等待事件的个数，必须确保在后台线程启动之前执行（如果放到后台线程之中执行则不能保证被正常执行到）。当后台线程完成打印工作之后，调用 `wg.Done()` 表示完成一个事件。`main` 函数的 `wg.Wait()` 是等待全部的事件完成。

## 生产者消费者模型

并发编程中最常见的例子就是生产者/消费者模式，该模式主要通过平衡生产线程和消费线程的工作能力来提高程序的整体处理数据的速度。简单地说，就是生产者生产一些数据，然后放到成果队列中，同时消费者从成果队列中来取这些数据。这样就让生产消费变成了异步的

两个过程。当成果队列中没有数据时，消费者就进入饥饿的等待中；而当成果队列中数据已满时，生产者则面临因产品挤压导致CPU被剥夺的下岗问题。

Go语言实现生产者消费者并发很简单：

```
// 生产者：生成 factor 整数倍的序列
func Producer(factor int, out chan<- int) {
    for i := 0; ; i++ {
        out <- i*factor
    }
}

// 消费者
func Consumer(in <-chan int) {
    for v := range in {
        fmt.Println(v)
    }
}

func main() {
    ch := make(chan int, 64) // 成果队列

    go Producer(3, ch) // 生成 3 的倍数的序列
    go Producer(5, ch) // 生成 5 的倍数的序列
    go Consumer(ch)    // 消费 生成的队列

    // 运行一定时间后退出
    time.Sleep(5 * time.Second)
}
```

我们开启了2个 `Producer` 生产流水线，分别用于生成3和5的倍数的序列。然后开启1个 `consumer` 消费者线程，打印获取的结果。我们通过在 `main` 函数休眠一定的时间来让生产者和消费者工作一定时间。正如前面一节说的，这种靠休眠方式是无法保证稳定的输出结果的。

我们可以让 `main` 函数保存阻塞状态不退出，只有当用户输入 `ctrl-c` 时才真正退出程序：

```
func main() {
    ch := make(chan int, 64) // 成果队列

    go Producer(3, ch) // 生成 3 的倍数的序列
    go Producer(5, ch) // 生成 5 的倍数的序列
    go Consumer(ch)    // 消费 生成的队列

    // Ctrl+C 退出
    sig := make(chan os.Signal, 1)
    signal.Notify(sig, syscall.SIGINT, syscall.SIGTERM)
    fmt.Printf("quit (%v)\n", <-sig)
}
```

我们这个例子中有2个生产者，并且2个生产者之间并无同步事件可参考，它们是并发的。因此，消费者输出的结果序列的顺序是不确定的，这并没有问题，生产者和消费者依然可以相互配合工作。

## 发布订阅模型

发布／订阅（publish-and-subscribe）模型通常被简写为pub／sub模型。在这个模型中，消息生产者成为发布者（publisher），而消息消费者则称对应订阅者（subscriber），生产者和消费者是M：N的关系。在传统生产者和消费者模型中，成果是将消息发送到一个队列中，而发布/订阅模型则是将消息发布给一个主题。

为此，我们构建了一个名为 `pubsub` 的发布订阅模型支持包：

```
// Package pubsub implements a simple multi-topic pub-sub library.
package pubsub

import (
    "sync"
    "time"
)

type (
    subscriber chan interface{}           // 订阅者为一个管道
    topicFunc  func(v interface{}) bool   // 主题为一个过滤器
)

// 发布者对象
type Publisher struct {
    m          sync.RWMutex           // 读写锁
    buffer     int                   // 订阅队列的缓存大小
    timeout    time.Duration        // 发布超时时间
    subscribers map[subscriber]topicFunc // 订阅者信息
}

// 构建一个发布者对象，可以设置发布超时时间和缓存队列的长度
func NewPublisher(publishTimeout time.Duration, buffer int) *Publisher {
    return &Publisher{
        buffer:     buffer,
        timeout:    publishTimeout,
        subscribers: make(map[subscriber]topicFunc),
    }
}

// 添加一个新的订阅者，订阅全部主题
func (p *Publisher) Subscribe() chan interface{} {
    return p.SubscribeTopic(nil)
}
```

```

// 添加一个新的订阅者，订阅过滤器筛选后的主题
func (p *Publisher) SubscribeTopic(topic topicFunc) chan interface{} {
    ch := make(chan interface{}, p.buffer)
    p.m.Lock()
    p.subscribers[ch] = topic
    p.m.Unlock()
    return ch
}

// 退出订阅
func (p *Publisher) Evict(sub chan interface{}) {
    p.m.Lock()
    defer p.m.Unlock()

    delete(p.subscribers, sub)
    close(sub)
}

// 发布一个主题
func (p *Publisher) Publish(v interface{}) {
    p.m.RLock()
    defer p.m.RUnlock()

    var wg sync.WaitGroup
    for sub, topic := range p.subscribers {
        wg.Add(1)
        go p.sendTopic(sub, topic, v, &wg)
    }
    wg.Wait()
}

// 关闭发布者对象，同时关闭所有的订阅者管道。
func (p *Publisher) Close() {
    p.m.Lock()
    defer p.m.Unlock()

    for sub := range p.subscribers {
        delete(p.subscribers, sub)
        close(sub)
    }
}

// 发送主题，可以容忍一定的超时
func (p *Publisher) sendTopic(sub subscriber, topic topicFunc, v interface{}, wg *sync.WaitGroup) {
    defer wg.Done()
    if topic != nil && !topic(v) {
        return
    }

    select {
    case sub <- v:
    case <-time.After(p.timeout):
    }
}

```

}

下面的例子中，有两个订阅者分别订阅了全部主题和含有"gomang"的主题：

```
import "path/to/pubsub"

func main() {
    p := pubsub.NewPublisher(100*time.Millisecond, 10)
    defer p.Close()

    all := p.Subscribe()
    golang := p.SubscribeTopic(func(v interface{}) bool {
        if s, ok := v.(string); ok {
            return strings.Contains(s, "golang")
        }
        return false
    })

    p.Publish("hello, world!")
    p.Publish("hello, golang!")

    go func() {
        for msg := range all {
            fmt.Println("all:", msg)
        }
    }()

    go func() {
        for msg := range golang {
            fmt.Println("golang:", msg)
        }
    }()
}

// 运行一定时间后退出
time.Sleep(3 * time.Second)
}
```

在发布订阅模型中，每条消息都会传送给多个订阅者。发布者通常不会知道、也不关心哪一个订阅者正在接收主题消息。订阅者和发布者可以在运行时动态添加是一种松散的耦合关心，这使得系统的复杂性可以随时间的推移而增长。在现实生活中，不同城市的象天气预报之类的应用就可以应用这个并发模式。

## 赢家为王

采用并发编程的动机有很多：并发编程可以简化问题，比如一类问题对应一个处理线程会更简单；并发编程还可以提升性能，在一个多核CPU上开2个线程一般会比开1个线程快一些。其实对于提升性能而言，程序并不是简单地运行速度快就表示用户体验好的；很多时候程序能快速响应用户请求才是最重要的，当没有用户请求需要处理的时候才合适处理一些低优先级的后台任务。

假设我们想快速地检索“golang”相关的主题，我们可能会同时打开Bing、Google或百度等多个搜索引擎。当某个检索最先返回结果后，就可以关闭其它检索页面了。因为受限于网络环境和搜索引擎算法的影响，某些搜索引擎可能很快返回检索结果，某些搜索引擎也可能遇到等到他们公司倒闭也没有完成检索的情况。我们可以采用类似的策略来编写这个程序：

```
func main() {
    ch := make(chan string, 32)

    go func() {
        ch <- searchByBing("golang")
    }
    go func() {
        ch <- searchByGoogle("golang")
    }
    go func() {
        ch <- searchByBaidu("golang")
    }

    fmt.Println(<-ch)
}
```

首先，我们创建了一个带缓存的管道，管道的缓存数目要足够大，保证不会因为缓存的容量引起不必要的阻塞。然后我们开启了多个后台线程，分别向不同的搜索引擎提交检索请求。当任意一个搜索引擎最先有结果之后，都会马上将结果发到管道中（因为管道带了足够的缓存，这个过程不会阻塞）。但是最终我们只从管道取第一个结果，也就是最先返回的结果。

通过适当开启一些冗余的线程，尝试用不同途径去解决同样的问题，最终以赢者为王的方式提升了程序的相应性能。

## 控制并发数

很多用户在适应了Go语言强大的并发特性之后，都倾向于编写最大并发的程序，因为这样似乎可以提供最大的性能。在现实中我们行色匆匆，但有时却需要我们放慢脚步享受生活，并发的程序也是一样：有时候我们需要适当地控制并发的程度，因为这样不仅仅可给其它的应用/任务让出/预留一定的CPU资源，也可以适当降低功耗缓解电池的压力。

在 Go 语言自带的 godoc 程序实现中有一个 `vfs` 的包对应虚拟的文件系统，在 `vfs` 包下面有一个 `gatefs` 的子包，`gatefs` 子包的目的就是为了控制访问该虚拟文件系统的最大并发数。`gatefs` 包的应用很简单：

```
import (
    "golang.org/x/tools/godoc/vfs"
    "golang.org/x/tools/godoc/vfs/gatefs"
)

func main() {
    fs := gatefs.New(vfs.OS("/path"), make(chan bool, 8))
    // ...
}
```

其中 `vfs.OS("/path")` 基于本地文件系统构造一个虚拟的文件系统，然后 `gatefs.New` 基于现有的虚拟文件系统构造一个并发受控的虚拟文件系统。并发数控制的原理在前面一节已经讲过，就是通过带缓存管道的发送和接收规则来实现最大并发阻塞：

```
var limit = make(chan int, 3)

func main() {
    for _, w := range work {
        go func() {
            limit <- 1
            w()
            <-limit
        }()
    }
    select{}
}
```

不过 `gatefs` 对此做一个抽象类型 `gate`，增加了 `enter` 和 `leave` 方法分别对应并发代码的进入和离开。当超出并发数目限制的时候，`enter` 方法会阻塞直到并发数降下来为止。

```
type gate chan bool

func (g gate) enter() { g <- true }
func (g gate) leave() { <-g }
```

`gatefs` 包装的新的虚拟文件系统就是将需要控制并发的方法增加了 `enter` 和 `leave` 调用而已：

```

type gatefs struct {
    fs vfs.FileSystem
    gate
}

func (fs gatefs) Lstat(p string) (os.FileInfo, error) {
    fs.enter()
    defer fs.leave()
    return fs.fs.Lstat(p)
}

```

我们不仅可以控制最大的并发数目，而且可以通过带缓存Channel的使用量和最大容量比例来判断程序运行的并发率。当管道为空的时候可以认为是空闲状态，当管道满了时任务是繁忙状态，这对于后台一些低级任务的运行是有参考价值的。增加的方法如下：

```

func (g gate) Len() int { return len(g) }
func (g gate) Cap() int { return cap(g) }

func (g gate) Idle() bool { return len(g) == 0 }
func (g gate) Busy() bool { return len(g) == cap(g) }

func (g gate) Fraction() float64 {
    return float64(len(g)) / float64(cap(g))
}

```

然后我们可以在相对空闲的时候处理一些后台低优先级的任务，在并发相对繁忙或超出一定比例的时候提供预警：

```

func New(fs vfs.FileSystem, gate chan bool) *gatefs {
    p := &gatefs{fs, gate}

    // 后台监控线程
    go func() {
        for {
            switch {
            case p.gate.Idle():
                // 处理后台任务
            case p.gate.Fraction() >= 0.7:
                // 并发预警
            default:
                time.Sleep(time.Second)
            }
        }
    }()
}

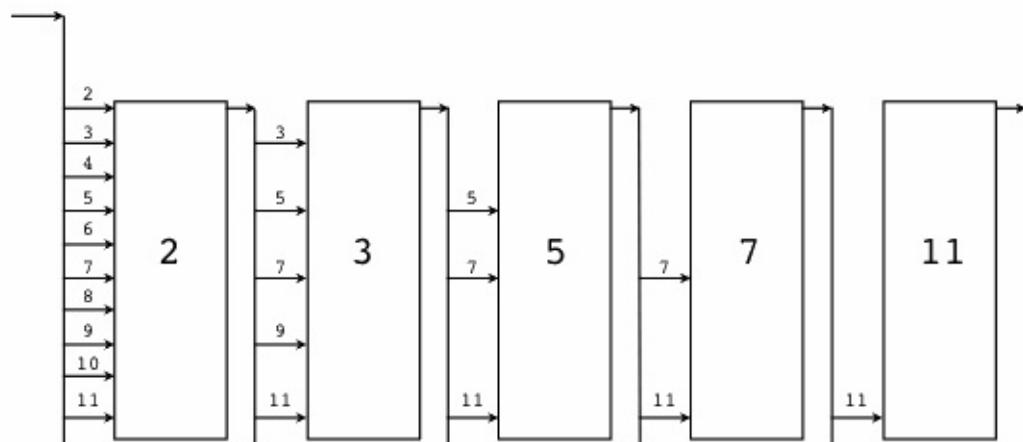
return p
}

```

这样我们通过后台线程就可以根据程序的状态动态调整自己的工作模式。

## 素数筛

在“Hello world 的革命”一节中，我们为了演示Newsqueak的并发特性，文中给出了并发版本素数筛的实现。并发版本的素数筛是一个经典的并发例子，通过它我们可以更深刻地理解Go语言的并发特性。“素数筛”的原理如图：



我们需要先生成最初的 `2, 3, 4, ...` 自然数序列（不包含开头的0、1）：

```
// 返回生成自然数序列的管道: 2, 3, 4, ...
func GenerateNatural() chan int {
    ch := make(chan int)
    go func() {
        for i := 2; ; i++ {
            ch <- i
        }
    }()
    return ch
}
```

`GenerateNatural` 函数内部启动一个Goroutine生产序列，返回对应的管道。

然后是为每个素数构造一个筛子：将输入序列中是素数倍数的数提出，并返回新的序列，是一个新的管道。

```
// 管道过滤器：删除能被素数整除的数
func PrimeFilter(in <-chan int, prime int) chan int {
    out := make(chan int)
    go func() {
        for {
            if i := <-in; i%prime != 0 {
                out <- i
            }
        }
    }()
    return out
}
```

`PrimeFilter` 函数也是内部启动一个 `Goroutine` 生产序列，返回过滤后序列对应的管道。

现在我们可以在 `main` 函数中驱动这个并发的素数筛选了：

```
func main() {
    ch := GenerateNatural() // 自然数序列：2, 3, 4, ...
    for i := 0; i < 100; i++ {
        prime := <-ch // 新出现的素数
        fmt.Printf("%v: %v\n", i+1, prime)
        ch = PrimeFilter(ch, prime) // 基于新素数构造的过滤器
    }
}
```

我们先是调用 `GenerateNatural()` 生成最原始的从2开始的自然数序列。然后开始一个100次迭代的循环，希望生成100个素数。在每次循环迭代开始的时候，管道中的第一个数必定是素数，我们先读取并打印这个素数。然后基于管道中剩余的数列，并以当前取出的素数为筛选过滤后面的素数。不同的素数筛选对应的管道是串联在一起的。

素数筛选展示了一种优雅的并发程序结构。但是因为每个并发体处理的任务粒度太细微，程序整体的性能并不理想。对于细力度的并发程序，CSP模型中固有的消息传递的代价太高了（多线程并发模型同样要面临线程启动的代价）。

## 并发的安全退出

有时候我们需要通知 `goroutine` 停止它正在干的事情，特别是当它工作在错误的方向上的时候。`Go` 语言并没有提供在一个直接终止 `Goroutine` 的方法，由于这样会导致 `goroutine` 之间的共享变量落在未定义的状态上。但是如果我们想要退出两个或者任意多个 `Goroutine` 怎么办呢？

`Go` 语言中不同 `Goroutine` 之间主要依靠管道进行通信和同步。要同时处理多个管道的发送或接收操作，我们需要使用 `select` 关键字（这个关键字和网络编程中的 `select` 函数的行为类似）。当 `select` 有多个分支时，会随机选择一个可用的管道分支，如果没有可用的管道分支

则选择 `default` 分支，否则会一直保存阻塞状态。

基于 `select` 实现的管道的超时判断：

```
select {
case v := <-in:
    fmt.Println(v)
case <-time.After(time.Second):
    return // 超时
}
```

通过 `select` 的 `default` 分支实现非阻塞的管道发送或接收操作：

```
select {
case v := <-in:
    fmt.Println(v)
default:
    // 没有数据
}
```

通过 `select` 来阻止 `main` 函数退出：

```
func main() {
    // do some things
    select{}}
}
```

当有多个管道均可操作时，`select` 会随机选择一个管道。基于该特性我们可以用 `select` 实现一个生成随机数列的程序：

```
func main() {
    ch := make(chan int)
    go func() {
        for {
            select {
            case ch <- 0:
            case ch <- 1:
            }
        }
    }()
    for v := range ch {
        fmt.Println(v)
    }
}
```

我们通过 `select` 和 `default` 分支可以很容易实现一个Goroutine的退出控制：

```

func worker(cancel chan bool) {
    for {
        select {
        default:
            fmt.Println("hello")
            // 正常工作
        case <-cancel:
            // 退出
        }
    }
}

func main() {
    cancel := make(chan bool)
    go worker(cancel)

    time.Sleep(time.Second)
    cancel <- true
}

```

但是管道的发送操作和接收操作是一一对应的，如果要停止多个Goroutine那么可能需要创建同样数量的管道，这个代价太大了。其实我们可以通过 `close` 关闭一个管道来实现广播的效果，所有从关闭管道接收的操作均会收到一个零值和一个可选的失败标志。

```

func worker(cancel chan bool) {
    for {
        select {
        default:
            fmt.Println("hello")
            // 正常工作
        case <-cancel:
            // 退出
        }
    }
}

func main() {
    cancel := make(chan bool)

    for i := 0; i < 10; i++ {
        go worker(cancel)
    }

    time.Sleep(time.Second)
    close(cancel)
}

```

我们通过 `close` 来关闭 `cancel` 管道向多个Goroutine广播退出的指令。不过这个程序依然不够稳健：当每个Goroutine收到退出指令退出时一般会进行一定的清理工作，但是退出的清理工作并不能保证被完成，因为 `main` 线程并没有等待各个工作Goroutine退出工作完成的机制。我们可以结合 `sync.WaitGroup` 来改进：

```
func worker(wg *sync.WaitGroup, cancel chan bool) {
    defer wg.Done()

    for {
        select {
        default:
            fmt.Println("hello")
        case <-cancel:
            return
        }
    }
}

func main() {
    cancel := make(chan bool)

    var wg sync.WaitGroup
    for i := 0; i < 10; i++ {
        wg.Add(1)
        go worker(&wg, cancel)
    }

    time.Sleep(time.Second)
    close(cancel)
    wg.Wait()
}
```

现在每个工作者并发体的创建、运行、暂停和退出都是在 `main` 函数的安全控制之下了。

## 消费海量的请求

在前面的生产者、消费者并发模型中，只有当生产者和消费的速度近似相等时才会达到最佳的效果，同时通过引入带缓存的管道可以消除因临时效率波动产生的影响。但是当生产者和消费者的速度严重不匹配时，我们是无法通过带缓存的管道来提高性能的（缓存的管道只能延缓问题发生的时间，无法消除速度差异带来的问题）。当消费者无法及时消费生产者的输出时，时间积累会导致问题越来越严重。

对于生产者、消费者并发模型，我们当然可以通过降低生产者的产能来避免资源的浪费。但在很多场景中，生产者才是核心对象，它们生产出各种问题或任务单据，这时候产出的问题是必须要解决的、任务单据也是必须要完成的。在现实生活中，制造各种生活垃圾的海量人类其实就是垃圾生产者，而清理生活垃圾的少量的清洁工就是垃圾消费者。在网络服务中，

提交POST数据的海量用户则变成了生产者，Web后台服务则对应POST数据的消费者。海量生产者的问题也就变成了：如何构造一个能够处理海量请求的Web服务（假设每分钟百万级请求）。

在Web服务中，用户提交的每个POST请求可以看作是一个Job任务，而服务器是通过后台的Worker工作者来消费这些Job任务。当面向海量的Job处理时，我们一般可以通过构造一个Worker工作者池来提高Job的处理效率；通过通过一个带缓存的Job管道来接收新的任务请求，避免任务请求功能无法响应；Job请求接收管道和Worker工作者池通过分发系统来衔接。

我们可以用管道来模拟工作者池：当需要处理一个任务时，先从工作者池取一个工作者，处理完任务之后将工作者返回给工作者池。`WorkerPool` 对应工作者池，`Worker` 对应工作者。

```

type WorkerPool struct {
    workers []*Worker
    pool     chan *Worker
}

// 构造工作者池
func NewWorkerPool(maxWorkers int) *WorkerPool {
    p := &WorkerPool{
        workers: make([]*Worker, maxWorkers),
        pool:    make(chan *Worker, maxWorkers),
    }

    // 初始化工作者
    for i, _ := range p.workers {
        worker := NewWorker(0)
        p.workers[i] = worker
        p.pool <- worker
    }
    return p
}

// 启动工作者
func (p *WorkerPool) Start() {
    for _, worker := range p.workers {
        worker.Start()
    }
}

// 停止工作者
func (p *WorkerPool) Stop() {
    for _, worker := range p.workers {
        worker.Stop()
    }
}

// 获取工作者(阻塞)
func (p *WorkerPool) Get() *Worker {
    return <-p.pool
}

// 返回工作者
func (p *WorkerPool) Put(w *Worker) {
    p.pool <- w
}

```

工作者池通过一个带缓存的管道来提高工作者的管理。当所有工作者都在处理任务时，工作者的获取会阻塞直到有工作者可用为止。

`Worker` 对应工作者实现，具体任务由后台一个固定的 Goroutine 完成，和外界通过专有的管道通信（工作者的私有管道也可以选择带有一定的缓存）具体实现如下：

```

type Worker struct {
    job  chan interface{}
    quit chan bool
    wg   sync.WaitGroup
}

// 构造工作者
func NewWorker(maxJobs int) *Worker {
    return &Worker{
        job:  make(chan interface{}, maxJobs),
        quit: make(chan bool),
    }
}

// 启动任务
func (w *Worker) Start() {
    p.wg.Add(1)

    go func() {
        defer p.wg.Done()

        for {
            // 接收任务
            // 此时工作中已经从工作者池中取出
            select {
            case job := <-p.job:
                // 处理任务

            case <-w.quit:
                return
            }
        }
    }()
}
}

// 关闭任务
func (p *Worker) Stop() {
    p.quit <- true
    p.wg.Wait()
}

// 提交任务
func (p *Worker) AddJob(job interface{}) {
    p.job <- job
}

```

任务的分发系统在 `Service` 对象中完成：

```

type Service struct {
    workers *WorkerPool
    jobs    chan interface{}
    maxJobs int
    wg      sync.WaitGroup
}

func NewService(maxWorkers, maxJobs int) *Service {
    return &Service {
        workers: NewWorkerPool(maxWorkers),
        jobs:    make(chan interface{}, maxJobs),
    }
}

func (p *Service) Start() {
    p.jobs = make(chan interface{}, maxJobs)

    p.wg.Add(1)
    p.workers.Start()

    go func() {
        defer p.wg.Done()

        for job := range p.jobs {
            go func(job Job) {
                // 从工作者池取一个工作者
                worker := p.workers.Get()

                // 完成任务后返回给工作者池
                defer p.workers.Put(worker)

                // 提交任务处理(异步)
                worker.AddJob(job)
            }(job)
        }
    }()
}

func (p *Service) Stop() {
    p.workers.Stop()
    close(p.jobs)
    p.wg.Wait()
}

// 提交任务
// 任务管道带较大的缓存，延缓阻塞的时间
func (p *Service) AddJob(job interface{}) {
    p.jobs <- job
}

```

主程序可以是一个**web**服务器：

```

var (
    MaxWorker = os.Getenv("MAX_WORKERS")
    MaxQueue = os.Getenv("MAX_QUEUE")
)

func main() {
    service := NewService(MaxWorker, MaxQueue)

    service.Start()
    defer service.Stop()

    // 处理海量的任务
    http.HandleFunc("/jobs", func(w http.ResponseWriter, r *http.Request) {
        if r.Method != "POST" {
            w.WriteHeader(http.StatusMethodNotAllowed)
            return
        }

        // Job以JSON格式提交
        var jobs []Job
        err := json.NewDecoder(io.LimitReader(r.Body, maxLength)).Decode(&jobs)
        if err != nil {
            w.Header().Set("Content-Type", "application/json; charset=UTF-8")
            w.WriteHeader(http.StatusBadRequest)
            return
        }

        // 处理任务
        for _, job := range jobs {
            service.AddJob(job)
        }

        // OK
        w.WriteHeader(http.StatusOK)
    })
}

// 启动web服务
log.Fatal(http.ListenAndServe(":8080", nil))
}

```

基于Go语言特有的管道和Goroutine特性，我们以非常简单的方式设计了一个针对海量请求的处理系统结构。在世纪的系统中，用户可以根据任务的具体类型和特性，将管道定义为具体类型以避免接口等动态特性导致的开销。

## 更多

在Go1.7发布时，标准库增加了一个 `context` 包，用来简化对于处理单个请求的多个 Goroutine 之间与请求域的数据、超时和退出等操作，官方有博文对此做了专门介绍。我们可以用 `context` 包来重新实现前面的线程安全退出或超时的控制：

```
func worker(ctx context.Context, wg *sync.WaitGroup) error {
    defer wg.Done()

    for {
        select {
        default:
            fmt.Println("hello")
        case <-ctx.Done():
            return ctx.Err()
        }
    }
}

func main() {
    ctx, cancel := context.WithTimeout(context.Background(), 10*time.Second)

    var wg sync.WaitGroup
    for i := 0; i < 10; i++ {
        wg.Add(1)
        go worker(ctx, &wg)
    }

    time.Sleep(time.Second)
    cancel()

    wg.Wait()
}
```

当并发体超时或 `main` 主动停止工作者 Goroutine 时，每个工作者都可以安全退出。

Go 语言是带内存自动回收的特性，因此内存一般不会泄漏。在前面素数筛的例子中，`GenerateNatural` 和 `PrimeFilter` 函数内部都启动了新的 Goroutine，当 `main` 函数不再使用管道时后台 Goroutine 有泄漏的风险。我们可以通过 `context` 包来避免做个问题，下面是改进的素数筛实现：

```

// 返回生成自然数序列的管道: 2, 3, 4, ...
func GenerateNatural(ctx context.Context) chan int {
    ch := make(chan int)
    go func() {
        for i := 2; ; i++ {
            select {
            case <- ctx.Done():
                return
            case ch <- i:
            }
        }
    }()
    return ch
}

// 管道过滤器: 删除能被素数整除的数
func PrimeFilter(ctx context.Context, in <-chan int, prime int) chan int {
    out := make(chan int)
    go func() {
        for {
            if i := <-in; i%prime != 0 {
                select {
                case <- ctx.Done():
                    return
                case out <- i:
                }
            }
        }
    }()
    return out
}

func main() {
    // 通过 Context 控制后台Goroutine状态
    ctx, cancel := context.WithCancel(context.Background())

    ch := GenerateNatural(ctx) // 自然数序列: 2, 3, 4, ...
    for i := 0; i < 100; i++ {
        prime := <-ch // 新出现的素数
        fmt.Printf("%v: %v\n", i+1, prime)
        ch = PrimeFilter(ctx, ch, prime) // 基于新素数构造的过滤器
    }

    cancel()
}

```

当main函数完成工作前，通过调用 `cancel()` 来通知后台Goroutine退出，这样就避免了 Goroutine的泄漏。

并发是一个非常大的主题，我们这里只是展示几个非常基础的并发编程的例子。官方文档也有很多关于并发编程的讨论，国内也有专门讨论Go语言并发编程的书籍。读者可以根据自己的需求查阅相关的文献。

## 1.7. 错误和异常

错误处理是每个编程语言都要考虑的一个重要话题。在Go语言的错误处理中，错误是软件包API和应用程序用户界面的一个重要组成部分。

在程序中总有一部分函数总是要求必须能够成功的运行。比如 `strconv.Itoa` 将整数转换为字符串，从数组或切片中读写元素，从 `map` 读取已经存在的元素等。这类操作在运行时几乎不会失败，除非程序中有BUG，或遇到灾难性的、不可预料的情况，比如运行时的内存溢出。如果真的遇到真正异常情况，我们只要简单终止程序就可以了。

排除异常的情况，如果程序运行失败仅被认为是几个预期的结果之一。对于那些将运行失败看作是预期结果的函数，它们会返回一个额外的返回值，通常是最后一个来传递错误信息。如果导致失败的原因只有一个，额外的返回值可以是一个布尔值，通常被命名为`ok`。比如，当从一个 `map` 查询一个结果时，可以通过额外的布尔值判断是否成功：

```
if v, ok := m["key"]; ok {
    return v
}
```

但是导致失败的原因通常不止一种，很多时候用户希望了解更多的错误信息。如果只是用简单的布尔类型的状态值将不能满足这个要求。在C语言中，默认采用一个整数类型的 `errno` 来表达错误，这样就可以根据需要定义多种错误类型。在Go语言中，`syscall.Errno` 就是对应C语言中 `errno` 类型的错误。在 `syscall` 包中的接口，如果有返回错误的话，底层也是 `syscall.Errno` 错误类型。

比如我们通过 `syscall` 包的接口来修改文件的模式时，如果遇到错误我们可以将 `err` 强制断言为 `syscall.Errno` 错误类型处理：

```
err := syscall.Chmod(":invalid path:", 0666)
if err != nil {
    log.Fatal(err.(syscall.Errno))
}
```

我们还可以进一步地通过类型查询或类型断言来获取底层真实的错误类型，这样就可以获取更详细的错误信息。不过一般情况下我们并不关心错误在底层的表达方式，我们只需要知道它是一个错误就可以了。当返回的错误值不是 `nil` 时，我们可以通过调用 `error` 接口类型的 `Error` 方法来获得字符串类型的错误信息。

在Go语言中，错误被认为是一种可以预期的结果；而异常则是一种非预期的结果，发生异常可能表示程序中存在BUG或发生了其它不可控的问题。Go语言推荐使用 `recover` 函数将内部异常转为错误处理，这使得用户可以真正的关心业务相关的错误处理。

如果某个接口简单地将所有普通的错误当做异常抛出，将会使错误信息杂乱且没有价值。就像在 `main` 函数中直接捕获全部一样，是没有意义的：

```
func main() {
    defer func() {
        if r := recover(); r != nil {
            log.Fatal(r)
        }
    }()
    ...
}
```

捕获异常不是最终的目的。如果异常不可预测，直接输出异常信息是最好的处理方式。

## 错误处理策略

让我们演示一个文件复制的例子：函数需要打开两个文件，然后将其中一个文件的内容复制到另一个文件：

```
func CopyFile(dstName, srcName string) (written int64, err error) {
    src, err := os.Open(srcName)
    if err != nil {
        return
    }

    dst, err := os.Create(dstName)
    if err != nil {
        return
    }

    written, err = io.Copy(dst, src)
    dst.Close()
    src.Close()
    return
}
```

上面的代码虽然能够工作，但是隐藏一个bug。如果第一个 `os.Open` 调用失败，那么会在没有释放 `src` 文件资源的情况下返回。虽然我们可以通过在第二个返回语句前添加 `src.Close()` 调用来修复这个BUG；但是当代码变得复杂时，类似的问题将很难被发现和修复。我们可以通过 `defer` 语句来确保每个被正常打开的文件都能被正常关闭：

```

func CopyFile(dstName, srcName string) (written int64, err error) {
    src, err := os.Open(srcName)
    if err != nil {
        return
    }
    defer src.Close()

    dst, err := os.Create(dstName)
    if err != nil {
        return
    }
    defer dst.Close()

    return io.Copy(dst, src)
}

```

`defer` 语句可以让我们在打开文件时马上思考如何关闭文件。不管函数如何返回，文件关闭语句始终会被执行。同时 `defer` 语句可以保证，即使 `io.Copy` 发生了异常，文件依然可以安全地关闭。

前文我们说到，Go语言中的导出函数一般不抛出异常，一个未受控的异常可以看作是程序的BUG。但是对于那些提供类似Web服务的框架而言；它们经常需要接入第三方的中间件。因为第三方的中间件是否存在BUG是否会抛出异常，Web框架本身是不能确定的。为了提高系统的稳定性，Web框架一般会通过 `recover` 来防御性地捕获所有处理流程中可能产生的异常，然后将异常转为普通的错误返回。

让我们以JSON解析器为例，说明`recover`的使用场景。考虑到JSON解析器的复杂性，即使某个语言解析器目前工作正常，也无法肯定它没有漏洞。因此，当某个异常出现时，我们不会选择让解析器崩溃，而是会将`panic`异常当作普通的解析错误，并附加额外信息提醒用户报告此错误。

```

func ParseJSON(input string) (s *Syntax, err error) {
    defer func() {
        if p := recover(); p != nil {
            err = fmt.Errorf("JSON: internal error: %v", p)
        }
    }()
    // ...parser...
}

```

在标准库中的 `json` 包，在内部递归解析JSON数据的时候如果遇到错误，会通过抛出异常的方式来快速跳出深度嵌套的函数调用，然后由最外以及的接口通过 `recover` 捕获 `panic`，然后返回相应的错误信息。

Go语言库的实现习惯：即使在包内部使用了 `panic`，但是在导出函数时会被转化为明确的错误值。

# 获取错误的上下文

有时候为了方便上层用户理解；很多时候底层实现者会将底层的错误重新包装为新的错误类型返回给用户：

```
if _, err := html.Parse(resp.Body); err != nil {
    return nil, fmt.Errorf("parsing %s as HTML: %v", url, err)
}
```

上层用户在遇到错误时，可以很容易从业务层面理解错误发生的原因。但是鱼和熊掌总是很难兼得，在上层用户获得新的错误的同时，我们也丢失了底层最原始的错误类型（只剩下错误描述信息了）。

为了记录这种错误类型在包装的变迁过程中的信息，我们一般会定义一个辅助的 `WrapError` 函数，用于包装原始的错误，同时保留完整的原始错误类型。为了问题定位的方便，同时也为了能记录错误发生时的函数调用状态，我们很多时候希望在出现致命错误的时候保存完整的函数调用信息。同时，为了支持RPC等跨网络的传输，我们可能要需要将错误序列化为类似JSON格式的数据，然后再从这些数据中将错误解码恢复出来。

为此，我们可以定义自己的 `github.com/chai2010/errors` 包，里面是以下的错误类型：

```
type Error interface {
    Caller() []CallerInfo
    Wraped() []error
    Code() int
    error

    private()
}

type CallerInfo struct {
    FuncName string
    FileName string
    FileLine int
}
```

其中 `Error` 为接口类型，是 `error` 接口类型的扩展，用于给错误增加调用栈信息，同时支持错误的多级嵌套包装，支持支持错误码格式。为了使用方便，我们可以定义以下的辅助函数：

```

func New(msg string) error
func NewWithCode(code int, msg string) error

func Wrap(err error, msg string) error
func WrapWithCode(code int, err error, msg string) error

func FromJson(json string) (Error, error)
func ToJson(err error) string

```

`New` 用于构建新的错误类型，和标准库中 `errors.New` 功能类似，但是增加了出错误时的函数调用栈信息。`FromJson` 用于从JSON字符串编码的错误中恢复错误对象。`NewWithCode` 则是构造一个带错误码的错误，同时也包含出错误时的函数调用栈信息。`Wrap` 和 `WrapWithCode` 则是错误二次包装函数，用于将底层的错误包装为新的错误，但是保留的原始的底层错误信息。这里返回的错误对象都可以直接调用 `json.Marshal` 将错误编码为JSON字符串。

我们可以这样使用包装函数：

```

import (
    "github.com/chai2010/errors"
)

func loadConfig() error {
    _, err := ioutil.ReadFile("/path/to/file")
    if err != nil {
        return errors.Wrap(err, "read failed")
    }

    // ...
}

func setup() error {
    err := loadConfig()
    if err != nil {
        return errors.Wrap(err, "invalid config")
    }

    // ...
}

func main() {
    if err := setup(); err != nil {
        log.Fatal(err)
    }

    // ...
}

```

上面的例子中，错误被进行了2层包装。我们可以这样遍历原始错误经历了哪些包装流程：

```
for i, e := range err.(errors.Error).Wrapped() {
    fmt.Printf("wrapped(%d): %v\n", i, e)
}
```

同时也可以获取每个包装错误的函数调用堆栈信息：

```
for i, x := range err.(errors.Error).Caller() {
    fmt.Printf("caller:%d: %s\n", i, x.FuncName)
}
```

如果需要将错误通过网络传输，可以用 `errorsToJson(err)` 编码为JSON字符串：

```
// 以JSON字符串方式发送错误
func sendError(ch chan<- string, err error) {
    ch <- errorsToJson(err)
}

// 接收JSON字符串格式的错误
func recvError(ch <-chan string) error {
    p, err := errors.FromJson(<-ch)
    if err != nil {
        log.Fatal(err)
    }
    return p
}
```

对于基于http协议的网络服务，我们还可以给错误绑定一个对应的http状态码：

```
err := errors.NewWithCode(404, "http error code")

fmt.Println(err)
fmt.Println(err.(errors.Error).Code())
```

在Go语言中，错误处理也有一套独特的编码风格。检查某个子函数是否失败后，我们通常将处理失败的逻辑代码放在处理成功的代码之前。如果某个错误会导致函数返回，那么成功时的逻辑代码不应放在 `else` 语句块中，而应直接放在函数体中。

```
f, err := os.Open("filename.ext")
if err != nil {
    // 失败的情形，马上返回错误
}

// 正常的处理流程
```

Go语言中大部分函数的代码结构几乎相同，首先是一系列的初始检查，用于防止错误发生，之后是函数的实际逻辑。

## 错误的错误返回

Go语言中的错误是一种接口类型。接口信息中包含了原始类型和原始的值。只有当接口的类型和原始的值都为空的时候，接口的值才对应 `nil`。其实当接口中类型为空的时候，原始值必然也是空的；反之，当接口对应的原始值为空的时候，接口对应的原始类型并不一定为空的。

在下面的例子中，试图返回自定义的错误类型，当没有错误的时候返回 `nil`：

```
func returnsError() error {
    var p *MyError = nil
    if bad() {
        p = ErrBad
    }
    return p // Will always return a non-nil error.
}
```

但是，最终返回的结果其实并非是 `nil`：是一个正常的错误，错误的值是一个 `MyError` 类型的空指针。下面是改进的 `returnsError`：

```
func returnsError() error {
    if bad() {
        return (*MyError)(err)
    }
    return nil
}
```

因此，在处理错误返回值的时候，没有错误的返回值最好直接写为 `nil`。

Go语言作为一个强类型语言，不同类型之前必须要显示的转换（而且必须有相同的基础类型）。但是，Go语言中 `interface` 是一个例外：非接口类型到接口类型，或者是接口类型之间的转换都是隐式的。这是为了支持方便的鸭子面向对象编程，当然会牺牲一定的安全特性。

## 剖析异常

`panic` 支持抛出任意类型的异常（而不仅仅是 `error` 类型的错误），`recover` 函数调用的返回值和 `panic` 函数的输入参数类型一致，它们的函数签名如下：

```
func panic(interface{})  
func recover() interface{}
```

Go语言函数调用的正常流程是函数执行返回语句返回结果，在这个流程中是没有异常的，因此在这个流程中执行 `recover` 异常捕获函数始终是返回 `nil`。另一种是异常流程：当函数调用 `panic` 抛出异常，函数将停止执行后续的普通语句，但是之前注册的 `defer` 函数调用仍然保证会被正常执行，然后再返回到的调用者。对于当前函数的调用者，因为处理异常状态还没有被捕获，和直接调用 `panic` 函数的行为类似。在异常发生时，如果在 `defer` 中执行 `recover` 调用，它可以捕获触发 `panic` 时的参数，并且恢复到正常的执行流程。

在非 `defer` 语句中执行 `recover` 调用是初学者常犯的错误：

```
func main() {  
    if r := recover(); r != nil {  
        log.Fatal(r)  
    }  
  
    panic(123)  
  
    if r := recover(); r != nil {  
        log.Fatal(r)  
    }  
}
```

上面程序中两个 `recover` 调用都不能捕获任何异常。在第一个 `recover` 调用执行时，函数必然是在正常的非异常执行流程中，这时候 `recover` 调用将返回 `nil`。发生异常时，第二个 `recover` 调用将没有机会被执行到，因为 `panic` 调用会导致函数马上执行已经注册 `defer` 的函数后返回。

其实 `recover` 函数调用有着更严格的要求：我们必须在 `defer` 函数中直接调用 `recover`。如果 `defer` 中调用的是 `recover` 函数的包装函数的话，异常的捕获工作将失败！比如，有时候我们可能希望包装自己的 `MyRecover` 函数，在内部增加必要的日志信息然后再调用 `recover`，这是错误的做法：

```

func main() {
    defer func() {
        // 无法捕获异常
        if r := MyRecover(); r != nil {
            fmt.Println(r)
        }
    }()
    panic(1)
}

func MyRecover() interface{} {
    log.Println("trace...")
    return recover()
}

```

同样，如果是在嵌套的 `defer` 函数中调用 `recover` 也将导致无法捕获异常：

```

func main() {
    defer func() {
        defer func() {
            // 无法捕获异常
            if r := recover(); r != nil {
                fmt.Println(r)
            }
        }()
    }()
    panic(1)
}

```

2层嵌套的 `defer` 函数中直接调用 `recover` 和1层 `defer` 函数中调用包装的 `MyRecover` 函数一样，都是经过了2个函数帧才到达真正的 `recover` 函数，这个时候Goroutine的对应上一级栈帧中已经没有异常信息。

如果我们直接在 `defer` 语句中调用 `MyRecover` 函数又可以正常工作了：

```

func MyRecover() interface{} {
    return recover()
}

func main() {
    // 可以正常捕获异常
    defer MyRecover()
    panic(1)
}

```

但是，如果 `defer` 语句直接调用 `recover` 函数，依然不能正常捕获异常：

```
func main() {
    // 无法捕获异常
    defer recover()
    panic(1)
}
```

必须要和有异常的栈帧只隔一个栈帧，`recover` 函数才能正常捕获异常。换言之，`recover` 函数捕获的是祖父一级调用函数栈帧的异常（刚好可以跨越一层 `defer` 函数）！

当然，为了避免 `recover` 调用者不能识别捕获到的异常，应该避免用 `nil` 为参数抛出异常：

```
func main() {
    defer func() {
        if r := recover(); r != nil { ... }
        // 虽然总是返回nil，但是可以恢复异常状态
    }()
    // 警告：用`nil`为参数抛出异常
    panic(nil)
}
```

当希望将捕获到的异常转为错误时，如果希望忠实返回原始的信息，需要针对不同的类型分别处理：

```
func foo() (err error) {
    defer func() {
        if r := recover(); r != nil {
            switch x := r.(type) {
            case string:
                err = errors.New(x)
            case error:
                err = x
            default:
                err = fmt.Errorf("Unknown panic: %v", r)
            }
        }
    }()
    panic("TODO")
}
```

基于这个代码模板，我们甚至可以模拟出不同类型的异常。通过为定义不同类型的保护接口，我们就可以区分异常的类型了：

```
func main {
    defer func() {
        if r := recover(); r != nil {
            switch x := r.(type) {
            case runtime.Error:
                // 这是运行时错误类型异常
            case error:
                // 普通错误类型异常
            default:
                // 其他类型异常
            }
        }
    }()
    ...
}
```

不过这样做和Go语言简单直接的编程哲学背道而驰了。

## 1.8. 配置开发环境

工欲善其事，必先利其器！Go语言编程对外部的编辑工具要求甚低，但是配置适合自己的开发环境却可以达到事倍功半的效果。本节简单介绍几个作者常用的Go语言编辑器和轻量级集成开发环境。

经过多年的发展完善，目前支持Go语言的开发工具已经很多了。其中LiteIDE是国人visualfc用Qt专门为Go语言开发的跨平台轻量级集成开发环境。在早期的Go语言的核心代码库中也包含了vim/Emacs/Netepad++/Eclipse等工具对Go语言支持的各种插件，目前这些第三方扩展已经从核心库剥离到外部仓库去独立维护。相对完整的IDE或插件列表可以从Go语言的官方wiki页面查看：<https://github.com/golang/go/wiki/IDEsAndTextEditorPlugins>。

对于Windows环境，Go语言纯代码编写的话推荐Notepad++工具，如果需要代码自动补全和调试的话推荐使用微软的Visual Studio Code集成开发环境。如果是Mac OS X用户，可以选择免费的TextMate编辑器，它被誉为macOS下的Notepad++。如果是想基于iPad Pro平台做轻办公，可以选择收费的Textastic应用，它可以完美地配合Working Copy的Git工作流程，同时支持WebDAV协议。在Linux环境，Go语言纯代码编写的话推荐Gtihub Atom工具，如果是命令行的老司机用户也可以配置自己的Vim/Emacs开发环境，调试环境依然推荐Visual Studio Code。

### Windows: Notepad++

Notepad++是Windows操作系统下严肃程序员们编写代码的利器！Notepad++不仅仅免费、体积小（安装程序7+MB）、启动迅速，而且对中文的各种编码支持非常友好，支持众多程序语言的语法高亮，对于正则表达式、函数列表、多工程等特性也有不错的支持。

首先去Notepad++的官网 <http://notepad-plus-plus.org> 下载最新的安装包。然后去<https://github.com/chai2010/notepadplus-go> 下载针对Go语言的配置文件并安装。需要说明的是，对于Go汇编语言用户来说，notepadplus-go是目前唯一支持Go汇编语言语法高亮和函数列表的开发环境。

下面是Go语言的语法高亮预览，其中右侧是Go函数列表：

The screenshot shows the Notepad++ interface with the file `webp.go` open. The code is a Go program that implements a decoder and encoder for WEBP images. It includes copyright notices, imports for the `image` package, and definitions for `GetInfo`, `DecodeGray`, and other functions. On the right side, there is a sidebar titled "函数功能表" (Function List) which displays a hierarchical tree of the functions defined in the file, such as `GetInfo`, `DecodeGray`, `EncodeGray`, etc.

```

1 // Copyright 2014 <chaishushan{AT}gmail.com>. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package webp implements a decoder and encoder for WEBP images.
6 //
7 // WEBP is defined at:
8 // https://developers.google.com/speed/webp/docs/riff_container
9 package webp
10
11 import (
12     "image"
13 )
14
15 func GetInfo(data []byte) (width, height int, hasAlpha bool, err error) {
16     return webpGetInfo(data)
17 }
18
19 func DecodeGray(data []byte) (*image.Gray, error) {
20     pix, w, h, err := webpDecodeGray(data)
21     if err != nil {
22         return
23     }
24     m = &image.Gray{
25         Pix:    pix,
26         Stride: 1 * w,
27         Rect:   image.Rect(0, 0, w, h),
28     }
29     return
30 }
31

```

下面是Go语言汇编的语法高亮预览：其中右侧是汇编函数列表：

```

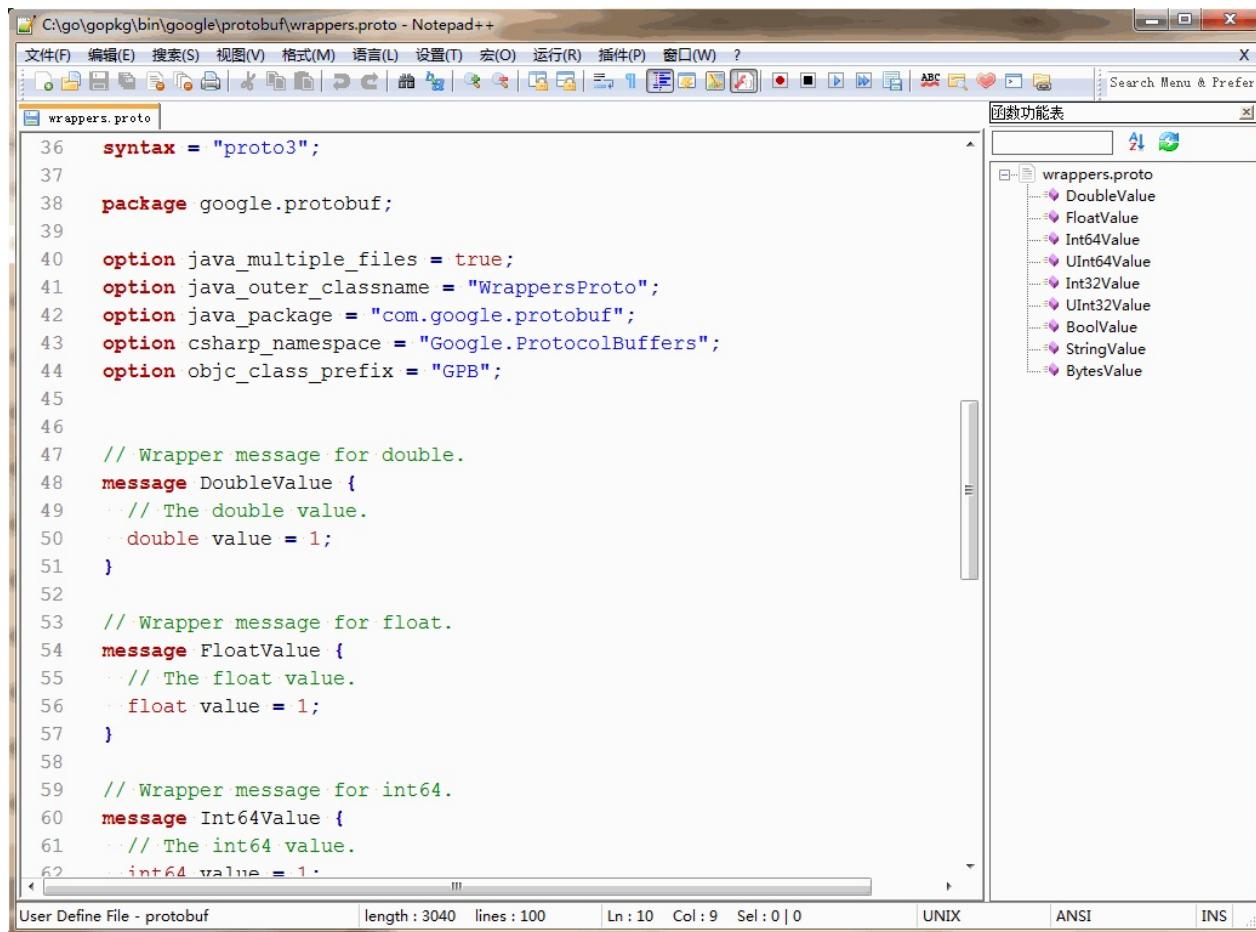
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 #include "go_asm.h"
6 #include "go_tls.h"
7 #include "funcdata.h"
8 #include "textflag.h"
9
10 TEXT runtime·rt0_go(SB),NOSPLIT,$0
11     // copy arguments forward on an even stack
12     MOVQ    DI, AX      // argc
13     MOVQ    SI, BX      // argv
14     SUBQ    $(4*8+7), SP      // 2args 2auto
15     ANDQ    $~15, SP
16     MOVQ    AX, 16(SP)
17     MOVQ    BX, 24(SP)
18
19     // create istack out of the given (operating system) stack
20     // _cgo_init may update stackguard.
21     MOVQ    $runtime·g0(SB), DI
22     LEAQ    (-64*1024+104)(SP), BX
23     MOVQ    BX, g_stackguard0(DI)
24     MOVQ    BX, g_stackguard1(DI)
25     MOVQ    BX, (g_stack+stack_lo)(DI)
26     MOVQ    SP, (g_stack+stack_hi)(DI)
27

```

User Define File - goasm | length : 47622 lines : 2119 | Ln : 10 Col : 35 Sel : 0 | 0 | UNIX | UTF-8 | INS

对于Protobuf或GRPC的用户，可以从 <https://github.com/chai2010/notepadplus-protobuf> 下载相应的插件。

下面是Protobuf的语法高亮预览：



### 配置Notepad++的语法高亮

Notepad++从v6.2版本之后，用户自定义语言文件 `userDefineLang.xml` 改用 `UDL2` 语法，这些的配置文件全部采用的是新的 `UDL2` 的语法。

如果是通过Notepad++安装程序安装的，需要将 `userDefineLang.xml` 文件中的内容添加到 `%APPDATA%\Notepad++\userDefineLang.xml` 文件中，放在 `<NotepadPlus> ... </NotepadPlus>` 标签中间，然后重启Notepad++程序。

如果是从Notepad++ zip/7z压缩包解压绿色安装，配置文件 `userDefineLang.xml` 在解压目录。

### 配置Notepad++函数列表支持

函数列表功能是Notepad++ v6.4新增加的特性，配置方法和语法高亮的配置过程类似。需要注意的是v6.4和v6.5对应的 `<associationMap>...</associationMap>` 配置语法稍有不同，具体请参考 `functionList.xml` 文件中的注释说明。

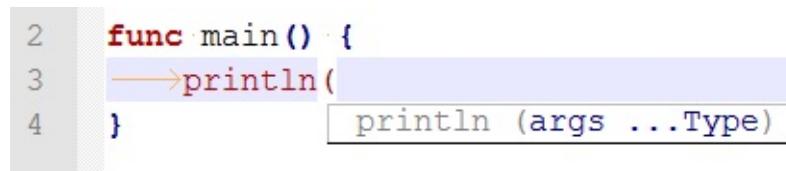
如果是采用Notepad++安装程序安装的，需要将 `functionList.xml` 文件中的内容添加到 `%APPDATA%\Notepad++\functionList.xml` 文件中，放到 `<associationMap> ... </associationMap> 和 <parsers> ... </parsers>` 标签中间，然后重启Notepad++程序。

如果是从Notepad++ zip/7z压缩包解压绿色安装，配置文件 `functionList.xml` 在解压目录。

### Notepad++内置函数的自动补全

Notepad++还支持关键字的自动补全。假设Notepad++安装在 `<DIR>` 目录，将 `go.xml` 文件复制到 `<DIR>\plugins\APIs` 目录下，然后重启Notepad++程序。

下面是内置函数 `println` 自动补全后函数参数提示的预览图：

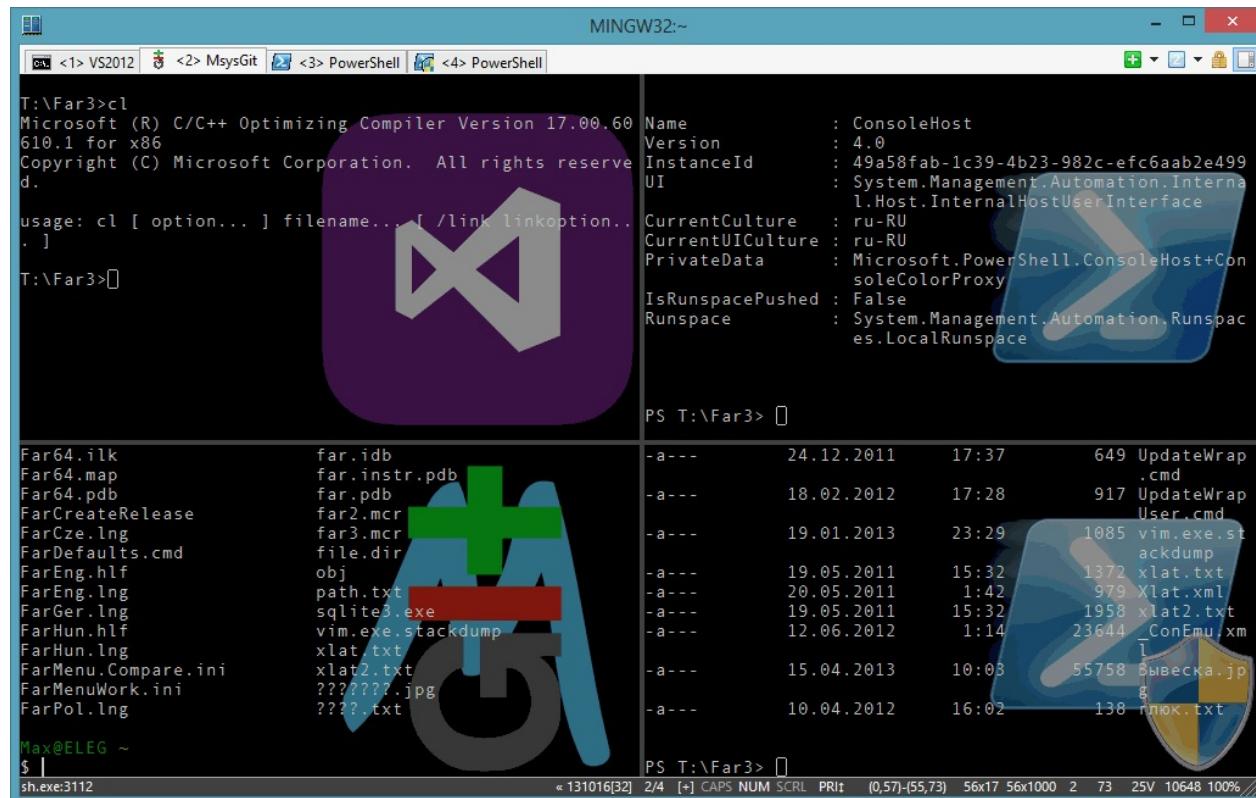


这是一个比较鸡肋的功能，建议用户根据自己需要选择安装。

## 命令行窗口

对于Go语言开发来说，需要经常在命令行运行 `go fmt`、`go test`、`go run x.go` 等辅助工具。虽然Notepad++也可以将这些工具配置成标准的菜单中，但是命令行依然是不可缺少的开发环境。

不过Windows自带的命令行工具比较简陋，不是理想的命令行开发环境。如果读者还没有自己合适命令行环境，可以试试ConEmu这个免费命令行软件。ConEmu支持多标签页窗口，复制粘贴也比较方便。下面是ConEmu的预览图：



ConEmu的主页在：<http://conemu.github.io/>。

## macOS: TextMate

对于macOS平台的用户，免费的轻量级编辑器软件推荐TextMate。TextMate是macOS下的Notepad++工具。支持目录列表，支持Go语言的诸多特性。下面是TextMate的预览图：

The screenshot shows the TextMate interface. On the left is a sidebar with a tree view of files in a 'cgo' directory, including 'c\_error.cc', 'c\_error.h', 'c\_string.cc', 'c\_string.h', 'cgo.go', 'char.go', 'error\_test.go', 'error.go', 'file.go', 'goobj.go', 'README.md', 'string.list.go', 'string\_test.go', 'string.go', 'types.go', and 'void\_pointer.go'. The main editor area displays a Go file named 'goobj.go' with the following code:

```
// Copyright 2016 <chaishushan{AT}gmail.com>. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.

package cgo

import (
    "sync"
)

type ObjectId int32

var refs struct {
    sync.Mutex
    objs map[ObjectId]interface{}
    next ObjectId
}

func init() {
    refs.Lock()
    defer refs.Unlock()

    refs.objs = make(map[ObjectId]interface{})
    refs.next = 1000
}

func NewObjectId(obj interface{}) ObjectId {
    refs.Lock()
    defer refs.Unlock()

    id := refs.next
    refs.next++
    return id
}
```

At the bottom, there are status indicators for tabs, line numbers (Line: 1), Go mode, tab size (Tab Size: 4), and other settings.

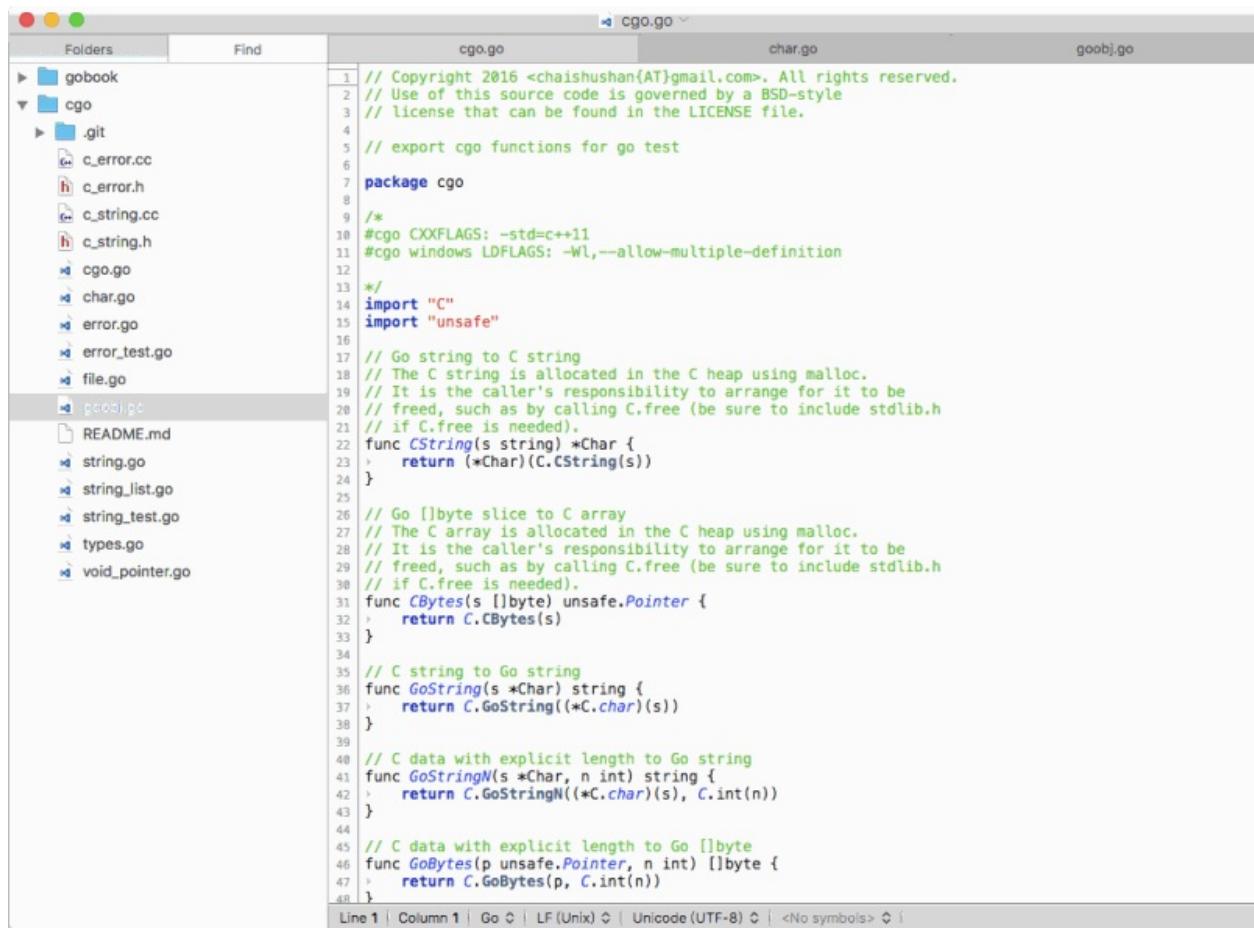
对于iPad Pro用户，目前也有不少编辑软件对Go语言提供了不错的支持。比如Textastic、Code、Coda等，很多都支持iPad和macOS平台的同步，它们一般都是需要单独购买的收费软件。

## iOS: Textastic

Textastic是一款收费应用，它是macOS/iOS下著名的轻量级代码编辑工具，支持包含Go语言在内的多达80多种编程语言的高亮显示。Textastic功能特点有：

- 句法高亮，同时支持80余种语言
- 与TextMate句法定义，主题兼容
- 对HTML、CSS、JavaScript、PHP、C、Objective-C支持自动补全代码
- Symbol list快速导航内容
- 自动保存代码和版本
- iCloud同步 (Mountain Lion only)

在macOS下，Textastic的界面和TextMate非常相似。不过Textastic在左边侧栏提供了基于工程的检索工具。下面是macOS下Textastic的预览图：



因为iOS环境不支持编译和调试，如果需要在iOS环境编写Go程序，首先要解决和其他平台的共享问题。这样可以在iOS环境编写代码，然后在其他电脑上进行编译和测试。

最简单的共享方式是在iCloud的Textastic专有的目录中创建Go语言的工作区目录，然后通过iCloud方案实现和其他平台共享。此外，还可以通过WebDAV标准协议来实现文件的共享，常见的NAS系统都会提供WebDAV协议的共享方式。另外，用Go语言也能很容易实现一个WebDAV的服务器，具体请参考第七章中WebDAV的相关主题。

下面是iPad Pro下Textastic的预览图：

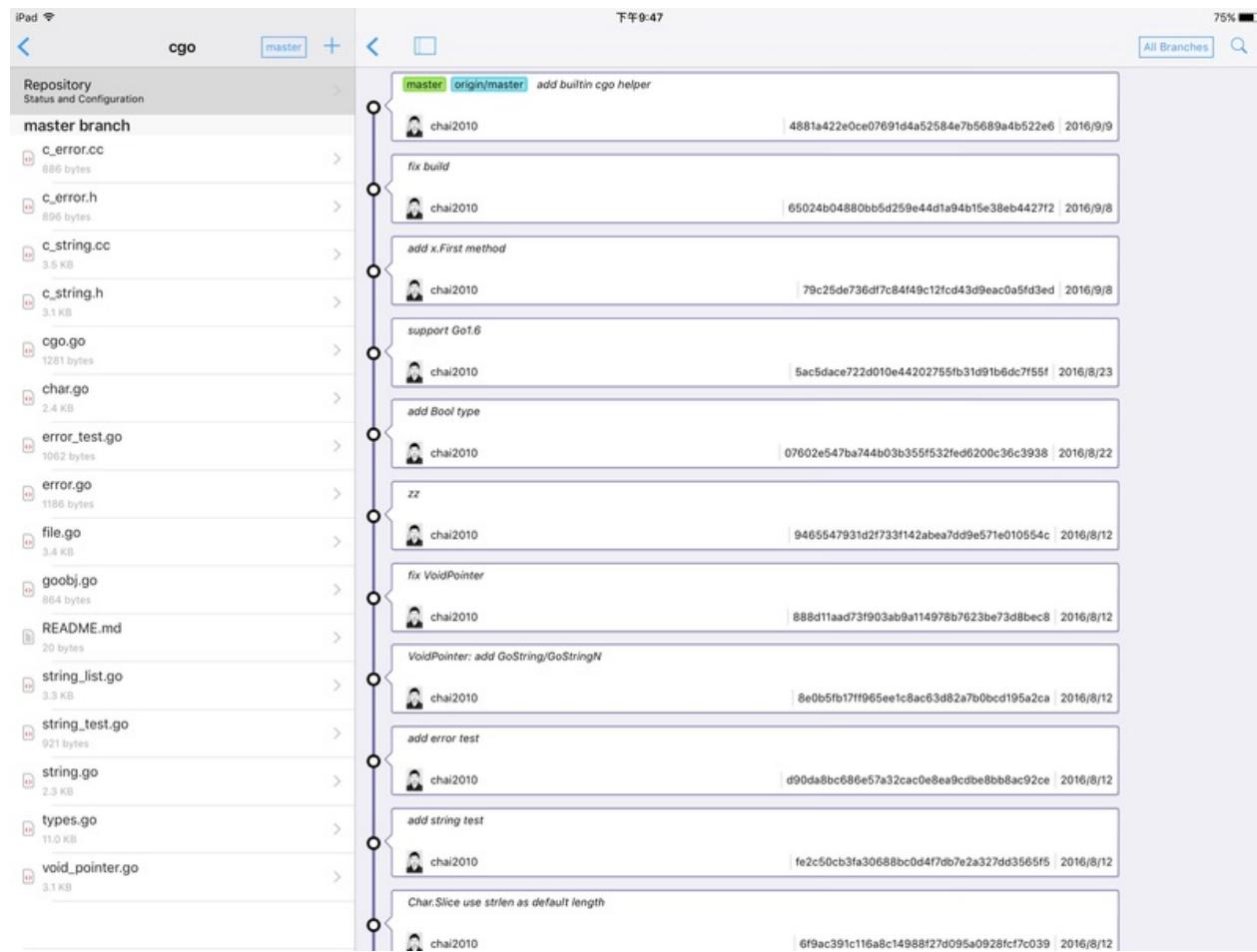
```

1 // Copyright 2016 <chaishushan{AT}gmail.com>. All rights reserved
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // export cgo functions for go test
6
7 package cgo
8
9 /*
10 #cgo CXXFLAGS: -std=c++11
11 #cgo windows LDFLAGS: -Wl,--allow-multiple-definition
12 */
13
14 import "C"
15 import "unsafe"
16
17 // Go string to C string
18 // The C string is allocated in the C heap using malloc.
19 // It is the caller's responsibility to arrange for it to be
20 // freed, such as by calling C.free (be sure to include stdlib.h
21 // if C.free is needed).
22 func CString(s string) *Char {
23     return (*Char)(C.CString(s))
24 }
25
26 // Go []byte slice to C array
27 // The C array is allocated in the C heap using malloc.
28 // It is the caller's responsibility to arrange for it to be
29 // freed, such as by calling C.free (be sure to include stdlib.h
30 // if C.free is needed).
31 func CBytes(s []byte) unsafe.Pointer {
32     return C.CBytes(s)
33 }
34
35 // C string to Go string
36 func GoString(s *Char) string {
37     return C.GoString((*C.char)(s))
38 }
39
40 // C data with explicit length to Go string
41 func GoStringN(s *Char, n int) string {
42     return C.GoStringN((*C.char)(s), C.int(n))
43 }
44
45 // C data with explicit length to Go []byte
46 func GoBytes(p unsafe.Pointer, n int) []byte {
47     return C.GoBytes(p, C.int(n))
48 }
49

```

如果Go语言代码是放在Git服务器中，可以通过Working Copy应用将仓库克隆到iOS中，然后再Textastic中通过iOS协议打开工作区文件。编辑完成之后，在通过Working Copy将修改提交到中心仓库中。

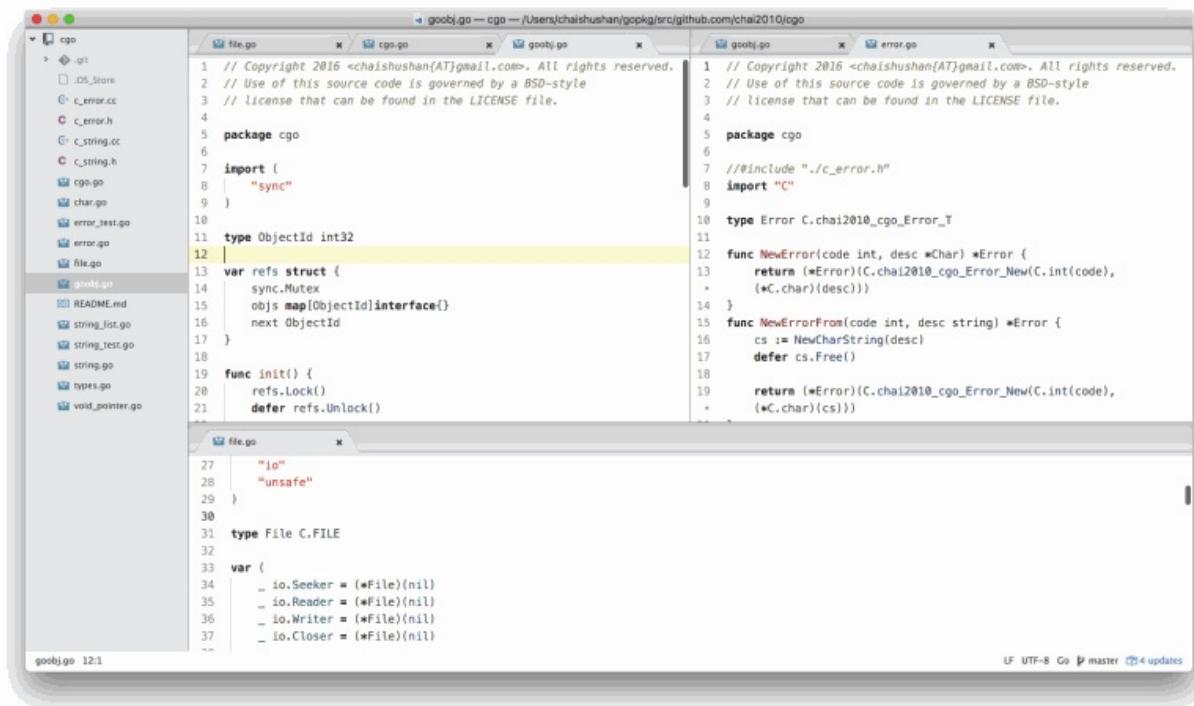
下面是iPad Pro下Working Copy查看Git更新日志的预览图：



## 跨平台编辑器: Github Atom

Github Atom是Github专门为程序员推出的一个跨平台文本编辑器。具有简洁和直观的图形用户界面，内置支持Go语言语法高亮。同时Github Atom支持宏、自动完成分屏功能，同时集成了文件管理器，对于macOS和Linux用户来说是一个优秀的Go语言编辑器。

Github Atom的预览图：



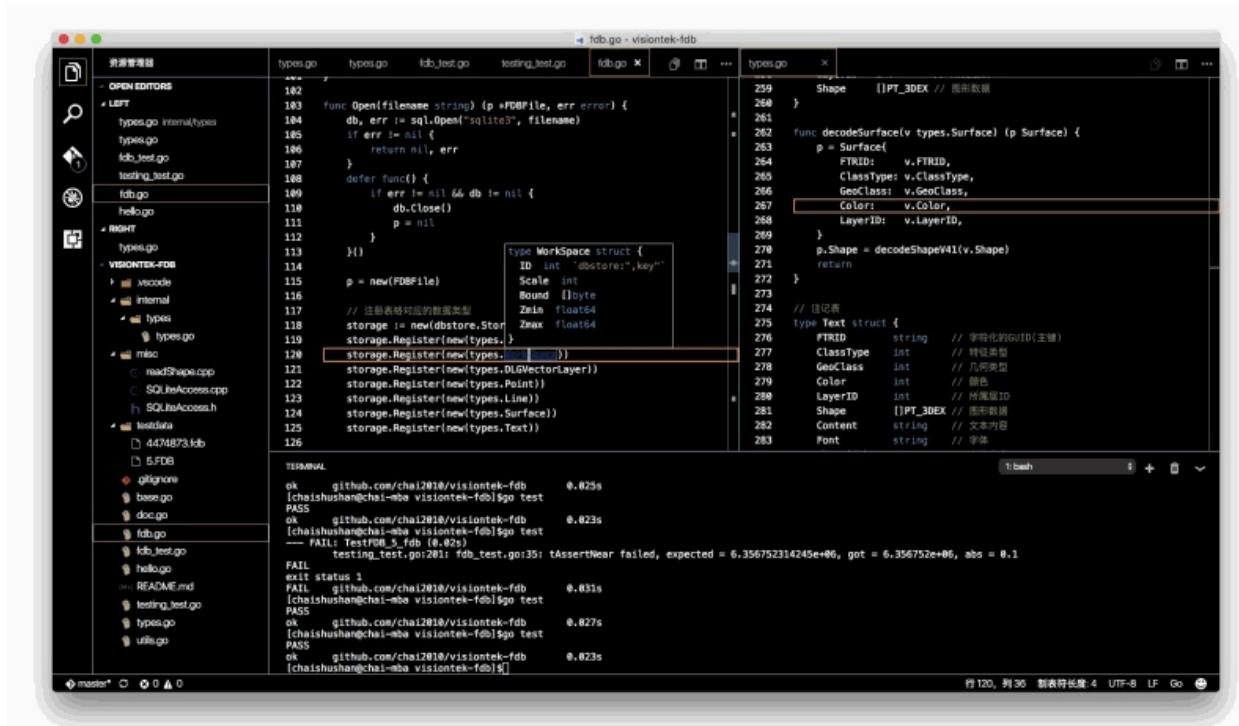
Github Atom作为一个Go语言编辑器，不足之处是没有Go汇编语言的高亮显示插件。而且Github Atom对于大文件的支持，性能不是很好。

# 跨平台 IDE: Visual Studio Code

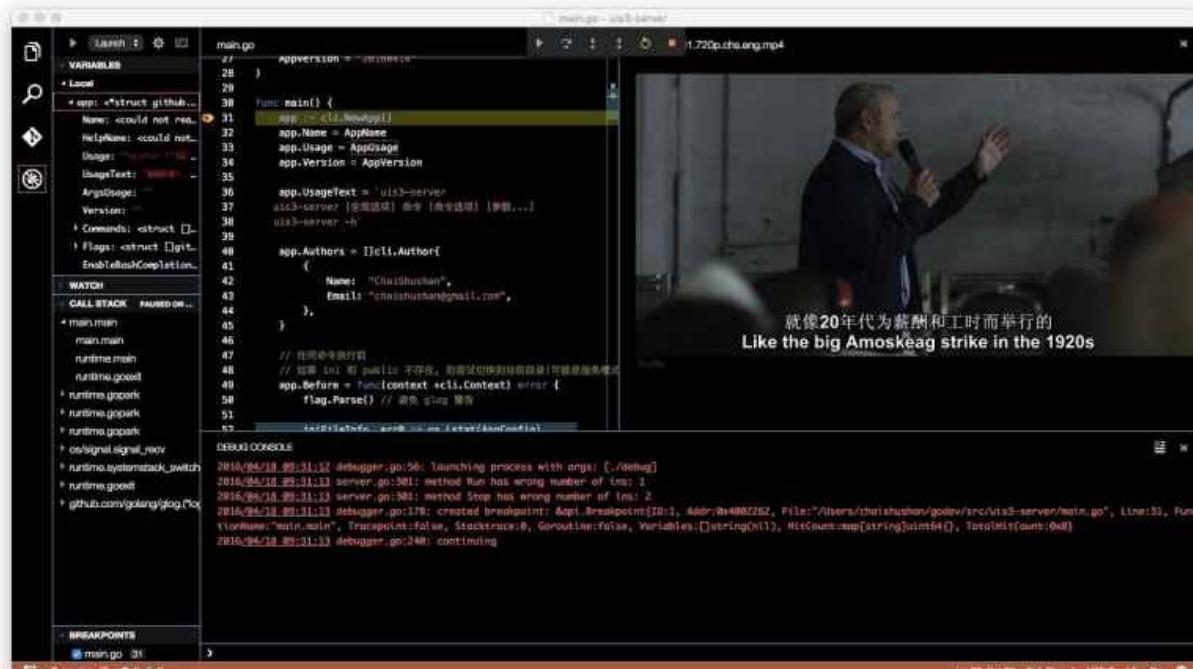
Visual Studio Code是微软推出的轻量级跨平台集成开发环境，简称VSCode。VSCode最初目的是支持JavaScript和TypeScript开发，但是它逐步增加了第三方编程语言的支持，目前它已经可以说是最完美的Go语言集成开发环境了。

VSCODE虽然是基于Github Atom而来，不过VSCODE支持Go语言的代码自动补全和调试功能，因此已经超越Github Atom单作为编辑器的定位，是一个轻量级的集成开发环境。VSCODE对于大文件的支持也比Github Atom优秀很多。

下面是用VScode打开的Go语言工程的预览图：



因为，VSCode和Github Atom都是采用的Chrome核心，它不仅仅能编辑显示代码，还可以用来显示网页查看图像，甚至可以在一个分屏窗口中播放视频文件：



VSCode安装Go语言插件中，默认的很多参数设置比较严格。比如，默认会使用 `golint` 来严格检查代码是否符合编码规范，对于git工程启动时还会自动获取和刷新。对于一般的Go语言代码来说，`golint` 检测过于严格，很难完全通过（Go语言标准库也无法完全通过），从而导致每次保存时都会提示很多干扰信息。当然，对相对稳定的程序定期做 `golint` 检查也是有必要，它的信息可以作为我们改进代码的参考。同样的，如果git仓库有密码认证的话，VSCode在启动的时候总是弹出输入密码的对话框。

我们可以在工程目录的 `.vscode/settings.json` 配置文件中定制这些选项。下面配置是强制在保存的时候采用 `gofmt` 格式化代码，并且关闭保存时 `golint` 检查。同时在VSCode刚启动的时候，禁止Git自动刷新和获取操作。

```
// 将设置放入此文件中以覆盖默认值和用户设置。
{
    // Pick 'gofmt', 'goimports' or 'goreturns' to run on format.
    "go.formatTool": "gofmt",

    // [EXPERIMENTAL] Run formatting tool on save.
    "go.formatOnSave": true,

    // Run 'golint' on save.
    "go.lintOnSave": false,
    "git.autorefresh": false,
    "git.autofetch": false
}
```

VSCode作为一个专业的Go语言集成开发环境，稍显不足之处是没有Go汇编语言的高亮显示插件。

## 第二章 CGO编程

C/C++经过几十年的发展，已经积累了庞大的软件资产，它们很多久经考验而且性能已经足够优化。Go语言必须能够站在C/C++这个巨人的肩膀之上，有了海量的C/C++软件资产兜底之后，我们才可以放心愉快地用Go语言编程。C语言作为一个通用语言，很多库会选择提供一个C兼容的API，然后用其他不同的编程语言实现。Go语言通过自带的一个叫CGO的工具来支持C语言函数调用，同时我们可以用Go语言导出C动态库接口给其它语言使用。本章主要讨论CGO编程中涉及的一些问题。

## 2.1. 你好, CGO!

在第一章的“Hello, World 的革命”一节中，我们已经见过一个CGO程序。这一节我们将重新给出三个版本的CGO实现，来简单展示CGO的用法。

### 基于C标准库

第一章那个CGO程序还不够简单，我们现在来看看更简单的版本：

```
package main

//#include <stdio.h>
import "C"

func main() {
    C.puts(C.CString("Hello, World\n"))
}
```

我们通过 `import "C"` 语句启用CGO特性，同时包含C语言的 `<stdio.h>` 头文件。然后通过CGO包的 `ccstring` 函数将Go语言字符串转为C语言字符串，最后调用C语言的 `c.puts` 函数向标准输出窗口打印转换后的C字符串。

相比“Hello, World 的革命”一节中的CGO程序最大的不同是：我们没有在程序退出前释放 `ccstring` 创建的C语言字符串；还有我们改用 `puts` 函数直接向标准输出打印，之前是采用 `fputs` 向标准输出打印。

没有释放使用 `ccstring` 创建的C语言字符串会导致内存泄露。但是对于这个小程序来说，这样是没有问题的，因为程序退出后操作系统会自动回收程序的所有资源。

### 使用自己的C函数

前面我们使用了标准库中已有的函数。现在我们先自定义一个叫 `SayHello` 的C函数来实现打印，然后从Go语言环境中调用这个 `SayHello` 函数：

```

package main

/*
#include <stdio.h>

static void SayHello(const char* s) {
    puts(s);
}
*/
import "C"

func main() {
    C.SayHello(C.CString("Hello, World\n"))
}

```

除了 `SayHello` 函数是我们自己实现的之外，其它的部分和前面的例子基本相似。

我们也可以将 `SayHello` 函数放到当前目录下的一个C语言源文件中（后缀名必须是 `.c`）。因为是编写在独立的C文件中，为了允许外部引用，所以需要去掉函数的 `static` 修饰符。

```

// hello.c

#include <stdio.h>

void SayHello(const char* s) {
    puts(s);
}

```

然后在CGO部分先声明 `SayHello` 函数，其它部分不变：

```

package main

//void SayHello(const char* s);
import "C"

func main() {
    C.SayHello(C.CString("Hello, World\n"))
}

```

既然 `SayHello` 函数已经放到独立的C文件中了，我们自然可以将对应的C文件编译打包为静态库或动态库文件供使用。如果是以静态库或动态库方式引用 `SayHello` 函数的话，需要将对应的C源文件移出当前目录（CGO构建程序会自动构建当前目录下的C源文件，从而导致C函数名冲突）。关于静态库等细节将在稍后章节讲解。

## 用 Go 来实现 C 函数

其实CGO不仅仅用于Go语言中调用C语言函数，还可以用于导出Go语言函数给C语言函数调用。

```
package main

/*
#include <stdio.h>

void cgoPuts(char* s);

static void SayHello(const char* s) {
    cgoPuts((char*)(s));
}

import "C"

func main() {
    C.SayHello(C.CString("Hello, World\n"))
}

//export cgoPuts
func cgoPuts(s *C.char) {
    fmt.Println(C.GoString(s))
}
```

我们通过CGO的 `//export cgoPuts` 指令将Go语言实现的函数 `cgoPuts` 导出给C语言函数使用。然后在C语言版本的 `SayHello` 函数中，用 `cgoPuts` 替换之前的 `puts` 函数调用。在使用之前，同样要先声明 `cgoPuts` 函数。

需要主要的是，这里其实有两个版本的 `cgoPuts` 函数：一个Go语言环境的；另一个是C语言环境的。在C语言环境中，`SayHello` 调用的也是C语言环境的 `cgoPuts` 函数；这是CGO自动生成的桥接函数，内部会调用Go语言环境的 `cgoPuts` 函数。因此，我们也可以直接在Go语言环境中调用C语言环境的 `cgoPuts` 函数。

现在我们可以改用Go语言重新实现C语言接口的 `SayHello` 函数，然后在 `main` 函数中还是和之前一样调用 `C.SayHello` 实现输出：

```

package main

//void SayHello(char* s);
import "C"

import (
    "fmt"
)

func main() {
    C.SayHello(C.CString("Hello, World\n"))
}

//export SayHello
func SayHello(s *C.char) {
    fmt.Println(C.GoString(s))
}

```

现在中国版本的CGO代码中C语言代码的比例已经很少了，但是我们依然可以进一步以Go语言的思维来提炼我们的CGO代码。通过分析可以发现 `SayHello` 函数的参数如果可以直接使用 Go字符串是最直接的。在 Go1.10 中 CGO 新增加了一个 `_GoString_` 预定义的 C 语言类型，用来表示 Go 语言字符串。下面是改进后的代码：

```

// +build go1.10

package main

//void SayHello(_GoString_ s);
import "C"

import (
    "fmt"
)

func main() {
    C.SayHello("Hello, World\n")
}

//export SayHello
func SayHello(s string) {
    fmt.Println(s)
}

```

虽然看起来全部是 Go 语言代码，但是执行的时候是先从 Go 语言的 `main` 函数，到 CGO 自动生成的 C 语言版本 `SayHello` 桥接函数，最后又回到了 Go 语言环境的 `SayHello` 函数。虽然看起来有点绕，但 CGO 确实是这样运行的。

需要注意的是，CGO 导出 Go 语言函数时，函数参数中不再支持 C 语言中 `const` 修饰符。



## 2.2. 基础类型转换

最初CGO是为了达到方便从Go语言函数调用C语言函数以复用C语言资源这一目的而出现的(因为C语言还会涉及回调函数，自然也会涉及到从C语言函数调用Go语言函数)。现在，它已经演变为C语言和Go语言双向通讯的桥梁。要想利用好CGO特性，自然需要了解此二语言类型之间的转换规则，这是本节要讨论的问题。

### 数值类型

在Go语言中访问C语言的符号时，一般是通过虚拟的“C”包访问，比如 `c.int` 对应C语言的 `int` 类型。有些C语言的类型是由多个关键字组成，但通过虚拟的“C”包访问C语言类型时名称部分不能有空格字符，比如 `unsigned int` 不能直接通过 `c.unsigned int` 访问。因此CGO为C语言的基础数值类型都提供了相应转换规则，比如 `c.uint` 对应C语言的 `unsigned int`。

Go语言中数值类型和C语言数据类型基本上是相似的，以下是它们的对应关系表。

C语言类型	CGO类型	Go语言类型
<code>char</code>	<code>C.char</code>	<code>byte</code>
<code>singed char</code>	<code>C.schar</code>	<code>int8</code>
<code>unsigned char</code>	<code>C uchar</code>	<code>uint8</code>
<code>short</code>	<code>C.short</code>	<code>int16</code>
<code>unsigned short</code>	<code>C.short</code>	<code>uint16</code>
<code>int</code>	<code>C.int</code>	<code>int32</code>
<code>unsigned int</code>	<code>C.uint</code>	<code>uint32</code>
<code>long</code>	<code>C.long</code>	<code>int32</code>
<code>unsigned long</code>	<code>C.ulong</code>	<code>uint32</code>
<code>long long int</code>	<code>C.longlong</code>	<code>int64</code>
<code>unsigned long long int</code>	<code>C.ulonglong</code>	<code>uint64</code>
<code>float</code>	<code>C.float</code>	<code>float32</code>
<code>double</code>	<code>C.double</code>	<code>float64</code>
<code>size_t</code>	<code>C.size_t</code>	<code>uint</code>

需要注意的是，虽然在C语言中 `int`、`short` 等类型没有明确定义内存大小，但是在CGO中它们的内存大小是确定的。在CGO中，C语言的 `int` 和 `long` 类型都是对应4个字节的内存大小，`size_t` 类型可以当作Go语言 `uint` 无符号整数类型对待。

CGO中，虽然C语言的 `int` 固定为4字节的大小，但是Go语言自己的 `int` 和 `uint` 却在32位和64位系统下分别对应4个字节和8个字节大小。如果需要在C语言中访问Go语言的 `int` 类型，可以通过 `GoInt` 类型访问，`GoInt` 类型在CGO工具生成的 `_cgo_export.h` 头文件中定义。其实在 `_cgo_export.h` 头文件中，每个基本的Go数值类型都定义了对应的C语言类型，它们一般都是以单词Go为前缀。下面是64位环境下，`_cgo_export.h` 头文件生成的Go数值类型的定义，其中 `GoInt` 和 `GoUint` 类型分别对应 `GoInt64` 和 `GoUint64`：

```
typedef signed char GoInt8;
typedef unsigned char GoUint8;
typedef short GoInt16;
typedef unsigned short GoUint16;
typedef int GoInt32;
typedef unsigned int GoUint32;
typedef long long GoInt64;
typedef unsigned long long GoUint64;
typedef GoInt64 GoInt;
typedef GoUint64 GoUint;
typedef float GoFloat32;
typedef double GoFloat64;
```

除了 `GoInt` 和 `GoUint` 之外，我们并不推荐直接访问 `GoInt32`、`GoInt64` 等类型。更好的做法是通过C语言的C99标准引入的 `<stdint.h>` 头文件。为了提高C语言的可移植性，在 `<stdint.h>` 文件中，不但每个数值类型都提供了明确内存大小，而且和Go语言的类型命名更加一致。

C语言类型	CGO类型	Go语言类型
<code>int8_t</code>	<code>C.int8_t</code>	<code>int8</code>
<code>uint8_t</code>	<code>C.uint8_t</code>	<code>uint8</code>
<code>int16_t</code>	<code>C.int16_t</code>	<code>int16</code>
<code>uint16_t</code>	<code>C.uint16_t</code>	<code>uint16</code>
<code>int32_t</code>	<code>C.int32_t</code>	<code>int32</code>
<code>uint32_t</code>	<code>C.uint32_t</code>	<code>uint32</code>
<code>int64_t</code>	<code>C.int64_t</code>	<code>int64</code>
<code>uint64_t</code>	<code>C.uint64_t</code>	<code>uint64</code>

前文说过，如果C语言的类型是由多个关键字组成，则无法通过虚拟的“C”包直接访问(比如C语言的 `unsigned short` 不能直接通过 `C,unsigned short` 访问)。但是，在 `<stdint.h>` 中通过使用C语言的 `typedef` 关键字将 `unsigned short` 重新定义为 `uint16_t` 这样一个单词的类型后，我们就可以通过 `C.uint16_t` 访问原来的 `unsigned short` 类型了。对于比较复杂的C语言类型，推荐使用 `typedef` 关键字提供一个规则的类型命名，这样更利于在CGO中访问。

## Go 字符串和切片

在CGO生成的 `_cgo_export.h` 头文件中还会为Go语言的字符串、切片、字典、接口和管道等特有的数据类型生成对应的C语言类型：

```
typedef struct { const char *p; GoInt n; } GoString;
typedef void *GoMap;
typedef void *GoChan;
typedef struct { void *t; void *v; } GoInterface;
typedef struct { void *data; GoInt len; GoInt cap; } GoSlice;
```

不过需要注意的是，其中只有字符串和切片在CGO中有一定的使用价值，因为此二者可以在Go调用C语言函数时马上使用；而CGO并未针对其他的类型提供相关的辅助函数，且Go语言特有的内存模型导致我们无法保持这些由Go语言管理的内存指针，所以它们C语言环境并无使用的价值。

在导出的C语言函数中我们可以直接使用Go字符串和切片。假设有以下两个导出函数：

```
//export helloString
func helloString(s string) {}

//export helloSlice
func helloSlice(s []byte) {}
```

CGO生成的 `_cgo_export.h` 头文件会包含以下的函数声明：

```
extern void helloString(GoString p0);
extern void helloSlice(GoSlice p0);
```

不过需要注意的是，如果使用了GoString类型则会对 `_cgo_export.h` 头文件产生依赖，而这个头文件是动态输出的。

Go1.10针对Go字符串增加了一个 `_GoString_` 预定义类型，可以降低在cgo代码中可能对 `_cgo_export.h` 头文件产生的循环依赖的风险。我们可以调整helloString函数的C语言声明为：

```
extern void helloString(_GoString_ p0);
```

因为 `_GoString_` 是预定义类型，我们无法通过此类型直接访问字符串的长度和指针等信息。Go1.10同时也增加了以下两个函数用于获取字符串结构中的长度和指针信息：

```
size_t _GoStringLen(_GoString_ s);
const char *_GoStringPtr(_GoString_ s);
```

更严谨的做法是为C语言函数接口定义严格的头文件，然后基于稳定的头文件实现代码。

## 结构体、联合、枚举类型

C语言的结构体、联合、枚举类型不能作为匿名成员被嵌入到Go语言的结构体中。在Go语言中，我们可以通过 `c.struct_xxx` 来访问C语言中定义的 `struct xxx` 结构体类型。结构体的内存布局按照C语言的通用对齐规则，在32位Go语言环境C语言结构体也按照32位对齐规则，在64位Go语言环境按照64位的对齐规则。对于指定了特殊对齐规则的结构体，无法在CGO中访问。

结构体的简单用法如下：

```
/*
struct A {
    int i;
    float f;
};

import "C"
import "fmt"

func main() {
    var a C.struct_A
    fmt.Println(a.i)
    fmt.Println(a.f)
}
```

如果结构体的成员名字中碰巧是Go语言的关键字，可以通过在成员名开头添加下划线来访问：

```
/*
struct A {
    int type; // type 是 Go 语言的关键字
};

import "C"
import "fmt"

func main() {
    var a C.struct_A
    fmt.Println(a._type) // _type 对应 type
}
```

但是如果有2个成员：一个是以Go语言关键字命名，另一个刚好是以下划线和Go语言关键字命名，那么以Go语言关键字命名的成员将无法访问（被屏蔽）：

```
/*
struct A {
    int type; // type 是 Go 语言的关键字
    float _type; // 将屏蔽CGO对 type 成员的访问
};

import "C"
import "fmt"

func main() {
    var a C.struct_A
    fmt.Println(a._type) // _type 对应 type
}
```

C语言结构体中位字段对应的成员无法在Go语言中访问，如果需要操作位字段成员，需要通过在C语言中定义辅助函数来完成。对应零长数组的成员，无法在Go语言中直接访问数组的元素，但其中零长的数组成员所在位置的偏移量依然可以通过 `unsafe.Offsetof(a.arr)` 来访问。

```
/*
struct A {
    int size: 10; // 位字段无法访问
    float arr[]; // 零长的数组也无法访问
};

import "C"
import "fmt"

func main() {
    var a C.struct_A
    fmt.Println(a.size) // 错误：位字段无法访问
    fmt.Println(a.arr) // 错误：零长的数组也无法访问
}
```

在C语言中，我们无法直接访问Go语言定义的结构体类型。

对于联合类型，我们可以通过 `C.union_xxx` 来访问C语言中定义的 `union xxx` 类型。但是Go语言中并不支持C语言联合类型，它们会被转为对应大小的字节数组。

```

/*
#include <stdint.h>

union B1 {
    int i;
    float f;
};

union B2 {
    int8_t i8;
    int64_t i64;
};
*/
import "C"
import "fmt"

func main() {
    var b1 C.union_B1;
    fmt.Printf("%T\n", b1) // [4]uint8

    var b2 C.union_B2;
    fmt.Printf("%T\n", b2) // [8]uint8
}

```

如果需要操作C语言的联合类型变量，一般有三种方法：第一种是在C语言中定义辅助函数；第二种是通过Go语言的"encoding/binary"手工解码成员(需要注意大端小端问题)；第三种是使用 `unsafe` 包强制转型为对应类型(这是性能最好的方式)。下面展示通过 `unsafe` 包访问联合类型成员的方式：

```

/*
#include <stdint.h>

union B {
    int i;
    float f;
};
*/
import "C"
import "fmt"

func main() {
    var b C.union_B;
    fmt.Println("b.i:", (*C.int)(unsafe.Pointer(&b)))
    fmt.Println("b.f:", (*C.float)(unsafe.Pointer(&b)))
}

```

虽然 `unsafe` 包访问最简单、性能也最好，但是对于有嵌套联合类型的情况处理会导致问题复杂化。对于复杂的联合类型，推荐通过在C语言中定义辅助函数的方式处理。

对于枚举类型，我们可以通过 `C.enum_xxx` 来访问C语言中定义的 `enum xxx` 结构体类型。

```
/*
enum C {
    ONE,
    TWO,
}
*/
import "C"
import "fmt"

func main() {
    var c C.enum_C = C.TWO
    fmt.Println(c)
    fmt.Println(C.ONE)
    fmt.Println(C.TWO)
}
```

在C语言中，枚举类型底层对应 `int` 类型，支持负数类型的值。我们可以通  
过 `C.ONE`、`C.TWO` 等直接访问定义的枚举值。

## 数组、字符串和切片

在C语言中，数组名其实对应于一个指针，指向特定类型特定长度的一段内存，但是这个指针不能被修改；当把数组名传递给一个函数时，实际上传递的是数组第一个元素的地址。为了讨论方便，我们将一段特定长度的内存统称为数组。C语言的字符串是一个`char`类型的数组，字符串的长度需要根据表示结尾的**NULL**字符的位置确定。C语言中没有切片类型。

在Go语言中，数组是一种值类型，而且数组的长度是数组类型的一个部分。Go语言字符串对应一段长度确定的只读`byte`类型的内存。Go语言的切片则是一个简化版的动态数组。

Go语言和C语言的数组、字符串和切片之间的相互转换可以简化为Go语言的切片和C语言中指向一定长度内存的指针之间的转换。

CGO的C虚拟包提供了以下一组函数，用于Go语言和C语言之间数组和字符串的双向转换：

```

// Go string to C string
// The C string is allocated in the C heap using malloc.
// It is the caller's responsibility to arrange for it to be
// freed, such as by calling C.free (be sure to include stdlib.h
// if C.free is needed).
func C.CString(string) *C.char

// Go []byte slice to C array
// The C array is allocated in the C heap using malloc.
// It is the caller's responsibility to arrange for it to be
// freed, such as by calling C.free (be sure to include stdlib.h
// if C.free is needed).
func C.CBytes([]byte) unsafe.Pointer

// C string to Go string
func C.GoString(*C.char) string

// C data with explicit length to Go string
func C.GoStringN(*C.char, C.int) string

// C data with explicit length to Go []byte
func C.GoBytes(unsafe.Pointer, C.int) []byte

```

其中 `C.CString` 针对输入的 Go 字符串，克隆一个 C 语言格式的字符串；返回的字符串由 C 语言的 `malloc` 函数分配，不使用时需要通过 C 语言的 `free` 函数释放。`C.CBytes` 函数的功能和 `C.CString` 类似，用于从输入的 Go 语言字节切片克隆一个 C 语言版本的字节数组，同样返回的数组需要在合适的时候释放。`C.GoString` 用于将从 NULL 结尾的 C 语言字符串克隆一个 Go 语言字符串。`C.GoStringN` 是另一个字符数组克隆函数。`C.GoBytes` 用于从 C 语言数组，克隆一个 Go 语言字节切片。

该组辅助函数都是以克隆的方式运行。当 Go 语言字符串和切片向 C 语言转换时，克隆的内存由 C 语言的 `malloc` 函数分配，最终可以通过 `free` 函数释放。当 C 语言字符串或数组向 Go 语言转换时，克隆的内存由 Go 语言分配管理。通过该组转换函数，转换前和转换后的内存依然在各自的语言环境中，它们并没有跨越 Go 语言和 C 语言。克隆方式实现转换的优点是接口和内存管理都很简单，缺点是克隆需要分配新的内存和复制操作都会导致额外的开销。

在 `reflect` 包中有字符串和切片的定义：

```

type StringHeader struct {
    Data uintptr
    Len   int
}

type SliceHeader struct {
    Data uintptr
    Len   int
    Cap   int
}

```

如果不希望单独分配内存，可以在Go语言中直接访问C语言的内存空间：

```

/*
static char arr[10];
static char *s = "Hello";
*/
import "C"
import "fmt"

func main() {
    // 通过 reflect.SliceHeader 转换
    var arr0 []byte
    var arr0Hdr = (*reflect.SliceHeader)(unsafe.Pointer(&arr0))
    arr0Hdr.Data = uintptr(unsafe.Pointer(&C.arr[0]))
    arr0Hdr.Len = 10
    arr0Hdr.Cap = 10

    // 通过切片语法转换
    arr1 := (*[31]byte)(unsafe.Pointer(&C.arr[0]))[:10:10]

    var s0 string
    var s0Hdr := (*reflect.StringHeader)(unsafe.Pointer(&s0))
    s0Hdr.Data = uintptr(unsafe.Pointer(C.s))
    s0Hdr.Len = int(C.strlen(C.s))

    sLen := int(C.strlen(C.s))
    s1 := string((*[31]byte)(unsafe.Pointer(&C.s[0]))[:sLen:sLen])
}

```

因为Go语言的字符串是只读的，用户需要自己保证Go字符串在使用期间，底层对应的C字符串内容不会发生变化、内存不会被提前释放掉。

在CGO中，会为字符串和切片生成和上面结构对应的C语言版本的结构体：

```

typedef struct { const char *p; GoInt n; } GoString;
typedef struct { void *data; GoInt len; GoInt cap; } GoSlice;

```

在C语言中可以通过 `GoString` 和 `Goslice` 来访问Go语言的字符串和切片。如果是Go语言中数组类型，可以将数组转为切片后再行转换。如果字符串或切片对应的底层内存空间由Go语言的运行时管理，那么在C语言中不能长时间保存Go内存对象。

关于CGO内存模型的细节在稍后章节中会详细讨论。

## 指针间的转换

在C语言中，不同类型的指针是可以显式或隐式转换的，如果是隐式只是会在编译时给出一些警告信息。但是Go语言对于不同类型的转换非常严格，任何C语言中可能出现的警告信息在Go语言中都可能是错误！指针是C语言的灵魂，指针间的自由转换也是cgo代码中经常要解决的第一个重要的问题。

在Go语言中两个指针的类型完全一致则不需要转换可以直接通用。如果一个指针类型是用`type`命令在另一个指针类型基础之上构建的，换言之两个指针底层是相同完全结构的指针，那么我们可以通过直接强制转换语法进行指针间的转换。但是cgo经常要面对的是2个完全不同类型的指针间的转换，原则上这种操作在纯Go语言代码是严格禁止的。

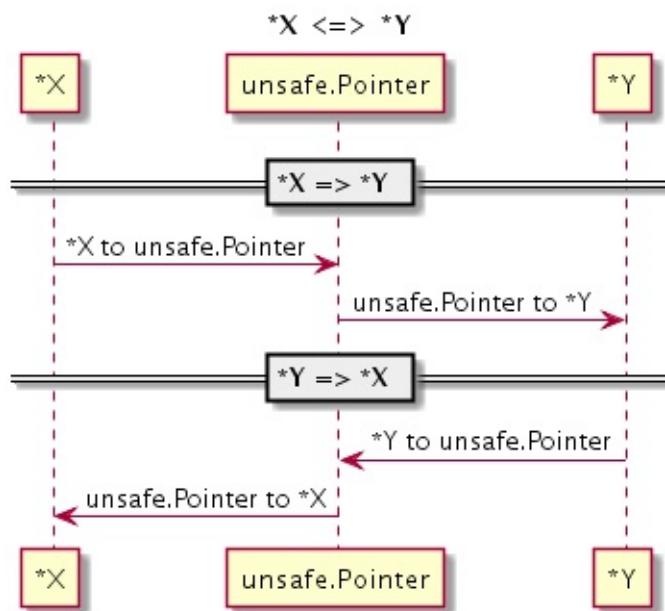
cgo存在的一个目的就是打破Go语言的禁制，恢复C语言应有的指针的自由转换和指针运算。以下代码演示了如何将X类型的指针转化为Y类型的指针：

```
var p *X
var q *Y

q = (*Y)(unsafe.Pointer(p)) // *X => *Y
p = (*X)(unsafe.Pointer(q)) // *Y => *X
```

为了实现X类型指针到Y类型指针的转换，我们需要借助 `unsafe.Pointer` 作为中间桥接类型实现不同类型指针之间的转换。`unsafe.Pointer` 指针类型类似C语言中的 `void*` 类型的指针。

下面是指针间的转换流程的示意图：



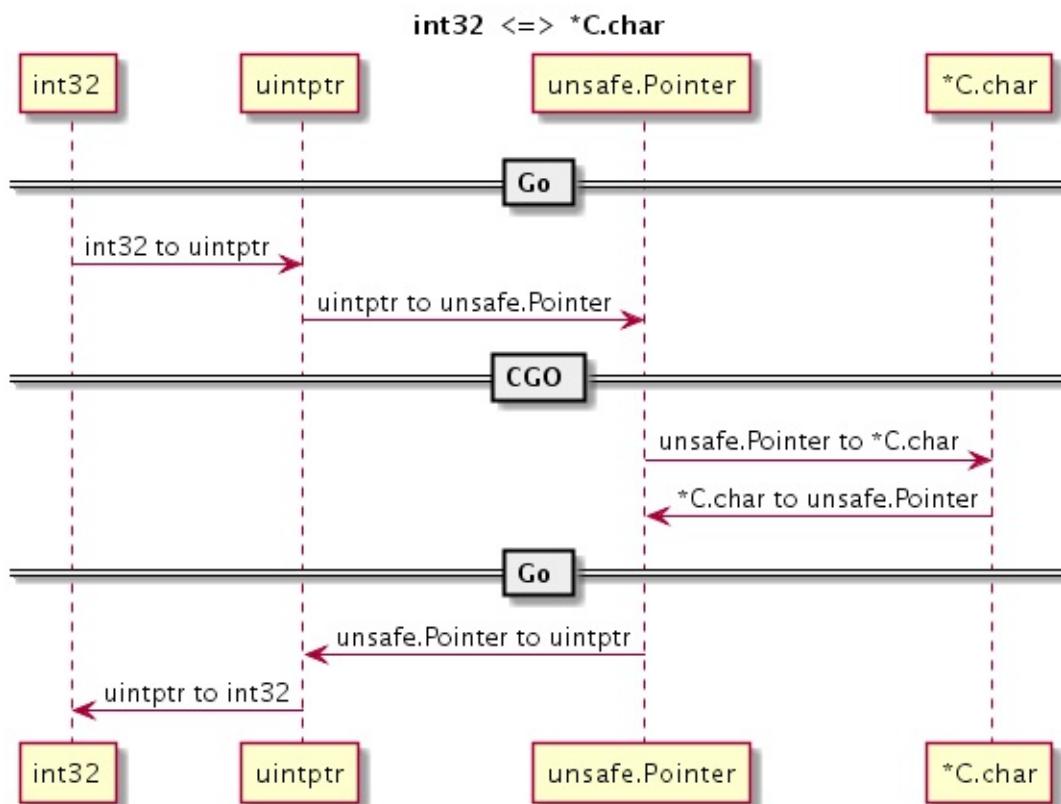
任何类型的指针都可以通过强制转换为 `unsafe.Pointer` 指针类型去掉原有的类型信息，然后再重新赋予新的指针类型而达到指针间的转换的目的。

## 数值和指针的转换

不同类型指针间的转换看似复杂，但是在cgo中已经算是比较简单的了。在C语言中经常遇到用普通数值表示指针的场景，也就是说如何实现数值和指针的转换也是cgo需要面对的一个问题。

为了严格控制指针的使用，Go语言禁止将数值类型直接转为指针类型！不过，Go语言针对 `unsafe.Pointer` 指针类型特别定义了一个`uintptr`类型。我们可以`uintptr`为中介，实现数值类型到 `unsafe.Pointer` 指针类型到转换。再结合前面提到的方法，就可以实现数值和指针的转换了。

下面流程图演示了如何实现`int32`类型到C语言的 `char*` 字符串指针类型的相互转换：



转换分为几个阶段，在每个阶段实现一个小目标：首先是 `int32` 到 `uintptr` 类型，然后是 `uintptr` 到 `unsafe.Pointer` 指针类型，最后是 `unsafe.Pointer` 指针类型到 `*C.char` 类型。

## 切片间的转换

在 C 语言中数组也是一种指针，因此两个不同类型数组之间到转换和指针间转换基本类似。但是在 Go 语言中，数组或数组对应到切片都不再是指针类型，因为我们也就无法直接实现不同类型到切片之间的转换。

不过 Go 语言的 `reflect` 包提供了切片类型到底层结构，再结合前面讨论到不同类型之间到指针转换技术就可以实现 `[]X` 和 `[]Y` 类型的切片转换：

```

var p []X
var q []Y

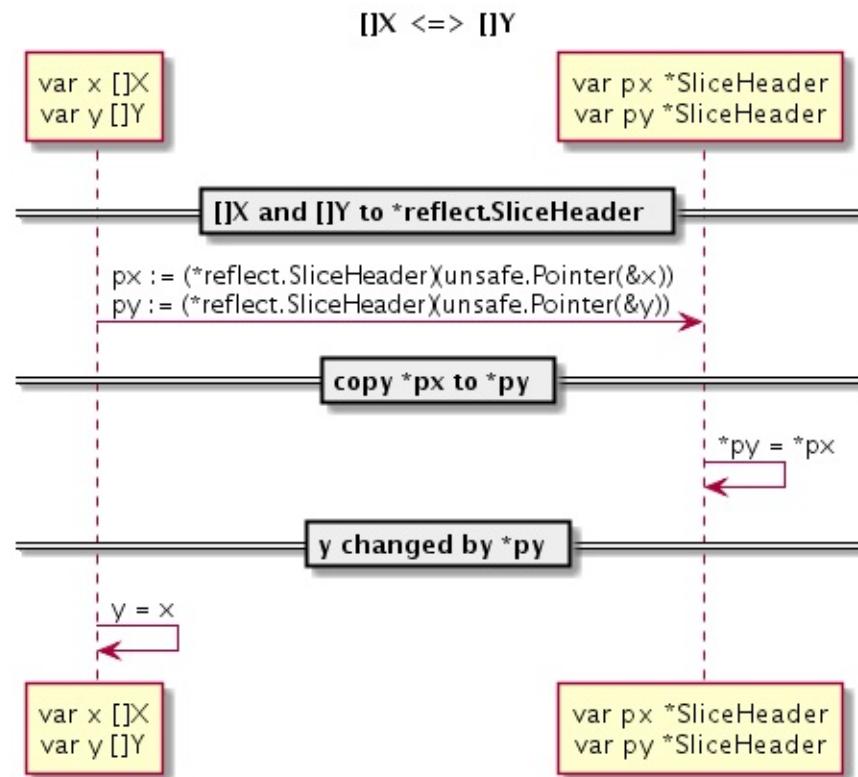
pHdr := (*reflect.SliceHeader)(unsafe.Pointer(&p))
qHdr := (*reflect.SliceHeader)(unsafe.Pointer(&q))

pHdr.Data = qHdr.Data
pHdr.Len = qHdr.Len * unsafe.Sizeof(q[0]) / unsafe.Sizeof(p[0])
pHdr.Cap = qHdr.Cap * unsafe.Sizeof(q[0]) / unsafe.Sizeof(p[0])

```

不同切片类型之间转换到思路是先为构造一个空的目标切片，然后用原有的切片底层数据填充目标切片。如果X和Y类型的大小不同，需要重新设置Len和Cap属性。需要注意的是，如果X或Y是空类型，上述代码中可能导致除0错误，实际代码需要根据情况酌情处理。

下面演示了切片间的转换的具体流程：



针对CGO中常用的功能，作者封装了 "github.com/chai2010/cgo" 包，提供基本的转换功能，具体的细节可以参考实现代码。

## 2.3. CGO编程基础

Go是一门以实用为主要目的的编程语言，我们可以通过cgo来直接调用C语言代码，也可以在C语言代码中直接调用Go函数。要使用CGO特性，需要安装C/C++构建工具链，在macOS和Linux下是要安装和GCC，在windows下是需要安装MinGW工具。同时需要保证环境变量`CGO_ENABLED`被设置为1，这表示CGO是被启用的状态。在本地构建时`CGO_ENABLED`默认是启用的，当交叉构建时CGO默认是禁止的。比如要交叉构建ARM环境运行的Go程序，需要手工设置好C/C++交叉构建的工具链，同时开启`CGO_ENABLED`环境变量。

### import "C" 语句

如果在Go代码中出现了`import "C"`语句则表示使用了CGO特性，紧跟在这行语句前面的注释是一种特殊语法，里面包含的是正常的C语言代码。当确保CGO启用的情况下，还可以在当前目录中包含C/C++对应的源文件。

举个最简单的例子：

```
package main

/*
#include <stdio.h>

void printint(int v) {
    printf("printint: %d\n", v);
}

import "C"
import "unsafe"

func main() {
    v := 42
    C.printint(C.int(v))
}
```

这个例子展示了cgo的基本使用方法。开头的注释中写了要调用的C函数和相关的头文件，头文件被`include`之后里面的所有的C语言元素都会被加入到"C"这个虚拟的包中。需要注意的是，`import "C"`导入语句需要单独一行，不能与其他包一同`import`。向C函数传递参数也很简单，就直接转化成对应C语言类型传递就可以。如上例中`C.int(v)`用于将一个Go中的`int`类型值强制类型转换转化为C语言中的`int`类型值，然后调用C语言定义的`printint`函数进行打印。

需要注意的是，Go是强类型语言，所以cgo中传递的参数类型必须与声明的类型完全一致，而且传递前必须用”C”中的转化函数转换成对应的C类型，不能直接传入Go中类型的变量。同时通过虚拟的C包导入的C语言符号并不需要是大写字母开头，它们不受Go语言的导出规则约束。

cgo将当前包引用的C语言符号都放到了虚拟的C包中，同时当前包依赖的其它Go语言包内部可能也通过cgo引入了相似的虚拟C包，但是不同的Go语言包引入的虚拟的C包之间的类型是不能通用的。这个约束对于要自己构造一些cgo辅助函数时有可能会造成一点的影响。

比如我们希望在Go中定义一个C语言字符指针对应的CChar类型，然后增加一个GoString方法返回Go语言字符串：

```
package cgo_helper

import "C"

type CChar C.char

func (p *CChar) GoString() string {
    return C.GoString((*C.char)(p))
}

func PrintCString(cs *C.char) {
    print(cs.GoString())
}
```

现在我们可能会想在其它的Go语言包中也使用这个辅助函数：

```
package main

// static char* cs = "hello"
import "C"
import "./cgo_helper"

func main() {
    cgo_helper.PrintCString(C.cs)
}
```

这段代码是不能正常工作的，因为当前main包引入的 `C.cs` 变量的类型是当前main包的cgo构造的虚拟的C包下的`char`类型，它和`cgo_helper`包引入的 `*C.char` 类型是不同的。在Go语言中方法是依附于类型存在的，不同Go包中引入的虚拟的C包的类型却是不同的，这导致从它们延伸出来的Go类型也是不同的类型，这最终导致了前面代码不能正常工作。

有Go语言使用经验的用户可能会建议参数转型后再传入。但是这个方法似乎也是不可行的，因为 `cgo_helper.PrintCString` 的参数是它自身包引入的 `*C.char` 类型，在外部是无法直接获取这个类型的。换言之，一个包如果在公开的接口中直接使用了 `*C.char` 等类似的虚拟C包的

类型，其它的Go包是无法直接使用这些类型的，除非这个Go包同时也提供了 `*C.char` 类型的构造函数。因为这些诸多因素，如果想在 `go test` 环境直接测试这些 `cgo` 导出的类型也会于通用的限制。

## #cgo 语句

在 `import "C"` 语句前的注释中可以通过 `#cgo` 语句设置编译阶段和链接阶段的相关参数。编译阶段的参数主要用于定义相关宏和指定头文件检索路径。链接阶段的参数主要是指定库文件检索路径和要链接的库文件。

```
// #cgo CFLAGS: -DPNG_DEBUG=1 -I./include
// #cgo LDFLAGS: -L/usr/local/lib -lpng
// #include <png.h>
import "C"
```

上面的代码中，`CFLAGS`部分，`-D` 部分定义了宏 `PNG_DEBUG`，值为 1；`-I` 定义了头文件包含的检索目录。`LDFLAGS`部分，`-L` 指定了链接时库文件检索目录，`-l` 指定了链接时需要链接 `png` 库。

因为C/C++遗留的问题，C头文件检索目录可以是相对目录，但是库文件检索目录则需要绝对路径。在库文件的检索目录中可以通过  `${SRCDIR}` 变量表示当前包目录的绝对路径：

```
// #cgo LDFLAGS: -L${SRCDIR}/libs -lfoo
```

上面的代码在链接时将被展开为：

```
// #cgo LDFLAGS: -L/go/src/foo/libs -lfoo
```

`#cgo` 语句主要影响 `CFLAGS`、`CPPFLAGS`、`CXXFLAGS`、`FFLAGS` 和 `LDFLAGS` 几个编译器环境变量。`LDFLAGS` 用于设置链接时的参数，除此之外的几个变量用于改变编译阶段的构建参数(`CFLAGS` 用于针对C语言代码设置编译参数)。

对于在 `cgo` 环境混合使用 C 和 C++ 的用户来说，可能有三种不同的编译选项：其中 `CFLAGS` 对应 C 语言特有的编译选项、`CXXFLAGS` 对应是 C++ 特有的编译选项、`CPPFLAGS` 则对应 C 和 C++ 共有的编译选项。但是在链接阶段，C 和 C++ 的链接选项是通用的，因此这个时候已经不再有 C 和 C++ 语言的区别，它们的目标文件的类型是相同的。

`#cgo` 指令还支持条件选择，当满足某个操作系统或某个 CPU 架构类型时后面的编译或链接选项生效。比如下面是分别针对 windows 和非 windows 下平台的编译和链接选项：

```
// #cgo windows CFLAGS: -DX86=1
// #cgo !windows LDFLAGS: -lm
```

其中在windows平台下，编译前会预定义X86宏为1；再非windows平台下，在链接阶段会要求链接math数学库。这种用法对于在不同平台下只有少数编译选项差异的场景比较适用。

如果在不同的系统下cgo对应着不同的c代码，我们可以先使用 `#cgo` 指令定义不同的C语言的宏，然后通过宏来区分不同的代码：

```
package main

/*
#cgo windows CFLAGS: -DCGO_OS_WINDOWS=1
#cgo darwin CFLAGS: -DCGO_OS_DARWIN=1
#cgo linux CFLAGS: -DCGO_OS_LINUX=1

#if defined(CGO_OS_WINDOWS)
    static char* os = "windows";
#elif defined(CGO_OS_DARWIN)
    static char* os = "darwin";
#elif defined(CGO_OS_LINUX)
    static char* os = "linux";
#else
#    error(unknown os)
#endif
*/
import "C"

func main() {
    print(C.GoString(C.os))
}
```

这样我们就可以用C语言中常用的技术来处理不同平台之间的差异代码。

## build tag 条件编译

`build tag` 是在Go或cgo环境下的C/C++文件开头的一种特殊的注释。条件编译类似于前面通过 `#cgo` 指令针对不同平台定义的宏，只有在对应平台的宏被定义之后才会构建对应的代码。但是通过 `#cgo` 指令定义宏有个限制，它只能是基于Go语言支持的windows、darwin和linux等已经支持的操作系统。如果我们希望定义一个DEBUG标志的宏，`#cgo` 指令就无能为力了。而Go语言提供的`build tag` 条件编译特性则可以简单做到。

比如下面的源文件只有在设置`debug`构建标志时才会被构建：

```
// +build debug

package main

var buildMode = "debug"
```

可以用以下命令构建：

```
go build -tags="debug"
go build -tags="windows,debug"
```

我们可以通过 `-tags` 命令行参数同时指定多个build标志，它们之间用逗号分割。

当有多个build tag时，我们将多个标志通过逻辑操作的规则来组合使用。比如以下的构建标志表示只有在linux/386或非cgo环境的darwin平台下才进行构建。

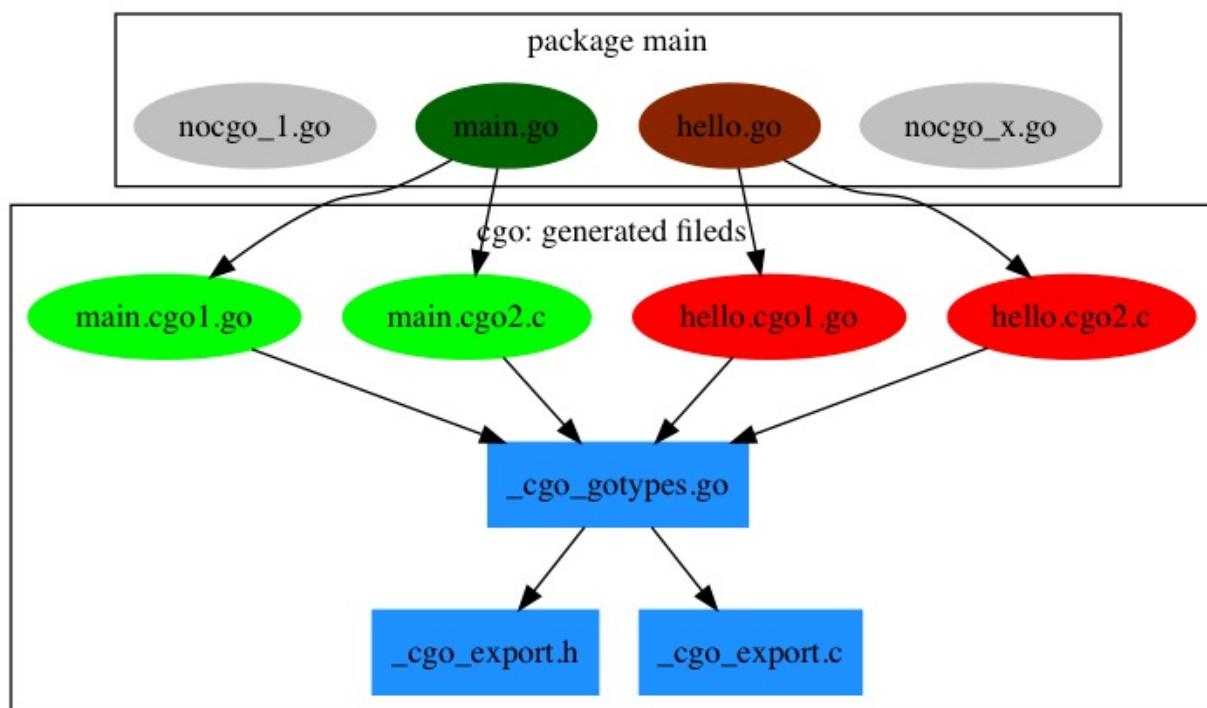
```
// +build linux,386 darwin,!cgo
```

其中 `linux,386` 中`linux`和`386`用逗号链接表示AND的意思；而 `linux,386` 和 `darwin,!cgo` 之间通过空白分割来表示OR的意思。

## CGO生成的中间文件

要了解CGO技术的底层秘密首先需要了解CGO生成了哪些中间文件。我们可以在构建一个cgo包时增加一个 `-work` 输出中间生成文件所在的目录并且在构建完成时保留中间文件。如果是比较简单的cgo代码我们也可以直接通过手工调用 `go tool cgo` 命令来查看生成的中间文件。

在一个Go源文件中，如果出现了 `import "C"` 指令则表示将调用cgo命令生成对应的中间文件。下图是cgo生成的中间文件的简单示意图：



包中有4个Go文件，其中nocgo开头的文件中没有 `import "C"` 指令，其它的2个文件则包含了cgo代码。cgo命令会为每个包含了cgo代码的Go文件创建2个中间文件，比如 `main.go` 会分别创建 `main.cgo1.go` 和 `main.cgo2.c` 两个中间文件。然后会为整个包创建一个 `_cgo_gotypes.go` Go文件，其中包含Go语言部分辅助代码。此外还会创建一个 `_cgo_export.h` 和 `_cgo_export.c` 文件，对应Go语言导出到C语言的类型和函数。

## 2.4. CGO内存模型

CGO是架接Go语言和C语言的桥梁，它使二者在二进制接口层面实现了互通，但是我们要注意因两种语言的内存模型的差异而可能引起的问题。如果在CGO处理的跨语言函数调用时涉及到了指针的传递，则可能会出现Go语言和C语言共享某一段内存的场景。我们知道C语言的内存在分配之后就是稳定的，但是Go语言因为函数栈的动态伸缩可能导致栈中内存地址的移动(这是Go和C内存模型的最大差异)。如果C语言持有的是移动之前的Go指针，那么以旧指针访问Go对象时会导致程序崩溃。

### Go访问C内存

C语言空间的内存是稳定的，只要不是被人为提前释放，那么在Go语言空间可以放心大胆地使用。在Go语言访问C语言内存是最简单的情形，我们在之前的例子中已经见过多次。

因为Go语言实现的限制，我们无法在Go语言中创建大于2GB内存的切片（具体请参考makeslice实现代码）。不过借助cgo技术，我们可以在C语言环境创建大于2GB的内存，然后转为Go语言的切片使用：

```

package main

/*
#include <stdlib.h>

void* makeslice(size_t memsize) {
    return malloc(memsize);
}

import "C"
import "unsafe"

func makeByteSlice(n int) []byte {
    p := C.makeslice(C.size_t(n))
    return (*[1<<31]byte)(p)[0:n:n]
}

func freeByteSlice(p []byte) {
    C.free(unsafe.Pointer(&p[0]))
}

func main() {
    s := makeByteSlice(1<<32+1)
    s[len[s]-1] = 1234
    print(s[len[s]-1])
    freeByteSlice(p)
}

```

例子中我们通过`makeByteSlice`来创建大于4G内存大小的切片，从而绕过了Go语言实现的限制（需要代码验证）。而`freeByteSlice`辅助函数则用于释放从C语言函数创建的切片。

因为C语言内存空间是稳定的，基于C语言内存构造的切片也是绝对稳定的，不会因为Go语言栈的变化而被移动。

## C临时访问传入的Go内存

cgo之所以存在的一大因素是为了方便在Go语言中接纳吸收过去几十年来使用C/C++语言软件构建的大量的软件资源。C/C++很多库都是需要通过指针直接处理传入的内存数据的，因此cgo中也有很多需要将Go内存传入C语言函数的应用场景。

假设一个极端场景：我们将一块位于某goroutine的栈上的Go语言内存传入了C语言函数后，在此C语言函数执行期间，此goroutine的栈因为空间不足的原因发生了扩展，也就是导致了原来的Go语言内存被移动到了新的位置。但是此时此刻C语言函数并不知道该Go语言内存已经移动了位置，仍然用之前的地址来操作该内存——这将导致内存越界。以上是一个推论（真实情况有些差异），也就是说C访问传入的Go内存可能是不安全的！

当然有RPC远程过程调用的经验的用户可能会考虑通过完全传值的方式处理：借助C语言内存稳定的特性，在C语言空间先开辟同样大小的内存，然后将Go的内存填充到C的内存空间；返回的内存也是如此处理。下面的例子是这种思路的具体实现：

```
package main

/*
void printString(const char* s) {
    printf("%s", s);
}
*/
import "C"

func printString(s string) {
    cs := C.CString(s)
    defer C.free(unsafe.Pointer(cs))

    C.printString(cs)
}

func main() {
    s := "hello"
    printString(s)
}
```

在需要将Go的字符串传入C语言时，先通过 `C.CString` 将Go语言字符串对应的内存数据复制到新创建的C语言内存空间上。上面例子的处理思路虽然是安全的，但是效率极其低下（因为要多次分配内存并逐个复制元素），同时也极其繁琐。

为了简化并高效处理此种向C语言传入Go语言内存的问题，`cgo`针对该场景定义了专门的规则：在CGO调用的C语言函数返回前，`cgo`保证传入的Go语言内存在此期间不会发生移动，C语言函数可以大胆地使用Go语言的内存！

根据新的规则我们可以直接传入Go字符串的内存：

```

package main

/*
void printString(const char* s) {
    printf("%s", s);
}
*/
import "C"

func printString(s string) {
    C.printString((*C.char)(unsafe.Pointer(&s[0])))
}

func main() {
    s := "hello"
    printString(s)
}

```

现在的处理方式更加直接，且避免了分配额外的内存。完美的解决方案！

任何完美的技术都有被滥用的时候，CGO的这种看似完美的规则也是存在隐患的。我们假设调用的C语言函数需要长时间运行，那么将会导致被它引用的Go语言内存在C语言返回前不能被移动，从而可能间接地导致这个Go内存栈对应的goroutine不能动态伸缩栈内存，也就是可能导致这个goroutine被阻塞。因此，在需要长时间运行的C语言函数（特别是在纯CPU运算之外，还可能因为需要等待其它的资源而需要不确定时间才能完成的函数），需要谨慎处理传入的Go语言内存。

不过需要小心的是在取得Go内存后需要马上传入C语言函数，不能保存到临时变量后再间接传入C语言函数。因为CGO只能保证在C函数调用之后被传入的Go语言内存不会发生移动，它并不能保证在传入C函数之前内存不发生变化。

以下代码是错误的：

```

// 错误的代码
tmp := uintptr(unsafe.Pointer(&x))
pb := (*int16)(unsafe.Pointer(tmp))
*pb = 42

```

因为tmp并不是指针类型，在它获取到Go对象地址之后x对象可能会被移动，但是因为不是指针类型，所以不会被Go语言运行时更新成新内存的地址。在非指针类型的tmp保持Go对象的地址，和在C语言环境保持Go对象的地址的效果是一样的：如果原始的Go对象内存发生了移动，Go语言运行时并不会同步更新它们。

## C长期持有Go指针对象

作为一个Go程序员在使用CGO时潜意识会认为总是Go调用C函数。其实CGO中，C语言函数也可以回调Go语言实现的函数。特别是我们可以用Go语言写一个动态库，导出C语言规范的接口给其它用户调用。当C语言函数调用Go语言函数的时候，C语言函数就成了程序的调用方，Go语言函数返回的Go对象内存的生命周期也就自然超出了Go语言运行时的管理。简言之，我们不能在C语言函数中直接使用Go语言对象的内存。

虽然Go语言禁止在C语言函数中长期持有Go指针对象，但是这种需求是切实存在的。如果需要在C语言中访问Go语言内存对象，我们可以将Go语言内存对象在Go语言空间映射为一个int类型的id，然后通过此id来间接访问和控制Go语言对象。

以下代码用于将Go对象映射为整数类型的ObjectId，用完之后需要手工调用free方法释放该对象ID：

```
package main

import "sync"

type ObjectId int32

var refs struct {
    sync.Mutex
    objs map[ObjectId]interface{}
    next ObjectId
}

func init() {
    refs.Lock()
    defer refs.Unlock()

    refs.objs = make(map[ObjectId]interface{})
    refs.next = 1000
}

func NewObjectId(obj interface{}) ObjectId {
    refs.Lock()
    defer refs.Unlock()

    id := refs.next
    refs.next++

    refs.objs[id] = obj
    return id
}

func (id ObjectId) IsNil() bool {
    return id == 0
}

func (id ObjectId) Get() interface{} {
    refs.Lock()
```

```
    defer refs.Unlock()

    return refs objs[id]
}

func (id *ObjectId) Free() interface{} {
    refs.Lock()
    defer refs.Unlock()

    obj := refs objs[*id]
    delete(refs objs, *id)
    *id = 0

    return obj
}
```

我们通过一个map来管理Go语言对象和id对象的映射关系。其中NewObjectId用于创建一个和对象绑定的id，而id对象的方法可用于解码出原始的Go对象，也可以用于结束id和原始Go对象的绑定。

下面一组函数以C接口规范导出，可以被C语言函数调用：

```

package main

/*
export char* NewGoString(char* );
export void FreeGoString(char* );
export void PrintGoString(char* );

void printString(const char* s) {
    char* gs = NewGoString(s);
    PrintGoString(gs);
    FreeGoString(gs);
}
*/
import "C"

//export NewGoString
func NewGoString(s *C.char) *C.char {
    gs := C.GoString(s)
    id := NewObjectId(gs)
    return (*C.char)(unsafe.Pointer(uintptr(id)))
}

//export FreeGoString
func FreeGoString(p *C.char) {
    id := ObjectId(uintptr(unsafe.Pointer(p)))
    id.Free()
}

//export PrintGoString
func PrintGoString(s *C.char) {
    id := ObjectId(uintptr(unsafe.Pointer(p)))
    gs := id.Get().(string)
    print(gs)
}

func main() {
    C.printString("hello")
}

```

在printString函数中，我们通过NewGoString创建一个对应的Go字符串对象，返回的其实是一个ID，不能直接使用。我们借助PrintGoString函数将id解析为Go语言字符串后打印。该字符串在C语言函数中完全跨越了Go语言的内存管理，在PrintGoString调用前即使发生了栈伸缩导致的Go字符串地址发生变化也依然可以正常工作，因为该字符串对应的id是稳定的，在Go语言空间通过id解码得到的字符串也就是有效的。

## 导出C函数不能返回Go内存

在Go语言中，Go是从一个固定的虚拟地址空间分配内存。而C语言分配的内存则不能使用Go语言保留的虚拟内存空间。在CGO环境，Go语言运行时默认会检查导出返回的内存是否是由Go语言分配的，如果是则会抛出运行时异常。

下面是CGO运行时异常的例子：

```
/*
extern int* getGoPtr();

static void Main() {
    int* p = getGoPtr();
    *p = 42;
}

import "C"

func main() {
    C.Main()
}

//export getGoPtr
func getGoPtr() *C.int {
    return new(C.int)
}
```

其中getGoPtr返回的虽然是C语言类型的指针，但是内存本身是从Go语言的new函数分配，也就是由Go语言运行时统一管理的内存。然后我们在C语言的Main函数中调用了getGoPtr函数，此时默认将发送运行时异常：

```
$ go run main.go
panic: runtime error: cgo result has Go pointer

goroutine 1 [running]:
main._cgoexpwrap_cfb3840e3af2_getGoPtr.func1(0xc420051dc0)
    command-line-arguments/_obj/_cgo_gotypes.go:60 +0x3a
main._cgoexpwrap_cfb3840e3af2_getGoPtr(0xc420016078)
    command-line-arguments/_obj/_cgo_gotypes.go:62 +0x67
main._Cfunc_Main()
    command-line-arguments/_obj/_cgo_gotypes.go:43 +0x41
main.main()
    /Users/chai/go/src/github.com/chai2010/advanced-go-programming-book/examples/ch2-
exit status 2
```

异常说明cgo函数返回的结果中含有Go语言分配的指针。指针的检查操作发生在C语言版的getGoPtr函数中，它是由cgo生成的桥接C语言和Go语言的函数。

下面是cgo生成的C语言版本getGoPtr函数的具体细节（在cgo生成的 `_cgo_export.c` 文件定义）：

```
int* getGoPtr()
{
    __SIZE_TYPE__ _cgo_ctxt = _cgo_wait_runtime_init_done();
    struct {
        int* r0;
    } __attribute__((__packed__)) a;
    _cgo_tsan_release();
    crosscall2(_cgoexp_95d42b8e6230_getGoPtr, &a, 8, _cgo_ctxt);
    _cgo_tsan_acquire();
    _cgo_release_context(_cgo_ctxt);
    return a.r0;
}
```

其中 `_cgo_tsan_acquire` 是从LLVM项目移植过来的内存指针扫描函数，它会检查cgo函数返回的结果是否包含Go指针。

需要说明的是，cgo默认对返回结果的指针的检查是有代价的，特别是cgo函数返回的结果是一个复杂的数据结构时将花费更多的时间。如果已经确保了cgo函数返回的结果是安全的话，可以通过设置环境变量 `GODEBUG=cgocheck=0` 来关闭指针检查行为。

```
$ GODEBUG=cgocheck=0 go run main.go
```

关闭cgocheck功能后再运行上面的代码就不会出现上面的异常的。但是要注意的是，如果C语言使用期间对应的内存被Go运行时释放了，将会导致更严重的崩溃问题。cgocheck默认的值是1，对应一个简化版本的检测，如果需要完整的检测功能可以将cgocheck设置为2。

关于cgo运行时指针检测的功能详细说明可以参考Go语言的官方文档。

## 2.5. C++ 类包装

CGO是C语言和Go语言之间的桥梁，原则上无法直接支持C++的类。CGO不支持C++语法的根本原因是C++至今为止还没有一个二进制接口规范(ABI)。一个C++类的构造函数在编译为目标文件时如何生成链接符号名称、方法在不同平台甚至是C++的不同版本之间都是不一样的。但是C++是兼容C语言，所以我们可以通过增加一组C语言函数接口作为C++类和CGO之间的桥梁，这样就可以间接地实现C++和Go之间的互联。当然，因为CGO只支持C语言中值类型的数据类型，所以我们是无法直接使用C++的引用参数等特性的。

### C++ 类到 Go 语言对象

实现C++类到Go语言对象的包装需要经过以下几个步骤：首先是用纯C函数接口包装该C++类；其次是通过CGO将纯C函数接口映射到Go函数；最后是做一个Go包装对象，将C++类到方法用Go对象的方法实现。

#### 准备一个 C++ 类

为了演示简单，我们基于 `std::string` 做一个最简单的缓存类`MyBuffer`。除了构造函数和析构函数之外，只有两个成员函数分别是返回底层的数据指针和缓存的大小。因为是二进制缓存，所以我们可以在里面放置任意数据。

```
// my_buffer.h
#include <string>

struct MyBuffer {
    std::string* s_;

    MyBuffer(int size) {
        this->s_ = new std::string(size, char('\0'));
    }
    ~MyBuffer() {
        delete this->s_;
    }

    int Size() const {
        return this->s_->size();
    }
    char* Data() {
        return (char*)this->s_->data();
    }
};
```

我们在构造函数中指定缓存的大小并分配空间，在使用完之后通过析构函数释放内部分配的内存空间。下面是简单的使用方式：

```
int main() {
    auto pBuf = new MyBuffer(1024);

    auto data = pBuf->Data();
    auto size = pBuf->Size();

    delete pBuf;
}
```

为了方便向C语言接口过渡，在此处我们故意没有定义C++的拷贝构造函数。我们必须以new和delete来分配和释放缓存对象，而不能以值风格的方式来使用。

## 用纯C函数接口封装 C++ 类

如果要将上面的C++类用C语言函数接口封装，我们可以从使用方式入手。我们可以将new和delete映射为C语言函数，将对象的方法也映射为C语言函数。

在C语言中我们期望MyBuffer类可以这样使用：

```
int main() {
    MyBuffer* pBuf = NewMyBuffer(1024);

    char* data = MyBuffer_Data(pBuf);
    auto size = MyBuffer_Size(pBuf);

    DeleteMyBuffer(pBuf);
}
```

先从C语言接口用户的角度思考需要什么样的接口，然后创建 `my_buffer_capi.h` 头文件接口规范：

```
// my_buffer_capi.h
typedef struct MyBuffer_T MyBuffer_T;

MyBuffer_T* NewMyBuffer(int size);
void DeleteMyBuffer(MyBuffer_T* p);

char* MyBuffer_Data(MyBuffer_T* p);
int MyBuffer_Size(MyBuffer_T* p);
```

然后就可以基于C++的MyBuffer类定义这些C语言包装函数。我们创建对应的 `my_buffer_capi.cc` 文件如下：

```
// my_buffer_capi.cc

#include "./my_buffer.h"

extern "C" {
    #include "./my_buffer_capi.h"
}

struct MyBuffer_T: MyBuffer {
    MyBuffer_T(int size): MyBuffer(size) {}
    ~MyBuffer_T() {}
};

MyBuffer_T* NewMyBuffer(int size) {
    auto p = new MyBuffer_T(size);
    return p;
}

void DeleteMyBuffer(MyBuffer_T* p) {
    delete p;
}

char* MyBuffer_Data(MyBuffer_T* p) {
    return p->Data();
}

int MyBuffer_Size(MyBuffer_T* p) {
    return p->Size();
}
```

因为头文件 `my_buffer_capi.h` 是用于CGO，必须是采用C语言规范的名字修饰规则。在C++源文件包含时需要用 `extern "C"` 语句说明。另外`MyBuffer_T`的实现只是从`MyBuffer`继承的类，这样可以简化包装代码的实现。同时和CGO通信时必须通过 `MyBuffer_T` 指针，我们无法将具体的实现暴露给CGO，因为实现中包含了C++特有的语法，CGO无法识别C++特性。

将C++类包装为纯C接口之后，下一步的工作就是将C函数转为Go函数。

## 将纯C接口函数转为Go函数

将纯C函数包装为对应的Go函数的过程比较简单。需要注意的是，因为我们的包中包含C++11的语法，因此需要通过 `#cgo CXXFLAGS: -std=c++11` 打开C++11的选项。

```
// my_buffer_capi.go

package main

/*
#cgo CXXFLAGS: -std=c++11

#include "my_buffer_capi.h"
*/
import "C"

type cgo_MyBuffer_T C.MyBuffer_T

func cgo_NewMyBuffer(size int) *cgo_MyBuffer_T {
    p := C.NewMyBuffer(C.int(size))
    return (*cgo_MyBuffer_T)(p)
}

func cgo_DeleteMyBuffer(p *cgo_MyBuffer_T) {
    C.DeleteMyBuffer((*C.MyBuffer_T)(p))
}

func cgo_MyBuffer_Data(p *cgo_MyBuffer_T) *C.char {
    return C.MyBuffer_Data((*C.MyBuffer_T)(p))
}

func cgo_MyBuffer_Size(p *cgo_MyBuffer_T) C.int {
    return C.MyBuffer_Size((*C.MyBuffer_T)(p))
}
```

为了区分，我们在Go中的每个类型和函数名称前面增加了 `cgo_` 前缀，比如`cgo_MyBuffer_T`是对应C中的`MyBuffer_T`类型。

为了处理简单，在包装纯C函数到Go函数时，除了`cgo_MyBuffer_T`类型外，对输入参数和返回值的基础类型，我们依然是用的C语言的类型。

## 包装为Go对象

在将纯C接口包装为Go函数之后，我们就可以很容易地基于包装的Go函数构造出Go对象来。因为`cgo_MyBuffer_T`是从C语言空间导入的类型，它无法定义自己的方法，因此我们构造了一个新的`MyBuffer`类型，里面的成员持有`cgo_MyBuffer_T`指向的C语言缓存对象。

```
// my_buffer.go

package main

import "unsafe"

type MyBuffer struct {
    cptr *cgo_MyBuffer_T
}

func NewMyBuffer(size int) *MyBuffer {
    return &MyBuffer{
        cptr: cgo_NewMyBuffer(size),
    }
}

func (p *MyBuffer) Delete() {
    cgo_DeleteMyBuffer(p.cptr)
}

func (p *MyBuffer) Data() []byte {
    data := cgo_MyBuffer_Data(p.cptr)
    size := cgo_MyBuffer_Size(p.cptr)
    return ((*[1<<31]byte)(unsafe.Pointer(data)))[0:int(size):int(size)]
}
```

同时，因为Go语言的切片本身含有长度信息，我们将cgo\_MyBuffer\_Data和cgo\_MyBuffer\_Size两个函数合并为 `MyBuffer.Data` 方法，它返回一个对应底层C语言缓存空间的切片。

现在我们就可以很容易在Go语言中使用包装后的缓存对象了（底层是基于C++的 `std::string` 实现）：

```
package main

//#include <stdio.h>
import "C"
import "unsafe"

func main() {
    buf := NewMyBuffer(1024)
    defer buf.Delete()

    copy(buf.Data(), []byte("hello\x00"))
    C.puts((*C.char)(unsafe.Pointer(&(buf.Data())[0])))
}
```

例子中，我们创建了一个1024字节大小的缓存，然后通过copy函数向缓存填充了一个字符串。为了方便C语言字符串函数处理，我们在填充字符串的默认用'\0'表示字符串结束。最后我们直接获取缓存的底层数据指针，用C语言的puts函数打印缓存的内容。

## Go 语言对象到 C++ 类

要实现Go语言对象到C++类的包装需要经过以下几个步骤：首先是将Go对象映射为一个id；然后基于id导出对应的C接口函数；最后是基于C接口函数包装为C++对象。

### 构造一个Go对象

为了便于演示，我们用Go语言构建了一个Person对象，每个Person可以有名字和年龄信息：

```
package main

type Person struct {
    name string
    age  int
}

func NewPerson(name string, age int) *Person {
    return &Person{
        name: name,
        age:  age,
    }
}

func (p *Person) Set(name string, age int) {
    p.name = name
    p.age = age
}

func (p *Person) Get() (name string, age int) {
    return p.name, p.age
}
```

Person对象如果想要在C/C++中访问，需要通过cgo导出C接口来访问。

### 导出C接口

我们前面仿照C++对象到C接口的过程，也抽象一组C接口描述Person对象。创建一个 person\_capi.h 文件，对应C接口规范文件：

```
// person_capi.h
#include <stdint.h>

typedef uintptr_t person_handle_t;

person_handle_t person_new(char* name, int age);
void person_delete(person_handle_t p);

void person_set(person_handle_t p, char* name, int age);
char* person_get_name(person_handle_t p, char* buf, int size);
int person_get_age(person_handle_t p);
```

然后是在Go语言中实现这一组C函数。

需要注意的是，通过CGO导出C函数时，输入参数和返回值类型都不支持const修饰，同时也不支持可变参数的函数类型。同时如内存模式一节所述，我们无法在C/C++中直接长期访问Go内存对象。因此我们使用前一节所讲述的技术将Go对象映射为一个整数id。

下面是 person\_capi.go 文件，对应C接口函数的实现：

```
// person_capi.go
package main

//#include "./person_capi.h"
import "C"
import "unsafe"

//export person_new
func person_new(name *C.char, age C.int) C.person_handle_t {
    id := NewObjectId(NewPerson(C.GoString(name), int(age)))
    return C.person_handle_t(id)
}

//export person_delete
func person_delete(h C.person_handle_t) {
    ObjectId(h).Free()
}

//export person_set
func person_set(h C.person_handle_t, name *C.char, age C.int) {
    p := ObjectId(h).Get().(*Person)
    p.Set(C.GoString(name), int(age))
}

//export person_get_name
func person_get_name(h C.person_handle_t, buf *C.char, size C.int) *C.char {
    p := ObjectId(h).Get().(*Person)
    name, _ := p.Get()

    n := int(size) - 1
    bufSlice := ((*[1<<31]byte)(unsafe.Pointer(buf)))[0:n:n]
    n = copy(bufSlice, []byte(name))
    bufSlice[n] = 0

    return buf
}

//export person_get_age
func person_get_age(h C.person_handle_t) C.int {
    p := ObjectId(h).Get().(*Person)
    _, age := p.Get()
    return C.int(age)
}
```

在创建Go对象后，我们通过NewObjectId将Go对应映射为id。然后将id强制转义为person\_handle\_t类型返回。其它的接口函数则是根据person\_handle\_t所表示的id，让根据id解析出对应的Go对象。

## 封装C++对象

有了C接口之后封装C++对象就比较简单了。常见的做法是新建一个Person类，里面包含一个person\_handle\_t类型的成员对应真实的Go对象，然后在Person类的构造函数中通过C接口创建Go对象，在析构函数中通过C接口释放Go对象。下面是采用这种技术的实现：

```
extern "C" {
    #include "./person_capi.h"
}

struct Person {
    person_handle_t goobj_;

    Person(const char* name, int age) {
        this->goobj_ = person_new((char*)name, age);
    }
    ~Person() {
        person_delete(this->goobj_);
    }

    void Set(char* name, int age) {
        person_set(this->goobj_, name, age);
    }
    char* GetName(char* buf, int size) {
        return person_get_name(this->goobj_, buf, size);
    }
    int GetAge() {
        return person_get_age(this->goobj_);
    }
}
```

包装后我们就可以像普通C++类那样使用了：

```
#include "person.h"

#include <stdio.h>

int main() {
    auto p = new Person("gopher", 10);

    char buf[64];
    char* name = p->GetName(buf, sizeof(buf)-1);
    int age = p->GetAge();

    printf("%s, %d years old.\n", name, age);
    delete p;

    return 0;
}
```

## 封装C++对象改进

在前面的封装C++对象的实现中，每次通过new创建一个Person实例需要进行两次内存分配：一次是针对C++版本的Person，再一次是针对Go语言版本的Person。其实C++版本的Person内部只有一个person\_handle\_t类型的id，用于映射Go对象。我们完全可以将person\_handle\_t直接当中C++对象来使用。

下面时改进后的包装方式：

```
extern "C" {
    #include "./person_cap.h"
}

struct Person {
    static Person* New(const char* name, int age) {
        return (Person*)person_new((char*)name, age);
    }
    void Delete() {
        person_delete(person_handle_t(this));
    }

    void Set(char* name, int age) {
        person_set(person_handle_t(this), name, age);
    }
    char* GetName(char* buf, int size) {
        return person_get_name(person_handle_t(this), buf, size);
    }
    int GetAge() {
        return person_get_age(person_handle_t(this));
    }
};
```

我们在Person类中增加了一个叫New静态成员函数，用于创建新的Person实例。在New函数中通过调用person\_new来创建Person实例，返回的是 person\_handle\_t 类型的id，我们将其强制转型作为 Person\* 类型指针返回。在其它的成员函数中，我们通过将this指针再反向转型为 person\_handle\_t 类型，然后通过C接口调用对应的函数。

到此，我们就达到了将Go对象导出为C接口，然后基于C接口再包装为C++对象以便于使用的目的。

## 2.6. 静态库和动态库

CGO在使用C/C++资源的时候一般有三种形式：直接使用源码；链接静态库；链接动态库。直接使用源码就是在`import "c"`之前的注释部分包含C代码，或者在当前包中包含C/C++源文件。链接静态库和动态库的方式比较类似，都是通过在LDFLAGS选项指定要链接的库方式链接。本节我们主要关注在CGO中如何使用静态库和动态库相关的问题。

### 使用C静态库

如果CGO中引入的C/C++资源有代码而且代码规模也比较小，直接使用源码是最理想的方式，但很多时候我们并没有源代码，或者从C/C++源代码开始构建的过程异常复杂，这种时候使用C静态库也是一个不错的选择。静态库因为是静态链接，最终的目标程序并不会产生额外的运行时依赖，也不会出现动态库特有的跨运行时资源管理的错误。不过静态库对链接阶段会有一定要求：静态库一般包含了全部的代码，里面有大量的符号，如果不同静态库之间出现了符号冲突则会导致链接的失败。

我们先用纯C语言构造一个简单的静态库。我们要构造的静态库名叫number，库中只有一个number\_add\_mod函数，用于表示数论中的模加法运算。number库的文件都在number目录下。

number/number.h 头文件只有一个纯C语言风格的函数声明：

```
int number_add_mod(int a, int b, int mod);
```

number/number.c 对应函数的实现：

```
#include "number.h"

int number_add_mod(int a, int b, int mod) {
    return (a+b)%mod;
}
```

因为CGO使用的是GCC命令来编译和链接C和Go桥接的代码。因此静态库也必须是GCC兼容的格式。

通过以下命令可以生成一个叫libnumber.a的静态库：

```
$ cd ./number
$ gcc -c -o number.o number.c
$ ar rcs libnumber.a number.o
```

生成libnumber.a静态库之后，我们就可以在CGO中使用该资源了。

创建main.go文件如下：

```
package main

//#cgo CFLAGS: -I./number
//#cgo LDFLAGS: -L${SRCDIR}/number -lnumber
//
//#include "number.h"
import "C"
import "fmt"

func main() {
    fmt.Println(C.number_add_mod(10, 5, 12))
}
```

其中有两个#cgo命令，分别是编译和链接参数。CFLAGS通过 -I./number 将number库对应头文件所在的目录加入头文件检索路径。LDFLAGS通过 -L\${SRCDIR}/number 将编译后number静态库所在目录加为链接库检索路径， -lnumber 表示链接libnumber.a静态库。需要注意的是，在链接部分的检索路径不能使用相对路径（C/C++代码的链接程序所限制），我们必须通过cgo特有的 \${SRCDIR} 变量将源文件对应的当前目录路径展开为绝对路径（因此在windows平台中绝对路径不能有空白符号）。

因为我们有number库的全部代码，所以我们可以用go generate工具来生成静态库，或者是通过Makefile来构建静态库。因此发布CGO源码包时，我们并不需要提前构建C静态库。

因为多了一个静态库的构建步骤，这种使用了自定义静态库并已经包含了静态库全部代码的Go包无法直接用go get安装。不过我们依然可以通过go get下载，然后用go generate触发静态库构建，最后才是go install来完成安装。

为了支持go get命令直接下载并安装，我们C语言的 #include 语法可以将number库的源文件链接到当前的包。

创建 z\_link\_number\_c.c 文件如下：

```
#include "./number/number.c"
```

然后在执行go get或go build之类命令的时候，CGO就是自动构建number库对应的代码。这种技术是在不改变静态库源代码组织结构的前提下，将静态库转化为了源代码方式引用。这种CGO包是最完美的。

如果使用的是第三方的静态库，我们需要先下载安装静态库到合适的位置。然后在#cgo命令中通过CFLAGS和LDFLAGS来指定头文件和库的位置。对于不同的操作系统甚至同一种操作系统的不同版本来说，这些库的安装路径可能都是不同的，那么如何在代码中指定这些可能

变化的参数呢？

在Linux环境，有一个pkg-config命令可以查询要使用某个静态库或动态库时的编译和链接参数。我们可以在#cgo命令中直接使用pkg-config命令来生成编译和链接参数。而且还可以通过PKG\_CONFIG环境变量订制pkg-config命令。因为不同的操作系统对pkg-config命令的支持不尽相同，通过该方式很难兼容不同的操作系统下的构建参数。不过对于Linux等特定的系统，pkg-config命令确实可以简化构建参数的管理。关于pkg-config的使用细节在此我们不深入展开，大家可以自行参考相关文档。

## 使用C动态库

动态库出现的初衷是对于相同的库，多个进程可以共享同一个，以节省内存和磁盘资源。但是在磁盘和内存已经白菜价的今天，这两个作用已经显得微不足道了，那么除此之外动态库还有哪些存在的价值呢？从库开发角度来说，动态库可以隔离不同动态库之间的关系，减少链接时出现符号冲突的风险。而且对于windows等平台，动态库是跨越VC和GCC不同编译器平台的唯一的可行方式。

对于CGO来说，使用动态库和静态库是一样的，因为动态库也必须要有一个小的静态导出库用于链接动态库（Linux下可以直接链接so文件，但是在Windows下必须为dll创建一个.a文件用于链接）。我们还是以前面的number库为例来说明如何以动态库方式使用。

对于在macOS和Linux系统下的gcc环境，我们可以用以下命令创建number库的动态库：

```
$ cd number
$ gcc -fPIC -shared -o libnumber.so number.c
```

因为动态库和静态库的基础名称都是libnumber，只是后缀名不同而已。因此Go语言部分的代码和静态库版本完全一样：

```
package main

//#cgo CFLAGS: -I./number
//#cgo LDFLAGS: -L${SRCDIR}/number -lnumber
//
//#include "number.h"
import "C"
import "fmt"

func main() {
    fmt.Println(C.number_add_mod(10, 5, 12))
}
```

编译时GCC会自动找到libnumber.a或libnumber.so进行链接。

对于windows平台，我们还可以用VC工具来生成动态库（windows下有一些复杂的C++库只能用VC构建）。我们需要先为number.dll创建一个def文件，用于控制要导出到动态库的符号。

number.def文件的内容如下：

```
LIBRARY number.dll

EXPORTS
number_add_mod
```

其中第一行的LIBRARY指明动态库的文件名，然后的EXPORTS语句之后是要导出的符号名列表。

现在我们可以用以下命令来创建动态库（需要进入VC对应的x64命令行环境）。

```
$ cl /c number.c
$ link /DLL /OUT:number.dll number.obj number.def
```

这时候会为dll同时生成一个number.lib的导出库。但是在CGO中我们无法使用lib格式的链接库。

要生成.a格式的导出库需要通过mingw工具箱中的dlltool命令完成：

```
$ dlltool -dllname number.dll --def number.def --output-lib libnumber.a
```

生成了libnumber.a文件之后，就可以通过-lnumber链接参数进行链接了。

需要注意的是，在运行时需要将动态库放到系统能够找到的位置。对于windows来说，可以将动态库和可执行程序放到同一个目录，或者将动态库所在的目录绝对路径添加到PATH环境变量中。对于macOS来说，需要设置DYLD\_LIBRARY\_PATH环境变量。而对于Linux系统来说，需要设置LD\_LIBRARY\_PATH环境变量。

## 导出C静态库

CGO不仅可以使用C静态库，也可以将Go实现的函数导出为C静态库。我们现在用Go实现前面的number库的模加法函数。

创建number.go，内容如下：

```

package main

import "C"

func main() {}

//export number_add_mod
func number_add_mod(a, b, mod C.int) C.int {
    return (a + b) % mod
}

```

根据CGO文档的要求，我们需要在main包中导出C函数。对于C静态库构建方式来说，会忽略main包中的main函数，只是简单导出C函数。采用以下命令构建：

```
$ go build -buildmode=c-archive -o number.a
```

在生成number.a静态库的同时，cgo还会生成一个number.h文件。

number.h文件的内容如下（为了便于显示，内容做了精简）：

```

#ifndef __cplusplus
extern "C" {
#endif

extern int number_add_mod(int p0, int p1, int p2);

#ifndef __cplusplus
}
#endif

```

其中 `extern "C"` 部分的语法是为了同时适配C和C++两种语言。核心内容是声明了要导出的 `number_add_mod` 函数。

然后我们创建一个 `_test_main.c` 的C文件用于测试生成的C静态库（用下划线作为前缀名是让为了让go build构建C静态库时忽略这个文件）：

```
#include "number.h"

#include <stdio.h>

int main() {
    int a = 10;
    int b = 5;
    int c = 12;

    int x = number_add_mod(a, b, c);
    printf("(%d+%d)%%d = %d\n", a, b, c, x);

    return 0;
}
```

通过以下命令编译并运行：

```
$ gcc -o a.out _test_main.c number.a
$ ./a.out
```

使用CGO创建静态库的过程非常简单。

## 导出C动态库

CGO导出动态库的过程和静态库类似，只是将构建模式改为 `c-shared`，输出文件名改为 `number.so` 而已：

```
$ go build -buildmode=c-shared -o number.so
```

`_test_main.c` 文件内容不变，然后用以下命令编译并运行：

```
$ gcc -o a.out _test_main.c number.so
$ ./a.out
```

## 导出非**main**包的函数

通过 `go help buildmode` 命令可以查看C静态库和C动态库的构建说明：

```
-buildmode=c-archive
Build the listed main package, plus all packages it imports,
into a C archive file. The only callable symbols will be those
functions exported using a cgo //export comment. Requires
exactly one main package to be listed.

-buildmode=c-shared
Build the listed main package, plus all packages it imports,
into a C shared library. The only callable symbols will
be those functions exported using a cgo //export comment.
Requires exactly one main package to be listed.
```

文档说明导出的C函数必须是在main包导出，然后才能在生成的头文件包含声明的语句。但是很多时候我们可能更希望将不同类型的导出函数组织到不同的Go包中，然后统一导出为一个静态库或动态库。

要实现从是从非main包导出C函数，或者是多个包导出C函数（因为只能有一个main包），我们需要自己提供导出C函数对应的头文件（因为CGO无法为非main包的导出函数生成头文件）。

假设我们先创建一个number子包，用于提供模加法函数：

```
package number

import "C"

//export number_add_mod
func number_add_mod(a, b, mod C.int) C.int {
    return (a + b) % mod
}
```

然后是当前的main包：

```

package main

import "C"

import (
    "fmt"
    _ "./number"
)

func main() {
    println("Done")
}

//export goPrintln
func goPrintln(s *C.char) {
    fmt.Println("goPrintln:", C.GoString(s))
}

```

其中我们导入了number子包，在number子包中有导出的C函数number\_add\_mod，同时我们在main包也导出了goPrintln函数。

通过以下命令创建C静态库：

```
$ go build -buildmode=c-archive -o main.a
```

这时候在生成main.a静态库的同时，也会生成一个main.h头文件。但是main.h头文件中只有main包中导出的goPrintln函数的声明，并没有number子包导出函数的声明。其实number\_add\_mod函数在生成的C静态库中是存在的，我们可以直接使用。

创建 \_test\_main.c 测试文件如下：

```

#include <stdio.h>

void goPrintln(char*);
int number_add_mod(int a, int b, int mod);

int main() {
    int a = 10;
    int b = 5;
    int c = 12;

    int x = number_add_mod(a, b, c);
    printf("(%d+%d)%%d = %d\n", a, b, c, x);

    goPrintln("done");
    return 0;
}

```

我们并没有包含CGO自动生成的main.h头文件，而是通过手工方式声明了goPrintln和number\_add\_mod两个导出函数。这样我们就实现了从多个Go包导出C函数了。

## 2.7 Go实现Python模块

前面章节我们已经讲述了如何通过CGO来引用和创建C动态库和静态库。实现了对C动态库和静态库的支持，理论上就可以应用到动态库的绝大部分场景。Python语言作为当下最红的语言，本节我们将演示如何通过Go语言来为Python脚本语言编写扩展模块。

### 基于ctypes

Python内置了非常丰富的模块，其中ctypes支持直接从C动态库调用函数。为了演示如何基于ctypes技术来扩展模块，我们需要先用Go语言创建一个C动态库。

我们使用的是之前出现过的例子：

```
// main.go
package main

import "C"
import "fmt"

func main() {}

//export SayHello
func SayHello(name *C.char) {
    fmt.Printf("hello %s!\n", C.GoString(name))
}
```

其中只导出了一个SayHello函数，用于打印字符串。通过以下命令基于上述Go代码创建say-hello.so动态库：

```
go build -buildmode=c-shared -o say-hello.so main.go
```

现在我们就可以通过ctypes模块调用say-hello.so动态库中的SayHello函数了：

```
// hello.py
import ctypes

libso = ctypes.CDLL("./say-hello.so")

SayHello = libso.SayHello
SayHello.argtypes = [ctypes.c_char_p]
SayHello.restype = None

SayHello(ctypes.c_char_p(b"hello"))
```

我们首先通过`ctypes.CDLL`加载动态库到`libso`，并通过`libso.SayHello`来获取`SayHello`函数。获取到`SayHello`函数之后设置函数的输入参数为一个C语言类型的字符串，该函数没有返回值。然后我们通过`ctypes.c_char_p(b"hello")`将Python字节串转为C语言格式的字符串作为参数调用`SayHello`。如果一切正常的话就可以输出字符串了。

从这个例子可以看出，给予`ctypes`构造Python扩展模块非常简单，本质上只是在构建一个纯C语言规格的动态库。比较复杂的部分在`ctypes`的具体使用，关于`ctypes`的具体细节就不详细展开的，用户可以自行参考Python自带的官方文档。

## 基于Python C接口创建

在前面的例子中，通过`ctypes`创建的模块必须要用Python再包装一层，否则就要直接面对C语言风格的接口。如果基于基于Python C接口，我们可以完全再Go和C语言层面创建灵活强大的模块，重点是不再需要在Python中重新包装。

基于Python C接口创建模块和使用C语言的静态库的流程类似：

```

package main

/*
// macOS:
#cgo darwin pkg-config: python3

// linux
#cgo linux pkg-config: python3

// windows
// should generate libpython3.a from python3.lib

#define Py_LIMITED_API
#include <Python.h>

extern PyObject* PyInit_gopkg();
extern PyObject* Py_gopkg_sum(PyObject *, PyObject *);

static int cgo_PyArg_ParseTuple_ii(PyObject *arg, int *a, int *b) {
    return PyArg_ParseTuple(arg, "ii", a, b);
}

static PyObject* cgo_PyInit_gopkg(void) {
    static PyMethodDef methods[] = {
        {"sum", Py_gopkg_sum, METH_VARARGS, "Add two numbers."},
        {NULL, NULL, 0, NULL},
    };
    static struct PyModuleDef module = {
        PyModuleDef_HEAD_INIT, "gopkg", NULL, -1, methods,
    };
    return PyModule_Create(&module);
}
*/
import "C"

func main() {}

//export PyInit_gopkg
func PyInit_gopkg() *C.PyObject {
    return C.cgo_PyInit_gopkg()
}

//export Py_gopkg_sum
func Py_gopkg_sum(self, args *C.PyObject) *C.PyObject {
    var a, b C.int
    if C.cgo_PyArg_ParseTuple_ii(args, &a, &b) == 0 {
        return nil
    }
    return C.PyLong_FromLong(C.long(a + b))
}

```

因为Python的链接参数要复杂了很多，我们借助pkg-config工具来获取编译参数和链接参数。然后我们在Go语言中分别导出了PyInit\_gopkg和Py\_gopkg\_sum函数，其中PyInit\_gopkg函数用于初始化名为gopkg的Python模块，而Py\_gopkg\_sum函数则是模块中sum方法的实现。

因此PyArg\_ParseTuple是可变参数类型，CGO中无法使用可变参数的C函数，因此我们通过增加一个cgo\_PyArg\_ParseTuple\_ii辅助函数小消除可变参数的影响。同样，模块的方法列表必须在C语言内存空间创建，因为CGO是禁止将Go语言内存直接返回到C语言空间的。

然后通过以下命令创建gopkg.so动态库：

```
go build -buildmode=c-shared -o gopkg.so main.go
```

这里需要注意几个出现gopkg名字的地方。gopkg是我们创建的Python模块的名字，因此它对应一个gopkg.so动态库。再gopkg.so动态库中必须有一个PyInit\_gopkg函数，该函数是模块的初始化函数。在PyInit\_gopkg函数初始化模块时，同样需要指定模块的名字时gopkg。模块中的方法函数是通过函数指针访问，具体的名字没有影响。

## macOS环境构建

因为在macOS中，pkg-config不支持Python3版本。不过macOS有一个python3-config的命令可以实现pkg-config类似的功能。不过python3-config生成的编译参数无法直接用于CGO编译选项（因为GCC不能识别部分参数会导致错误构建）。

我们在python3-config的基础只是又包装了一个工具，在通过python3-config获取到编译参数之后将GCC不支持的参数剔除掉。

创建py3-config.go文件：

```
func main() {
    for _, s := range os.Args {
        if s == "--cflags" {
            out, _ := exec.Command("python3-config", "--cflags").CombinedOutput()
            out = bytes.Replace(out, []byte("-arch"), []byte{}, -1)
            out = bytes.Replace(out, []byte("i386"), []byte{}, -1)
            out = bytes.Replace(out, []byte("x86_64"), []byte{}, -1)
            fmt.Println(string(out))
            return
        }
        if s == "--libs" {
            out, _ := exec.Command("python3-config", "--ldflags").CombinedOutput()
            fmt.Println(string(out))
            return
        }
    }
}
```

cgo中的pkg-config只需要两个参数`--cflags`和`--libs`。其中`--libs`选项的输出我们采用的是`python3-config --ldflags`的输出，因为`--libs`选项没有包含库的检索路径，而`--ldflags`选项则是在指定链接库参数的基础上增加了库的检索路径。

基于`py3-config.go`可以创建一个`py3-config`命令。然后通过`PKG_CONFIG`环境变量将cgo使用的`pkg-config`命令指定为我们订制的命令：

```
PKG_CONFIG=./py3-config go build -buildmode=c-shared -o gopkg.so main.go
```

对于不支持`pkg-config`的平台我们都可以基于类似的方法处理。

## 2.8. 补充说明

为何要话费巨大的精力学习CGO是一个问题。任何技术和语言都有它自身的优点和不足，Go语言不是银弹，它无法解决全部问题。而通过CGO可以继承C/C++将近半个世纪的软件遗产，通过CGO可以用Go给其它系统写C接口的共享库，通过CGO技术可以让Go语言编写的代码可以很好地融入现有的软件生态——而现在的软件正式建立在C/C++语言之上的。因此说CGO是一个保底的后备技术，它是Go的一个重量级的替补技术，值得任何一个严肃的Go语言开发人员学习。

本章讨论了CGO的一些常见用法，并给出相关的例子。关于CGO有几点补充：如果有纯Go的解决方法就不要使用CGO；CGO中涉及的C和C++构建问题非常繁琐；CGO有一定的限制无法实现解决全部的问题；不要试图越过CGO的一些限制。而且CGO只是一种官方提供并推荐的Go语言和C/C++交互的方法。如果是使用的gccgo的版本，可以通过gccgo的方式实现Go和C/C++的交互。同时SWIG也是一种选择，并对C++诸多特性提供了支持。

## 第三章 Go汇编语言(TODO)

TODO

## 3.1. 汇编基础(TODO)

TODO

## 3.2. 控制流(TODO)

TODO

### 3.3. 包变量(TODO)

TODO

## 3.4. 叶子函数(TODO)

TODO

## 3.5. 动态栈(TODO)

TODO

## 3.6. 垃圾回收(TODO)

TODO

## 3.7. 例子：图像降采样(TODO)

TODO

## 3.8. 补充说明(TODO)

TODO

## 第四章 移动平台

TODO

## 第六章 go 和 web

本章将会阐述 go 在 web 开发方面的现状，并以几个典型的开源 web 框架为例，带大家深入 web 框架本身的执行流程。

同时会介绍现代企业级 web 开发面临的一些问题，以及在 golang 中如何面对，并解决这些问题。

## 6.1. web 开发简介

由于 golang 的 `net/http` 提供了基础的路由函数组合，并且也提供了丰富的功能函数。所以在 golang 社区里有一种观点认为用 golang 写 api 不需要框架。其看法也存在一定的道理，如果你的项目路由在个位数，URI 固定且不通过 URI 来传递参数，那么使用官方库也就足够。但在复杂场景下，官方的 http 库还是有些力不从心。例如下面这样的路由：

```
GET /card/:id
POST /card/:id
DELETE /card/:id
GET /card/:id/name
...
GET /card/:id/relations
```

可见是否该用框架还是要具体问题具体分析的。

golang 的 web 框架大致可以分为这么两类：

1. router 框架
2. mvc 类框架

在使用哪种框架上，大多数情况下都是看个人的喜好和公司技术人员的背景。例如公司有很多技术人员是 php 出身，那么他们一定会非常喜欢像 beego 这样的框架，但如果公司有很多 C 程序员，那么他们的想法可能是越简单越好。比如很多大厂的 C 程序员可能甚至都会去用 C 去写很小的 CGI 程序，可能本身并没有什么意愿去学习你的 MVC 或者更复杂的 web 框架，他们需要的只是一个非常简单的路由(甚至连路由都不需要，只需要一个基础的 http 协议处理库来帮他省掉没什么意思的体力劳动)。

golang 的 `net/http` 库提供的就是这样基础的功能，写一个 http echo server 只需要三十秒。

```
//brief_intro/echo.go
package main
import (...)

func echo(wr http.ResponseWriter, r *http.Request) {
    msg, err := ioutil.ReadAll(r.Body)
    if err != nil {
        wr.Write([]byte("echo error"))
        return
    }

    writeLen, err := wr.Write(msg)
    if err != nil || writeLen != len(msg) {
        log.Println(err, "write len:", writeLen)
    }
}

func main() {
    http.HandleFunc("/", echo)
    err := http.ListenAndServe(":8080", nil)
    if err != nil {
        log.Fatal(err)
    }
}
```

如果你 30 秒没有完成这个程序，检查一下自己的打字速度是不是慢了。开个玩笑。这个例子是为了说明如果你想写一个 `http` 协议的小程序有多么简单。如果你面临的情况比较复杂，例如几十个接口的企业级应用，直接用 `net/http` 库就显得不太合适了。

我们来看看开源社区中一个 `kafka` 监控项目中的做法：

```
//Burrow: http_server.go
func NewHttpServer(app *ApplicationContext) (*HttpServer, error) {
    ...
    server mux.HandleFunc("/", handleDefault)

    server mux.HandleFunc("/burrow/admin", handleAdmin)

    server mux.Handle("/v2/kafka", appHandler{server.app, handleClusterList})
    server mux.Handle("/v2/kafka/", appHandler{server.app, handleKafka})
    server mux.Handle("/v2/zookeeper", appHandler{server.app, handleClusterList})
    ...
}
```

上面这段代码来自大名鼎鼎的 `linkedin` 公司的 `kafka` 监控项目 `Burrow`，没有使用任何 `router` 框架，只使用了 `net/http`。只看上面这段代码似乎非常优雅，我们的项目里大概只有这五个简单的 `URI`，所以我们提供的服务就是下面这个样子：

```

/
/burrow/admin
/v2/kafka
/v2/kafka/
/v2/zookeeper

```

如果你确实这么想的话就被骗了。我们再进 `handleKafka` 这个函数一探究竟：

```

func handleKafka(app *ApplicationContext, w http.ResponseWriter, r *http.Request) (int, string) {
    pathParts := strings.Split(r.URL.Path[1:], "/")
    if _, ok := app.Config.Kafka[pathParts[2]]; !ok {
        return makeErrorResponse(http.StatusNotFound, "cluster not found", w, r)
    }
    if pathParts[2] == "" {
        // Allow a trailing / on requests
        return handleClusterList(app, w, r)
    }
    if (len(pathParts) == 3) || (pathParts[3] == "") {
        return handleClusterDetail(app, w, r, pathParts[2])
    }

    switch pathParts[3] {
    case "consumer":
        switch {
        case r.Method == "DELETE":
            switch {
            case (len(pathParts) == 5) || (pathParts[5] == ""):
                return handleConsumerDrop(app, w, r, pathParts[2], pathParts[4])
            default:
                return makeErrorResponse(http.StatusMethodNotAllowed, "request method not allowed", w, r)
            }
        case r.Method == "GET":
            switch {
            case (len(pathParts) == 4) || (pathParts[4] == ""):
                return handleConsumerList(app, w, r, pathParts[2])
            case (len(pathParts) == 5) || (pathParts[5] == ""):
                // Consumer detail - list of consumer streams/hosts? Can be config info later
                return makeErrorResponse(http.StatusNotFound, "unknown API call", w, r)
            case pathParts[5] == "topic":
                switch {
                case (len(pathParts) == 6) || (pathParts[6] == ""):
                    return handleConsumerTopicList(app, w, r, pathParts[2], pathParts[4])
                case (len(pathParts) == 7) || (pathParts[7] == ""):
                    return handleConsumerTopicDetail(app, w, r, pathParts[2], pathParts[4])
                }
            case pathParts[5] == "status":
                return handleConsumerStatus(app, w, r, pathParts[2], pathParts[4], false)
            case pathParts[5] == "lag":
                return handleConsumerStatus(app, w, r, pathParts[2], pathParts[4], true)
            }
        default:
    }
}

```

```

        return makeErrorResponse(http.StatusMethodNotAllowed, "request method not sup
    }
    case "topic":
        switch {
        case r.Method != "GET":
            return makeErrorResponse(http.StatusMethodNotAllowed, "request method not sup
        case (len(pathParts) == 4) || (pathParts[4] == ""):
            return handleBrokerTopicList(app, w, r, pathParts[2])
        case (len(pathParts) == 5) || (pathParts[5] == ""):
            return handleBrokerTopicDetail(app, w, r, pathParts[2], pathParts[4])
        }
    case "offsets":
        // Reserving this endpoint to implement later
        return makeErrorResponse(http.StatusNotFound, "unknown API call", w, r)
    }

    // If we fell through, return a 404
    return makeErrorResponse(http.StatusNotFound, "unknown API call", w, r)
}

```

因为默认的 `http` 库中的 `mux` 不支持带参数的路由，Burrow 这个项目使用了非常蹩脚的字符串 `Split` 和乱七八糟的 `switch case` 来达到自己的目的，但实际上却让本来应该很集中的路由管理逻辑变得复杂，散落在系统的各处，难以维护和管理。如果读者细心地看过这些代码之后，可能会发现其它的几个 `handler` 函数逻辑上较简单，最复杂的也就是这个 `handleKafka`。但实际上我们的系统总是从这样微不足道的混乱开始积少成多，最终变得难以收拾。

简单地来说，只要你的路由带有参数，并且这个项目的 api 数目超过了 10，就尽量不要使用 `net/http` 中默认的路由。在 golang 开源圈应用最广泛的 router 是 `httpRouter`，很多开源的 router 框架都是基于 `httpRouter` 进行一定程度的改造。关于 `httpRouter` 路由的原理，会在本章节的 `router` 一节中进行详细的阐释。

再来说一下文章开头说的，开源界有这么几种框架，第一种是对 `httpRouter` 进行简单的封装，然后提供定制的 `middleware` 和一些简单的小工具集成比如 `gin`，主打轻量，易学，高性能。第二种是借鉴其它语言的编程风格的一些 `MVC` 类框架，例如 `beego`，方便从其它语言迁移过来的程序员快速上手，快速开发。还有一些框架功能更为强大，除了 `db` 设计，大部分代码直接生成，例如 `goa`。不管哪种框架，适合读者背景的就是最好的。

本章的内容除了会展开讲解 `router` 和 `middleware` 的原理，还会以现在工程界面临的问题结合 `golang` 来进行一些实践性的说明。希望没有接触过相关内容的读者能够有所受用。

## 6.2. router 请求路由

在常见的 web 框架中，router 是必备的组件。golang 圈子里 router 也时常被称为 http 的 multiplexer。在上一节中我们通过对 Burrow 代码的简单学习，已经知道如何用 http 标准库中内置的 mux 来完成简单的路由功能了。如果开发 web 系统对路径中带参数没什么兴趣的话，用 http 标准库中的 mux 就可以。

restful 是几年前刮起的 API 设计风潮，在 restful 中使用了 http 标准库还没有支持的一些语义。来看看 restful 中常见的请求路径：

```
GET /repos/:owner/:repo/comments/:id/reactions
POST /projects/:project_id/columns
PUT /user/starred/:owner/:repo
DELETE /user/starred/:owner/:repo
```

相信聪明的你已经猜出来了，这是 github 官方文档中挑出来的几个 api 设计。restful 风格的 API 重度依赖请求路径。会将很多参数放在请求 URI 中。除此之外还会使用很多并不那么常见的 HTTP 状态码，不过本节只讨论路由，所以先略过不谈。

如果我们的系统也想要这样的 URI 设计，使用标准库的 mux 显然就力不从心了。

## httprouter

较流行的开源 golang web 框架大多使用 httprouter，或是基于 httprouter 的变种对路由进行支持。前面提到的 github 的参数式路由在 httprouter 中都是可以支持的。

因为 httprouter 中使用的是显式匹配，所以在设计路由的时候需要规避一些会导致路由冲突的情况，例如：

```
conflict:
GET /user/info/:name
GET /user/:id

no conflict:
GET /user/info/:name
POST /user/:id
```

简单来讲的话，如果两个路由拥有一致的 http method (指 GET/POST/PUT/DELETE) 和请求路径前缀，且在某个位置出现了 A 路由是 wildcard (指 :id 这种形式) 参数，B 路由则是普通字符串，那么就会发生路由冲突。路由冲突会在初始化阶段直接 panic：

```
panic: wildcard route ':id' conflicts with existing children in path '/user/:id'

goroutine 1 [running]:
github.com/cch123/httprouter.(*node).insertChild(0xc4200801e0, 0xc42004fc01, 0x126b177, 0
    /Users/caochunhui/go_work/src/github.com/cch123/httprouter/tree.go:256 +0x841
github.com/cch123/httprouter.(*node).addRoute(0xc4200801e0, 0x126b171, 0x9, 0x127b668)
    /Users/caochunhui/go_work/src/github.com/cch123/httprouter/tree.go:221 +0x22a
github.com/cch123/httprouter.(*Router).Handle(0xc42004ff38, 0x126a39b, 0x3, 0x126b171, 0x
    /Users/caochunhui/go_work/src/github.com/cch123/httprouter/router.go:262 +0xc3
github.com/cch123/httprouter.(*Router).GET(0xc42004ff38, 0x126b171, 0x9, 0x127b668)
    /Users/caochunhui/go_work/src/github.com/cch123/httprouter/router.go:193 +0x5e
main.main()
    /Users/caochunhui/test/go_web/httprouter_learn2.go:18 +0xaf
exit status 2
```

还有一点需要注意，因为 `httprouter` 考虑到字典树的深度，在初始化时会对参数的数量进行限制，所以在路由中的参数数目不能超过 255，否则会导致 `httprouter` 无法识别后续的参数。不过这一点上也不用考虑太多，毕竟 URI 是人设计且给人来看的，相信没有变态的 URI 能在一条路径中带有 200 个以上的参数。

除支持路径中的 wildcard 参数之外，`httprouter` 还可以支持 \* 号来进行通配，不过 \* 号开头的参数只能放在路由的结尾，例如下面这样：

```
Pattern: /src/*filepath

/src/                 filepath = ""
/src/somefile.go      filepath = "somefile.go"
/src/subdir/somefile.go  filepath = "subdir/somefile.go"
```

这种设计在 restful 中可能不太常见，主要是为了能够使用 `httprouter` 来做简单的 http 静态文件服务器。

除了正常情况下的路由支持，`httprouter` 也支持对一些特殊情况下的回调函数进行定制，例如 404 的时候：

```
r := httprouter.New()
r.NotFound = http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("oh no, not found"))
})
```

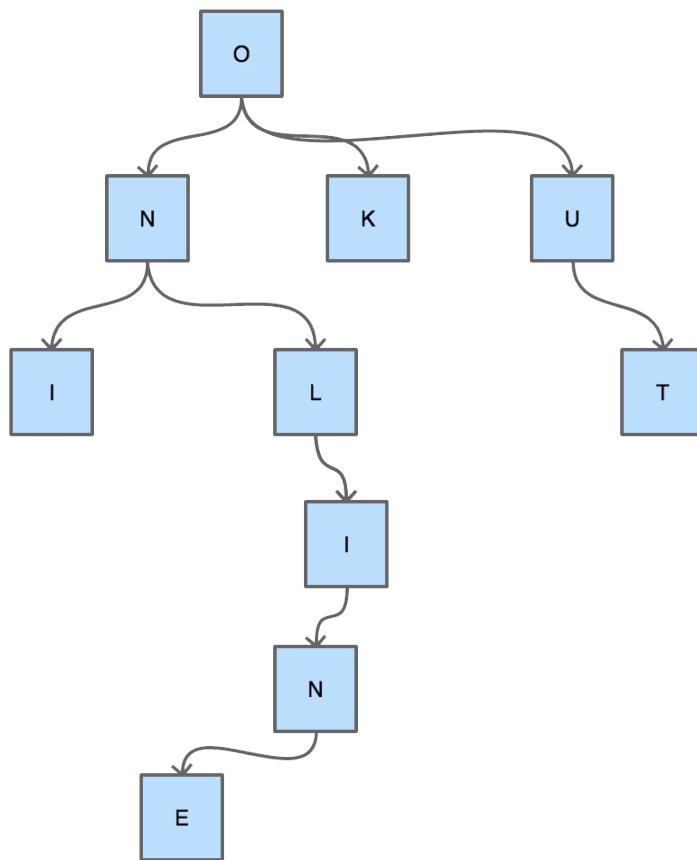
或者内部 panic 的时候：

```
r.PanicHandler = func(w http.ResponseWriter, r *http.Request, c interface{}) {
    log.Printf("Recovering from panic, Reason: %#v", c.(error))
    w.WriteHeader(http.StatusInternalServerError)
    w.Write([]byte(c.(error).Error()))
}
```

目前开源界最为流行(star 数最多)的 web 框架 [gin](#) 使用的就是 `httprouter` 的变种。

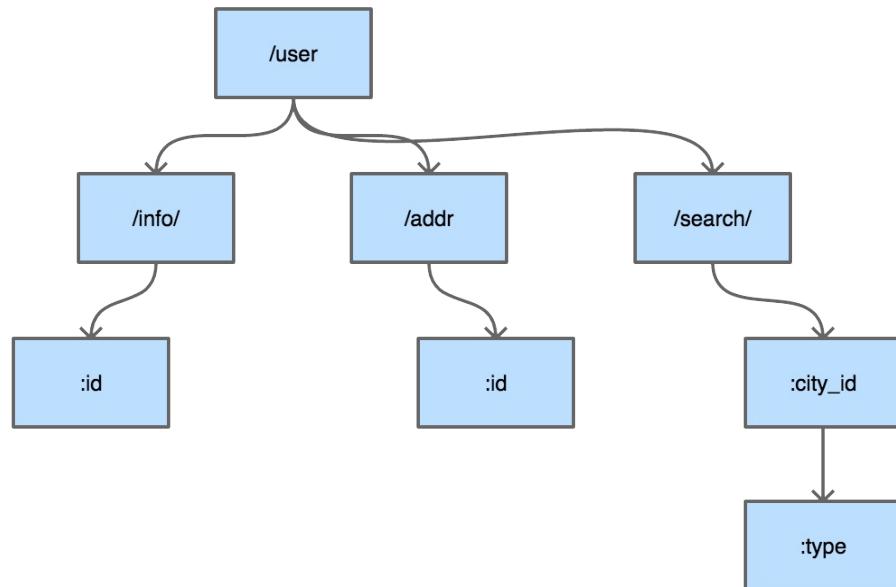
## 原理

`httprouter` 和众多衍生 `router` 使用的数据结构被称为 `radix tree`，压缩字典树。读者可能没有接触过压缩字典树，但对字典树 `trie tree` 应该有所耳闻。下图是一个典型的字典树结构：



字典树常用来进行字符串检索，例如用给定的字符串序列建立字典树。对于目标字符串，只要从根节点开始深度优先搜索，即可判断出该字符串是否曾经出现过，时间复杂度为  $O(n)$ ， $n$  可以认为是目标字符串的长度。为什么要这样做？字符串本身不像数值类型可以进行数值比较，两个字符串对比的时间复杂度取决于字符串长度。如果不用字典树来完成上述功能，要对历史字符串进行排序，再利用二分查找之类的算法去搜索，时间复杂度只高不低。可认为字典树是一种空间换时间的典型做法。

普通的字典树有一个比较明显的缺点，就是每个字母都需要建立一个孩子节点，这样会导致字典树的层树比较深，压缩字典树相对好地平衡了字典树的优点和缺点。下图是典型的压缩字典树结构：



每个节点上不只存储一个字母了，这也是压缩字典树中“压缩”的主要含义。使用压缩字典树可以减少树的层数，同时因为每个节点上数据存储也比通常的字典树要多，所以程序的局部性较好(一个节点的 path 加载到 cache 即可进行多个字符的对比)，从而对 CPU 缓存友好。

## 压缩字典树创建过程

我们来跟踪一下 `httprouter` 中，一个典型的压缩字典树的创建过程，路由设定如下：

```

PUT /user/installations/:installation_id/repositories/:repository_id

GET /marketplace_listing/plans/
GET /marketplace_listing/plans/:id/accounts
GET /search
GET /status
GET /support

补充路由：
GET /marketplace_listing/plans/ohyes

```

最后一条补充路由是我们臆想的，除此之外所有 API 路由均来自于 [api.github.com](https://api.github.com)。

## root 节点创建

`httprouter` 的 `Router struct` 中存储压缩字典树使用的是下述数据结构：

```

// 略去了其它部分的 Router struct
type Router struct {
    // ...
    trees map[string]*node
    // ...
}

```

`trees` 中的 `key` 即为 `http 1.1` 的 `RFC` 中定义的各种 `method`，具体有：

```

GET
HEAD
OPTIONS
POST
PUT
PATCH
DELETE

```

每一种 `method` 对应的都是一棵独立的压缩字典树，这些树彼此之间不共享数据。具体到我们上面用到的路由，`PUT` 和 `GET` 是两棵树而非一棵。

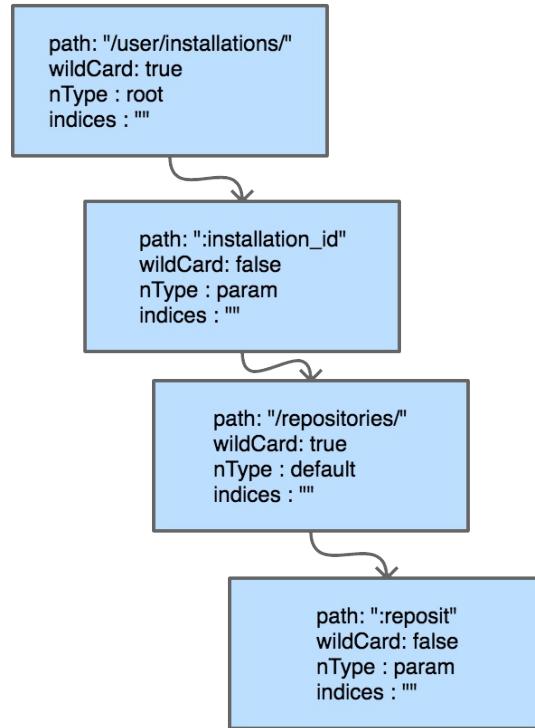
简单来讲，某个 `method` 第一次插入的路由就会导致对应字典树的根节点被创建，我们按顺序，先是一个 `PUT`：

```

r := httprouter.New()
r.PUT("/user/installations/:installation_id/repositories/:repository_id", Hello)

```

这样 PUT 对应的根节点就会被创建出来。把这棵 PUT 的树画出来：



radix 的节点类型为 `*httprouter.node`，为了说明方便，我们留下了目前关心的几个字段：

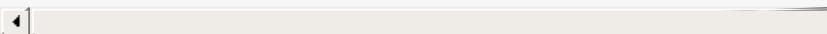
`path`: 当前节点对应的路径中的字符串

`wildCard`: 子节点是否为参数节点，即 `wildCard node`，或者说 `:id` 这种类型的节点

`nType`: 当前节点类型，有四个枚举值：分别为 `static/root/param/catchAll`。

static	// 非根节点的普通字符串节点
root	// 根节点
param	// 参数节点，例如 <code>:id</code>
catchAll	// 通配符节点，例如 <code>*anyway</code>

`indices`: 子节点索引，当子节点为非参数类型，即本节点的 `wildCard` 为 `false` 时，会将每个子节点的首字母放在



当然，PUT 路由只有唯一的一条路径。接下来，我们以后续的多条 GET 路径为例，讲解子节点的插入过程。

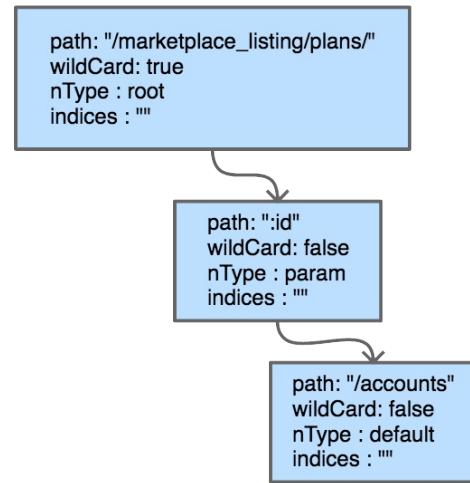
## 子节点插入

当插入 `GET /marketplace_listing/plans` 时，类似前面 `PUT` 的过程，`GET` 树的结构如图所示：

```
path: "/marketplace_listing/plans/"
wildCard: false
nType : root
indices : ""
```

因为第一个路由没有参数，`path` 都被存储到根节点上了。所以只有一个节点。

然后插入 `GET /marketplace_listing/plans/:id/accounts`，新的路径与之前的路径有共同的前缀，且可以直接在之前叶子节点后进行插入，那么结果也很简单，插入后树变成了这样：

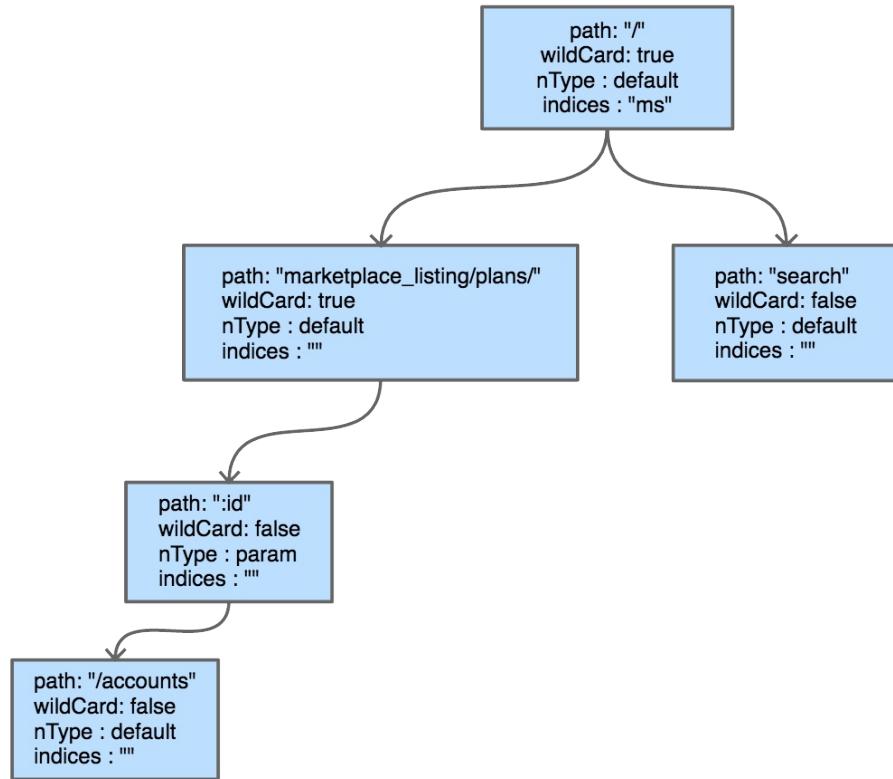


由于 `:id` 这个节点只有一个字符串的普通子节点，所以 `indices` 还依然不需要处理。

上面这种情况比较简单，新的路由可以直接作为原路由的子节点进行插入。实际情况不会这么美好。

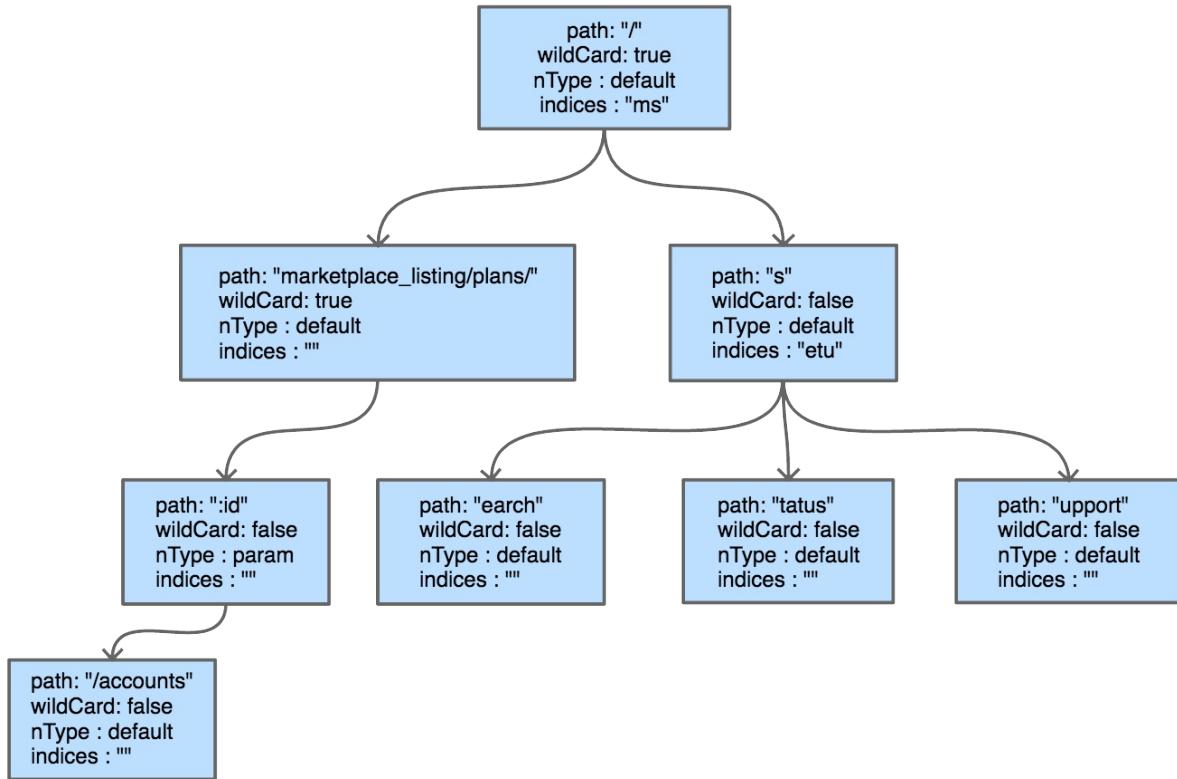
## 边分裂

接下来我们插入 `GET /search`，这时会导致树的边分裂。



原有路径和新的路径在初始的 / 位置发生分裂，这样需要把原有的 root 节点内容下移，再将新路由 search 同样作为子节点挂在 root 节点之下。这时候因为子节点出现多个，root 节点的 indices 提供子节点索引，这时候该字段就需要派上用场了。“ms” 代表子节点的首字母分别为 m(markplace) 和 s(search)。

我们一口作气，把 GET /status 和 GET /support 也插入到树中。这时候会导致在 search 节点上再次发生分裂，来看看最终的结果：



## 子节点冲突处理

在路由本身只有字符串的情况下，不会发生任何冲突。只有当路由中含有 `wildcard`(类似 `:id`) 或者 `catchAll` 的情况下才可能冲突。这一点在前面已经提到了。

子节点的冲突处理很简单，分几种情况：

1. 在插入 `wildcard` 节点时，父节点的 `children` 数组非空且 `wildCard` 被设置为 `false`。例如：`GET /user/getAll` 和 `GET /user/:id/getAddr`，或者 `GET /user/*aaa` 和 `GET /user/:id`。
2. 在插入 `wildcard` 节点时，父节点的 `children` 数组非空且 `wildCard` 被设置为 `true`，但该父节点的 `wildCard` 子节点要插入的 `wildCard` 名字不一样。例如：`GET /user/:id/info` 和 `GET /user/:name/info`。
3. 在插入 `catchAll` 节点时，父节点的 `children` 非空。例如：`GET /src/abc` 和 `GET /src/*filename`，或者 `GET /src/:id` 和 `GET /src/*filename`。
4. 在插入 `static` 节点时，父节点的 `wildCard` 字段被设置为 `true`。
5. 在插入 `static` 节点时，父节点的 `children` 非空，且子节点 `nType` 为 `catchAll`。

只要发生冲突，都会在初始化的时候 `panic`。



## 6.3. middleware 中间件

本章将对现在流行的 web 框架中的中间件技术原理进行分析，并介绍如何使用中间件技术将业务和非业务代码功能进行解耦。

### 代码泥潭

先来看一段代码：

```
// middleware/hello.go
package main

func hello(wr http.ResponseWriter, r *http.Request) {
    wr.Write([]byte("hello"))
}

func main() {
    http.HandleFunc("/", hello)
    err := http.ListenAndServe(":8080", nil)
    ...
}
```

这是一个典型的 web 服务，挂载了一个简单的路由。我们的线上服务一般也是从这样简单的服务开始逐渐拓展开去的。

现在突然来了一个新的需求，我们想要统计之前写的 hello 服务的处理耗时，需求很简单，我们对上面的程序进行少量修改：

```
// middleware/hello_with_time_elapse.go
var logger = log.New(os.Stdout, "", 0)

func hello(wr http.ResponseWriter, r *http.Request) {
    timeStart := time.Now()
    wr.Write([]byte("hello"))
    timeElapsed := time.Since(timeStart)
    logger.Println(timeElapsed)
}
```

这样便可以在每次接收到 http 请求时，打印出当前请求所消耗的时间。

完成了这个需求之后，我们继续进行业务开发，提供的 api 逐渐增加，现在我们的路由看起来是这个样子：

```
// middleware/hello_with_more_routes.go
// 省略了一些相同的代码
package main

func helloHandler(wr http.ResponseWriter, r *http.Request) {
    ...
}

func showInfoHandler(wr http.ResponseWriter, r *http.Request) {
    ...
}

func showEmailHandler(wr http.ResponseWriter, r *http.Request) {
    ...
}

func showFriendsHandler(wr http.ResponseWriter, r *http.Request) {
    timeStart := time.Now()
    wr.Write([]byte("your friends is tom and alex"))
    timeElapsed := time.Since(timeStart)
    logger.Println(timeElapsed)
}

func main() {
    http.HandleFunc("/", helloHandler)
    http.HandleFunc("/info/show", showInfoHandler)
    http.HandleFunc("/email/show", showEmailHandler)
    http.HandleFunc("/friends/show", showFriendsHandler)
    ...
}
```

每一个 `handler` 里都有之前提到的记录运行时间的代码，每次增加新的路由我们也同样需要把这些看起来长得差不多的代码拷贝到我们需要的地方去。因为代码不太多，所以实施起来也没有遇到什么大问题。

渐渐的我们的系统增加到了 30 个路由和 `handler` 函数，每次增加新的 `handler`，我们的第一件工作就是把之前写的所有和业务逻辑无关的周边代码先拷贝过来。

接下来系统安稳地运行了一段时间，突然有一天，老板找到你，我们最近找人新开发了监控系统，为了系统运行可以更加可控，需要把每个接口运行的耗时数据主动上报到我们的监控系统里。给监控系统起个名字吧，叫 `metrics`。现在你需要修改代码并把耗时通过 `http post` 的方式发给 `metrics` 了。我们来修改一下 `helloHandler`：

```
func helloHandler(wr http.ResponseWriter, r *http.Request) {
    timeStart := time.Now()
    wr.Write([]byte("hello"))
    timeElapsed := time.Since(timeStart)
    logger.Println(timeElapsed)
    // 新增耗时上报
    metrics.Upload("timeHandler", timeElapsed)
}
```

修改到这里，本能地发现我们的开发工作开始陷入了泥潭。无论未来对我们的这个 web 系统有任何其它的非功能或统计需求，我们的修改必然牵一发而动全身。只要增加一个非常简单的非业务统计，我们就需要去几十个 handler 里增加这些业务无关的代码。虽然一开始我们似乎并没有做错，但是显然随着业务的发展，我们的行事方式让我们陷入了代码的泥潭。

## 使用 **middleware** 剥离非业务逻辑

我们来分析一下，一开始在哪里做错了呢？我们只是一步一步地满足需求，把我们需要的逻辑按照流程写下去呀？

实际上，我们犯的最大的错误是把业务代码和非业务代码揉在了一起。对于大多数的场景来讲，非业务的需求都是在 http 请求处理前做一些事情，或者/并且在响应完成之后做一些事情。我们有没有办法使用一些重构思路把这些公共的非业务功能代码剥离出去呢？回到刚开头的例子，我们需要给我们的 helloHandler 增加超时时间统计，我们可以使用一种叫

`function adapter` 的方法来对 helloHandler 进行包装：

```

func hello(wr http.ResponseWriter, r *http.Request) {
    wr.Write([]byte("hello"))
}

func timeMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(wr http.ResponseWriter, r *http.Request) {
        timeStart := time.Now()

        // next handler
        next.ServeHTTP(wr, r)

        timeElapsed := time.Since(timeStart)
        logger.Println(timeElapsed)
    })
}

func main() {
    http.HandleFunc("/", timeMiddleware(hello))
    err := http.ListenAndServe(":8080", nil)
    ...
}

```

这样就非常轻松地实现了业务与非业务之间的剥离，魔法就在于这个 `timeMiddleware`。可以从代码中看到，我们的 `timeMiddleware` 也是一个函数，其参数为 `http.Handler`，`http.Handler` 的定义在 `net/http` 包中：

```

type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}

```

任何方法实现了 `ServeHTTP`，即是一个合法的 `http.Handler`，读到这里你可能会有一些混乱，我们先来梳理一下 `http` 库的 `Handler`，`HandlerFunc` 和 `ServeHTTP` 的关系：

```

type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}

type HandlerFunc func(ResponseWriter, *Request)

func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request)
    f(w, r)
}

```

实际上只要你的 `handler` 函数签名是：

```
func (ResponseWriter, *Request)
```

那么这个 `handler` 和 `http.HandlerFunc` 就有了一致的函数签名，可以将该 `handler` 函数进行类型转换，转为 `http.HandlerFunc`。而 `http.HandlerFunc` 实现了 `http.Handler` 这个接口。在 `http` 库需要调用你的 `handler` 函数来处理 `http` 请求时，会调用 `HandlerFunc` 的 `ServeHTTP` 函数，可见一个请求的基本调用链是这样的：

```
h = getHandler() => h.ServeHTTP(w, r) => h(w, r)
```

上面提到的把自定义 `handler` 转换为 `http.HandlerFunc` 这个过程是必须的，因为我们的 `handler` 没有直接实现 `ServeHTTP` 这个接口。上面的代码中我们看到的 `HandleFunc`(注意 `HandlerFunc` 和 `HandleFunc` 的区别)里也可以看到这个强制转换过程：

```
func HandleFunc(pattern string, handler func(ResponseWriter, *Request)) {
    DefaultServeMux.HandleFunc(pattern, handler)
}

// 调用

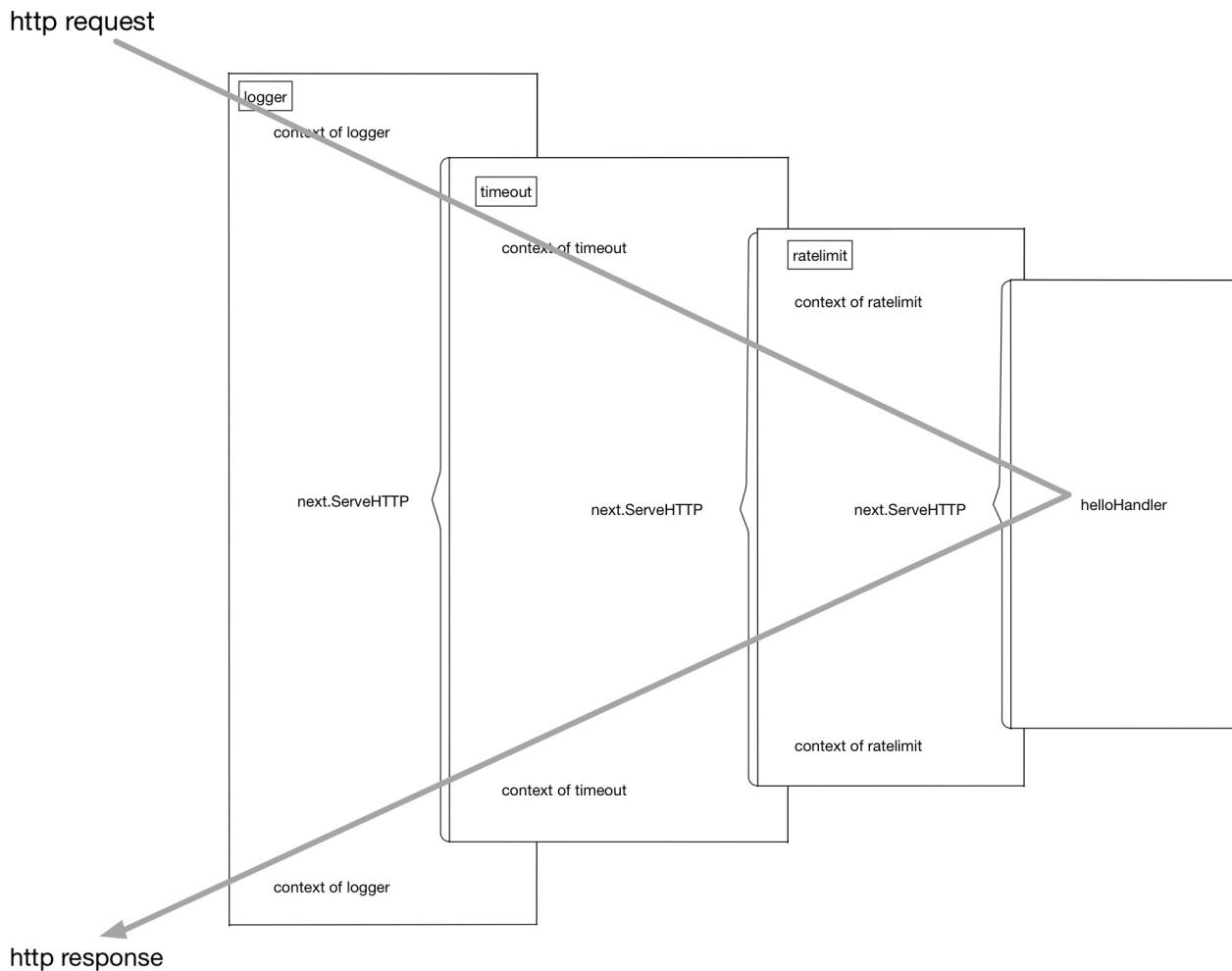
func (mux *ServeMux) HandleFunc(pattern string, handler func(ResponseWriter, *Request)) {
    mux.Handle(pattern, HandlerFunc(handler))
}
```

知道 `handler` 是怎么回事，我们的中间件通过包装 `handler`，再返回一个新的 `handler` 就好理解了。

总结一下，我们的中间件要做的事情就是通过一个或多个函数对 `handler` 进行包装，返回一个包括了各个中间件逻辑的函数链。我们把上面的包装再做得复杂一些：

```
customizedHandler = logger(timeout(ratelimit(helloHandler)))
```

这个函数链在执行过程中的上下文可以用下面这张图来表示。



再直白一些，这个流程在进行请求处理的时候实际上就是不断地进行函数压栈再出栈，有一些类似于递归的执行流：

[exec of logger logic]	函数栈: []
[exec of timeout logic]	函数栈: [logger]
[exec of ratelimit logic]	函数栈: [timeout/logger]
[exec of helloHandler logic]	函数栈: [ratelimit/timeout/logger]
[exec of ratelimit logic part2]	函数栈: [timeout/logger]
[exec of timeout logic part2]	函数栈: [logger]
[exec of logger logic part2]	函数栈: []

功能实现了，但在上面的使用过程中我们也看到了，这种函数套函数的用法不是很美观，同时也不具备什么可读性。

## 更优雅的 **middleware** 写法

上一节中解决了业务功能代码和非业务功能代码的解耦，但也提到了，看起来并不美观，如果需要修改这些函数的顺序，或者增删 **middleware** 还是有点费劲，本节我们来进行一些“写法”上的优化。

看一个例子：

```
r = NewRouter()
r.Use(logger)
r.Use(timeout)
r.Use(ratelimit)
r.Add("/", helloHandler)
```

通过多步设置，我们拥有了和上一节差不多的执行函数链。胜在直观易懂，如果我们要增加或者删除 **middleware**，只要简单地增加删除对应的 **Use** 调用就可以了。非常方便。

从框架的角度来讲，怎么实现这样的功能呢？也不复杂：

```
type middleware func(http.Handler) http.Handler

type Router struct {
    middlewareChain [] func(http.Handler) http.Handler
    mux map[string] http.Handler
}

func NewRouter() *Router{
    return &Router{}
}

func (r *Router) Use(m middleware) {
    r.middlewareChain = append(r.middlewareChain, m)
}

func (r *Router) Add(route string, h http.Handler) {
    var mergedHandler = h

    for i := len(r.middlewareChain) - 1; i >= 0; i-- {
        mergedHandler = r.middlewareChain[i](mergedHandler)
    }

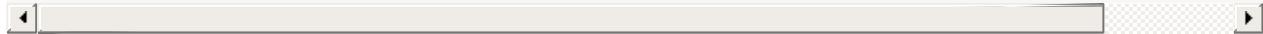
    r.mux[route] = mergedHandler
}
```

注意代码中的 **middleware** 数组遍历顺序，和用户希望的调用顺序应该是“相反”的。应该不难理解。

## 哪些事情适合在 **middleware** 中做

以较流行的开源 golang 框架 chi 为例：

```
compress.go  
=> 对 http 的 response body 进行压缩处理  
heartbeat.go  
=> 设置一个特殊的路由，例如 /ping, /healthcheck，用来给 load balancer 一类的前置服务进行探活  
logger.go  
=> 打印 request 处理日志，例如请求处理时间，请求路由  
profiler.go  
=> 挂载 pprof 需要的路由，如 /pprof、/pprof/trace 到系统中  
realip.go  
=> 从请求头中读取 X-Forwarded-For 和 X-Real-IP，将 http.Request 中的 RemoteAddr 修改为得到的  
requestid.go  
=> 为本次请求生成单独的 requestid，可一路透传，用来生成分布式调用链路，也可用于在日志中串连单次请求的  
timeout.go  
=> 用 context.Timeout 设置超时时间，并将其通过 http.Request 一路透传下去  
throttler.go  
=> 通过定长大小的 channel 存储 token，并通过这些 token 对接口进行限流
```



每一个 web 框架都会有对应的 **middleware** 组件，如果你有兴趣，也可以向这些项目贡献有用的 **middleware**，只要合理一般项目的维护人也愿意合并你的 pull request。

比如开源界很火的 gin 这个框架，就专门为用户贡献的 **middleware** 开了一个仓库：

Collection of middlewares created by the community <https://gin-gonic.github.io/gin/>

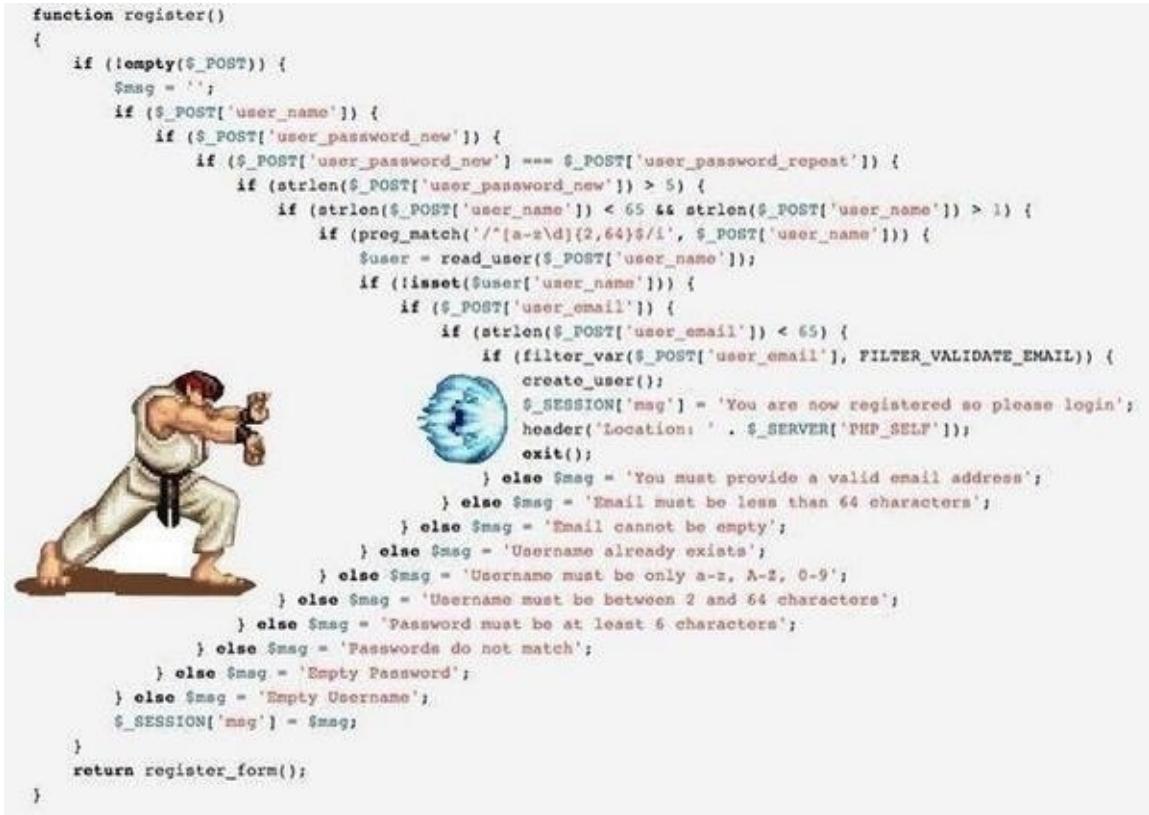
Branch: master ▾ New pull request Create new file Upload files Find file

File	Description
<code>cache</code>	[ci skip] Add EOL-warning.
<code>commonlog</code>	Add missing user identifier part
<code>cors</code>	[ci skip] Add EOL-warning.
<code>expvar</code>	[ci skip] update readme.
<code>ginrus</code>	Fixed ginrus example code
<code>gzip</code>	[ci skip] Add EOL-warning.
<code>jwt</code>	fixes broken tests (#124)
<code>newrelic</code>	Fixes contrib for Gin v1.0
<code>renders/multitemplate</code>	add EOL warning on multitemplate package.

如果读者去阅读 gin 的源码的话，可能会发现 gin 的 middleware 中处理的并不是 `http.Handler`，而是一个叫 `gin.HandlerFunc` 的函数类型，和本节中讲解的 `http.Handler` 签名并不一样。不过实际上 gin 的 handler 也只是针对其框架的一种封装，middleware 的原理与本节中的说明是一致的。

## 6.4. validator 请求校验

社区里曾经有人用这张图来嘲笑 PHP：



实际上这是一个语言无关的场景，需要进行字段校验的情况有很多，web 系统的 Form/json 提交只是一个典型的例子。我们用 go 来写一个类似上图的校验 demo。然后研究怎么一步步对其进行改进。

### 重构请求校验函数

假设我们的数据已经通过某个 binding 库绑定到了具体的 struct 上。

```

type RegisterReq struct {
    Username      string `json:"username"`
    PasswordNew   string `json:"password_new"`
    PasswordRepeat string `json:"password_repeat"`
    Email         string `json:"email"`
}

func register(req RegisterReq) error{
    if len(req.Username) > 0 {
        if len(req.PasswordNew) > 0 && len(req.PasswordRepeat) > 0 {
            if req.PasswordNew == req.PasswordRepeat {
                if emailFormatValid(req.Email) {
                    createUser()
                    return nil
                } else {
                    return errors.New("invalid email")
                }
            } else {
                return errors.New("password and reinput must be the same")
            }
        } else {
            return errors.New("password and password reinput must be longer than 0")
        }
    } else {
        return errors.New("length of username cannot be 0")
    }
}

```

我们在 golang 里成功写出了 hadoken 开路的箭头型代码。。这种代码一般怎么进行优化呢？

很简单，在《重构》一书中已经给出了方案：[Guard Clauses](#)。

```

func register(req RegisterReq) error{
    if len(req.Username) == 0 {
        return errors.New("length of username cannot be 0")
    }

    if len(req.PasswordNew) == 0 || len(req.PasswordRepeat) == 0 {
        return errors.New("password and password reinput must be longer than 0")
    }

    if req.PasswordNew != req.PasswordRepeat {
        return errors.New("password and reinput must be the same")
    }

    if emailFormatValid(req.Email) {
        return errors.New("invalid email")
    }

    createUser()
    return nil
}

```

代码更清爽，看起来也不那么别扭了。这是比较通用的重构理念。虽然使用了重构方法使我们的 validate 过程看起来优雅了，但我们还是得为每一个 http 请求都去写这么一套差不多的 validate 函数，有没有更好的办法来帮助我们解除这项体力劳动？答案就是 validator。

## 用 **validator** 解放体力劳动

从设计的角度讲，我们一定会为每个请求都声明一个 struct。前文中提到的校验场景我们都可以通过 validator 完成工作。还以前文中的 struct 为例。为了美观起见，我们先把 json tag 省略掉。

这里我们引入一个新的 validator 库：

<https://github.com/go-playground/validator>

```

import "gopkg.in/go-playground/validator.v9"

type RegisterReq struct {
    // 字符串的 gt=0 表示长度必须 > 0, gt = greater than
    Username      string   `validate:"gt=0"`
    // 同上
    PasswordNew   string   `validate:"gt=0"`
    // eqfield 跨字段相等校验
    PasswordRepeat string   `validate:"eqfield=PasswordNew"`
    // 合法 email 格式校验
    Email         string   `validate:"email"`
}

func validate(req RegisterReq) error {
    err := validate.Struct(mystruct)
    if err != nil {
        doSomething()
    }
    ...
}

```

这样就不需要在每个请求进入业务逻辑之前都写重复的 `validate` 函数了。本例中只列出了这个 `validator` 非常简单的几个功能。

我们试着跑一下这个程序，输入参数设置为：

```

// ...

var req = RegisterReq {
    Username      : "Xargin",
    PasswordNew   : "ohno",
    PasswordRepeat : "ohn",
    Email         : "alex@abc.com",
}

err := validate.Struct(mystruct)
fmt.Println(err) // Key: 'RegisterReq.PasswordRepeat' Error:Field validation for 'Passwor

```

如果觉得这个 `validator` 提供的错误信息不够人性化，例如要把错误信息返回给用户，那就不应该直接显示英文了。可以针对每种 `tag` 进行错误信息订制，读者可以自行探索。

## 原理

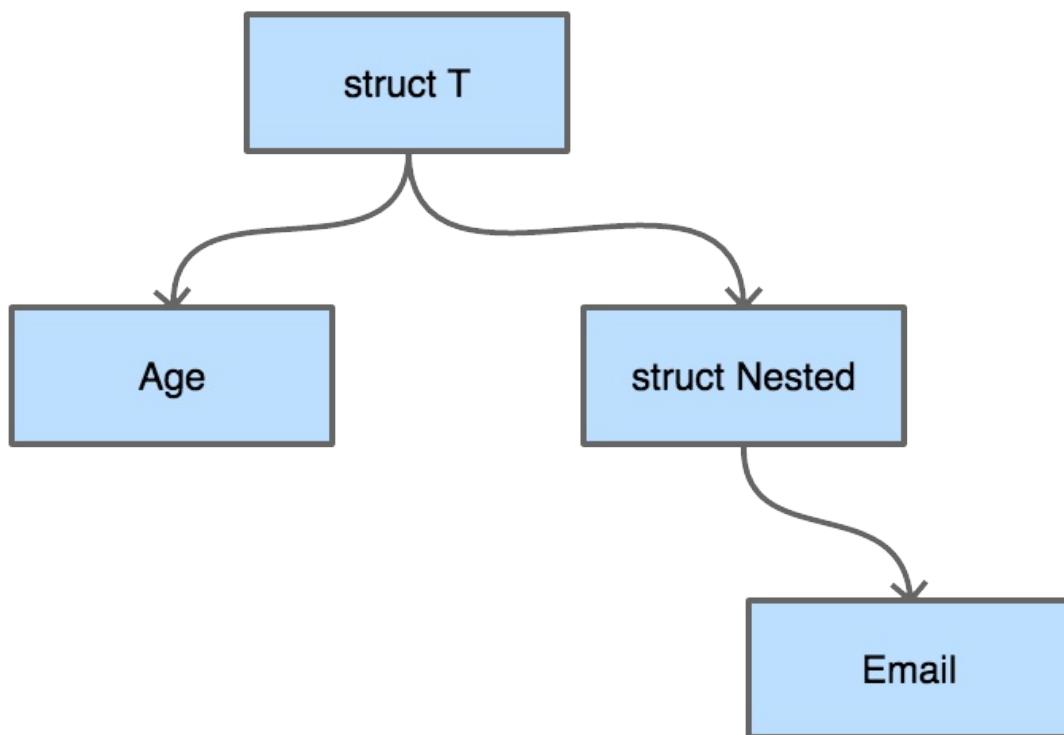
从结构上来看，每一个 `struct` 都可以看成是一棵树。假如我们有如下定义的 `struct`：

```

type Nested struct {
    Email string `validate:"email"`
}
type T struct {
    Age     int `validate:"eq=10"`
    Nested Nested
}

```

把这个 struct 画成一棵树：



从字段校验的需求来讲，无论我们采用深度优先搜索还是广度优先搜索来对这棵 struct 树来进行遍历，都是可以的。

我们来写一个递归的深度优先搜索方式的遍历 demo：

```

package main

import (
    "fmt"
    "reflect"
)

```

```

    "regexp"
    "strconv"
    "strings"
)

type Nested struct {
    Email string `validate:"email"`
}

type T struct {
    Age     int `validate:"eq=10"`
    Nested Nested
}

func validateEmail(input string) bool {
    if pass, _ := regexp.MatchString(`^([\w\.\_]{2,10})@(\w{1,}).([a-z]{2,4})$`, input);
        return true
    }
    return false
}

func validate(v interface{}) (bool, string) {
    validateResult := true
    errmsg := "success"
    vt := reflect.TypeOf(v)
    vv := reflect.ValueOf(v)
    for i := 0; i < vv.NumField(); i++ {
        fieldVal := vv.Field(i)
        tagContent := vt.Field(i).Tag.Get("validate")
        k := fieldVal.Kind()

        switch k {
        case reflect.Int:
            val := fieldVal.Int()
            tagValStr := strings.Split(tagContent, "=")
            tagVal, _ := strconv.ParseInt(tagValStr[1], 10, 64)
            if val != tagVal {
                errmsg = "validate int failed, tag is: "+ tagVal
                return false
            }
        case reflect.String:
            val := fieldVal.String()
            tagValStr := tagContent
            switch tagValStr {
            case "email":
                nestedResult := validateEmail(val)
                if nestedResult == false {
                    errmsg = "validate mail failed, field val is: "+ val
                    validateResult = false
                }
            }
        case reflect.Struct:
            // 如果有内嵌的 struct，那么深度优先遍历
            // 就是一个递归过程
    }
}

```

```

        valInter := fieldVal.Interface()
        nestedResult := validate(valInter)
        if nestedResult == false {
            validateResult = false
        }
    }
}

return validateResult
}

func main() {
    var a = T{Age: 10, Nested: Nested{Email: "abc@abc.com"}}

    validateResult := validate(a)
    fmt.Println(validateResult)
}

```

这里我们简单地对 `eq=x` 和 `email` 这两个 `tag` 进行了支持，读者可以对这个程序进行简单的修改以查看具体的 `validate` 效果。为了演示精简掉了错误处理和复杂 `case` 的处理，例如 `reflect.Int8/16/32/64`，`reflect.Ptr` 等类型的处理，如果给生产环境编写 `validate` 库的话，请务必做好功能的完善和容错。

在前一小节中介绍的 `validator` 组件在功能上要远比我们这里的 `demo` 复杂的多。但原理很简单，就是用 `reflect` 对 `struct` 进行树形遍历。有心的读者这时候可能会产生一个问题，我们对 `struct` 进行 `validate` 时大量使用了 `reflect`，而 `go` 的 `reflect` 在性能上不太出众，有时甚至会影响到我们程序的性能。这样的考虑确实有一些道理，但需要对 `struct` 进行大量校验的场景往往出现在 `web` 服务，这里并不一定是程序的性能瓶颈所在，实际的效果还是要从 `pprof` 中做更精确的判断。

如果基于反射的 `validator` 真的成为了你服务的性能瓶颈怎么办？现在也有一种思路可以避免反射：使用 `golang` 内置的 `parser` 对源代码进行扫描，然后根据 `struct` 的定义生成校验代码。我们可以将所有需要校验的结构体放在单独的 `package` 内。这就交给读者自己去探索了。

## 6.4. Database 和数据库打交道

本节将对 `db/sql` 官方标准库作一些简单分析，并介绍一些应用比较广泛的开源 `orm` 和 `sql builder`。并从企业级应用开发和公司架构的角度来分析哪种技术栈对于现代的企业级应用更为合适。

### 从 `database/sql` 讲起

golang 官方提供了 `database/sql` 包来给用户进行和数据库打交道的工作，实际上 `database/sql` 库就只是提供了一套操作 `db` 的接口和规范，例如抽象好的 `sql` 预处理 (`prepare`)，连接池管理，数据绑定，事务，错误处理等等。官方并没有提供具体某种数据库实现的协议支持。

和具体的数据库，例如 MySQL 打交道，还需要再引入 MySQL 的驱动，像下面这样：

```
import "database/sql"
import _ "github.com/go-sql-driver/mysql"

db, err := sql.Open("mysql", "user:password@/dbname")
```

```
import _ "github.com/go-sql-driver/mysql"
```

这一句 `import`，实际上是调用了 `mysql` 包的 `init` 函数，做的事情也很简单：

```
func init() {
    sql.Register("mysql", &MySQLDriver{})
}
```

在 `sql` 包的全局 `map` 里把 `mysql` 这个名字的 `driver` 注册上。实际上 `Driver` 在 `sql` 包中是一个 `interface`：

```
type Driver interface {
    Open(name string) (Conn, error)
}
```

调用 `sql.Open()` 返回的 `db` 对象实际上就是这里的 `Conn`。

```
type Conn interface {
    Prepare(query string) (Stmt, error)
    Close() error
    Begin() (Tx, error)
}
```

也是一个接口。实际上如果你仔细地查看 `database/sql/driver/driver.go` 的代码会发现，这个文件里所有的成员全都是 `interface`，对这些类型进行操作，实际上还是会调用具体的 `driver` 里的方法。

从用户的角度来讲，在使用 `database/sql` 包的过程中，你能够使用的也就是这些 `interface` 里提供的函数。来看一个使用 `database/sql` 和 `go-sql-driver/mysql` 的完整的例子：

```

package main

import (
    "database/sql"
    _ "github.com/go-sql-driver/mysql"
)

func main() {
    // db 是一个 sql.DB 类型的对象
    // 该对象线程安全，且内部已包含了一个连接池
    // 连接池的选项可以在 sql.Open 中设置，这里为了简单省略了
    db, err := sql.Open("mysql",
        "user:password@tcp(127.0.0.1:3306)/hello")
    if err != nil {
        log.Fatal(err)
    }
    defer db.Close()

    var (
        id int
        name string
    )
    rows, err := db.Query("select id, name from users where id = ?", 1)
    if err != nil {
        log.Fatal(err)
    }

    defer rows.Close()

    // 必须要把 rows 里的内容读完，否则连接永远不会释放
    for rows.Next() {
        err := rows.Scan(&id, &name)
        if err != nil {
            log.Fatal(err)
        }
        log.Println(id, name)
    }

    err = rows.Err()
    if err != nil {
        log.Fatal(err)
    }
}

```

如果读者想了解官方这个 database/sql 库更加详细的用法的话，可以参考：

<http://go-database-sql.org/>

包括该库的功能介绍、用法、注意事项和反直觉的一些实现方式(例如同一个 goroutine 内对 sql.DB 的查询，可能在多个连接上)都有涉及，本章中不再赘述。

聪明如你的话，在上面这段简短的程序中可能已经嗅出了一些不好的味道。官方的 db 库提供的功能这么简单，我们每次去数据库里读取内容岂不是都要去写这么一套差不多的代码？或者如果我们的对象是 struct，把 sql.Rows 绑定到对象的工作就会变得更加得重复而无聊。

是的，所以社区才会有各种各样的 sql builder 和 orm 百花齐放。

## 提高生产效率的 orm 和 sql builder

在 web 开发领域常常提到的 orm 是什么？我们先看看万能的维基百科：

对象关系映射（英语：Object Relational Mapping，简称ORM，或O/RM，或O/R mapping），是一种程序设计技术，用于实现面向对象编程语言里不同类型系统的数据之间的转换。从效果上说，它其实是创建了一个可在编程语言里使用的“虚拟对象数据库”。

最为常见的 orm 实际上做的是从 db -> 程序的 class / struct 这样的映射。所以你手边的程序可能是从 mysql 的表 -> 你的程序内 class。我们可以先来看看其它的程序语言里的 orm 写起来是怎么样的感觉：

```
>>> from blog.models import Blog
>>> b = Blog(name='Beatles Blog', tagline='All the latest Beatles news.')
>>> b.save()
```

完全没有数据库的痕迹，没错。orm 的目的就是屏蔽掉 db 层，实际上很多语言的 orm 只要把你的 class/struct 定义好，再用特定的语法将结构体之间的一对一或者一对多关系表达出来。那么任务就完成了。然后你就可以对这些映射好了数据库表的对象进行各种操作，例如 save，create，retrieve，delete。至于 orm 背着你背地里做了什么阴险的勾当，你是不一定清楚的。使用 orm 的时候，我们往往比较容易有一种忘记了数据库的直观感受。举个例子，我们有个一需求，是向用户展示最新的商品列表，我们再假设，我们的商品和商家是一对一的关联关系，我们就很容易写出像下面这样的代码：

```
# 伪代码
shopList := []
for product in productList {
    shopList = append(shopList, product.GetShop)
}
```

当然了，我们不能批判这样写代码的程序员是偷懒的程序员。因为 orm 一类的工具在出发点上就是屏蔽 sql，让我们对数据库的操作更接近于人类的思维方式。这样很多只接触过 orm 而且又是刚入行的程序员就很容易写出上面这样的代码。

这样的代码将对数据库的读请求放大了 N 倍。也就是说，如果你的商品列表有 15 个 sku，那么每次用户打开这个页面，至少需要执行 1(查询商品列表) + 15(查询相关的商铺信息) 次查询。这里 N 是 16。如果你的列表页很大，比如说有 600 个条目，那么你就至少要执行 1 + 600 次查询。如果说你的数据库能够承受的最大的简单查询是 12w qps，而上述这样的查询正好是你最常用的查询的话，实际上你能对外提供的服务能力是多少呢？是 200 qps！互联网系统的忌讳之一，就是这种无端的读放大。

当然，你也可以说这不是 orm 的问题，如果你手写 sql 你还是可能会写出差不多的程序，那么再来看两个 demo：

```
o := orm.NewOrm()
num, err := o.QueryTable("cardgroup").Filter("Cards__Card__Name", cardName).All(&cardgrou
```

很多 orm 都提供了这种 Filter 类型的查询方式，beego 也不例外。不过实际上在这段 orm 背后隐藏了非常难以察觉的细节，那就是生成的 sql 语句会自动 limit 1000。

也许喜欢 beego 的读者读到这里会反驳了，你是没有认真阅读 beego 的文档就瞎写。是的，尽管 beego 在文档里说明了 All 查询在不显式地指定 Limit 的话会自动 limit 1000，但对于很多没有阅读过文档或者看过 beego 源码的人，这依然是一个非常难以察觉的“魔鬼”细节。喜欢强类型语言的人一般都不喜欢语言隐式地去做什么事情，例如各种语言在赋值操作时进行的隐式类型转换然后又在转换中丢失了精度的勾当，一定让你非常的头疼。所以一个程序库背地里做的事情还是越少越好，如果一定要做，那也一定要在显眼的地方做。比如上面的例子，去掉这种默认的自作聪明的行为，或者要求用户强制传入 limit 参数都是更好的选择。

除了 limit 的问题，我们再看一遍这个 beego orm 的查询：

```
num, err := o.QueryTable("cardgroup").Filter("Cards__Card__Name", cardName).All(&cardgrou
```

你可以看得出来这个 Filter 是有表 join 的操作么？当然了，对 beego orm 有过深入使用经验的用户还是会觉得这是在吹毛求疵。但这样的分析想证明的是，orm 想从设计上隐去太多的细节。而方便的代价是其背后的运行完全失控。这样的项目在经过几任维护人员之后，将变得面目全非，难以维护。

当然，我们不能否认 orm 的进步意义，orm 的设计初衷是为了让数据的操作和存储的具体实现所剥离。但是上了规模的公司的人们渐渐达成了一个共识，由于隐藏重要的细节，orm 可能是失败的设计。其所隐藏的重要细节对于上了规模的系统开发来说至关重要。

相比 orm 来说，sql builer 在 sql 和项目可维护性之间取得了比较好的平衡。首先 sql builer 不像 orm 那样屏蔽了过多的细节，其次从开发的角度来讲，sql builder 简单进行封装后也可以非常高效地完成开发，举个例子：

```

where := map[string]interface{} {
    "order_id > ?" : 0,
    "customer_id != ?" : 0,
}
limit := []int{0,100}
orderBy := []string{"id asc", "create_time desc"}

orders := orderModel.GetList(where, limit, orderBy)

```

写 `sql builder` 的相关代码，或者读懂都不费劲。把这些代码脑内转换为 `sql` 也不会太费劲。所以通过代码就可以对这个查询是否命中数据库索引，是否走了覆盖索引，是否能够用上联合索引进行分析了。

说白了 `sql builder` 是 `sql` 在代码里的一种特殊方言，如果你们没有 `dba` 并且研发有自己分析和优化 `sql` 的能力，或者你们公司的 `dba` 对于学习这样一些 `sql` 的方言没有异议。那么使用 `sql builder` 是一个比较好的选择，不会导致什么问题。

另外在一些本来也不需要 `dba` 介入的场景内，使用 `sql builder` 也是可以的，例如你要做一套运维系统，且将 `mysql` 当作了系统中的一个组件，系统的 `qps` 不高，查询不复杂等等。

一旦你做的是高并发的 `OLTP` 在线系统，且想在人员充足分工明确的前提下最大程度控制系统的风险，使用 `sql builder` 就不合适了。

## 脆弱的 db

无论是 `orm` 还是 `sql builder` 都有一个致命的缺点，就是没有办法进行系统上线的事前 `sql` 审核。虽然很多 `orm` 和 `sql builder` 也提供了运行期打印 `sql` 的功能，但只在查询的时候才能进行输出。而 `sql builder` 和 `orm` 本身提供的功能太过灵活。使得你不可能通过测试枚举出所有可能在线上执行的 `sql`。例如你可能用 `sql builder` 写出下面这样的代码：

```

where := map[string]interface{} {
    "product_id = ?" : 10,
    "user_id = ?" : 1232 ,
}

if order_id != 0 {
    where["order_id = ?"] = order_id
}

res, err := historyModel.GetList(where, limit, orderBy)

```

你的系统里有类似上述样例的大量 `if` 的话，就难以通过 `test case` 来覆盖到所有可能的 `sql` 组合了。

这样的系统只要发布，就已经孕育了初期的巨大风险。

对于现在 **7 \* 24** 服务的互联网公司来说，服务不可用是非常重大的问题。存储层的技术栈虽经历了多年的发展，在整个系统中依然是最为脆弱的一环。系统宕机对于 **24** 小时对外提供服务的公司来说，意味着直接的经济损失。个中风险不可忽视。

从行业分工的角度来讲，现今的互联网公司都有专职的 **dba**。大多数 **dba** 并不一定有写代码的能力，去阅读 **sql builder** 的相关“拼 **sql**”代码多多少少还是会有一点障碍。从 **dba** 角度出发，还是希望能够有专门的事前 **sql** 审核机制，并能让其低成本地获取到系统的所有 **sql** 内容，而不是去阅读业务研发编写的 **sql builder** 的相关代码。

所以现如今，大型的互联网公司核心线上业务都会在代码中把 **sql** 放在显眼的位置提供给 **dba review**，以此来控制系统在数据层的风险。结合 **golang** 举一个例子：

```
const (
    getAllByProductIDAndCustomerID = `select * from p_orders where product_id in (:product_ids) and customer_id = :customer_id`
)

// GetAllByProductIDAndCustomerID
// @param driver_id
// @param rate_date
// @return []Order, error
func GetAllByProductIDsAndCustomerID(ctx context.Context, productIDs []uint64, customerID uint64) ([]Order, error) {
    var orderList []Order

    params := map[string]interface{}{
        "product_id": productIDs,
        "customer_id": customerID,
    }

    // getAllByProductIDsAndCustomerID 是 const 类型的 sql 字符串
    sql, args, err := sqlutil.Named(getAllByProductIDsAndCustomerID, params)
    if err != nil {
        return nil, err
    }

    err = dao.QueryList(ctx, sqldbInstance, sql, args, &orderList)
    if err != nil {
        return nil, err
    }

    return orderList, err
}
```

像这样的代码，在上线之前把 **dao** 层的变更集的 **const** 部分直接拿给 **dba** 来进行审核，就比较方便了。代码中的 **sqlutil.Named** 是类似于 **sqlx** 中的 **Named** 函数，同时支持 **where** 表达式中的比较操作符和 **in**。

这里为了说明简便，函数写得稍微复杂一些，仔细思考一下的话查询的导出函数还可以进一步进行简化。请读者朋友们自行尝试。

## 附录

附录部分主要包含量三个部分：第一部分是摘录量一些Go语言常见的坑和解决方案；第二部分是参考网站和参考数目；第三部分是作者信息。

## 附录A：Go语言常见坑

这里列举的Go语言常见坑都是符合Go语言语法的，可以正常的编译，但是可能是运行结果错误，或者是有资源泄漏的风险。

### 数组是值传递

在函数调用参数中，数组是值传递，无法通过修改数组类型的参数返回结果。

```
func main() {
    x := [3]int{1, 2, 3}

    func(arr [3]int) {
        arr[0] = 7
        fmt.Println(arr)
    }(x)

    fmt.Println(x)
}
```

必要时需要使用切片。

### map遍历是顺序不固定

map是一种hash表实现，每次遍历的顺序都可能不一样。

```
func main() {
    m := map[string]string{
        "1": "1",
        "2": "2",
        "3": "3",
    }

    for k, v := range m {
        println(k, v)
    }
}
```

### 返回值被屏蔽

在局部作用域中，命名的返回值内同名的局部变量屏蔽：

```
func Foo() (err error) {
    if err := Bar(); err != nil {
        return
    }
    return
}
```

## recover必须在defer函数中运行

recover捕获的是祖父级调用时的异常，直接调用时无效：

```
func main() {
    recover()
    panic(1)
}
```

直接defer调用也是无效：

```
func main() {
    defer recover()
    panic(1)
}
```

defer调用时多层嵌套依然无效：

```
func main() {
    defer func() {
        func() { recover() }()
    }()
    panic(1)
}
```

必须在defer函数中直接调用才有效：

```
func main() {
    defer func() {
        recover()
    }()
    panic(1)
}
```

## main函数提前退出

后台 Goroutine 无法保证完成任务。

```
func main() {
    go println("hello")
}
```

## 通过 Sleep 来回避并发中的问题

休眠并不能保证输出完整的字符串：

```
func main() {
    go println("hello")
    time.Sleep(time.Second)
}
```

类似的还有通过插入调度语句：

```
func main() {
    go println("hello")
    runtime.Gosched()
}
```

## 独占 CPU 导致其它 Goroutine 饿死

Goroutine 是协作式调度，Goroutine 本身不会主动放弃 CPU：

```
func main() {
    runtime.GOMAXPROCS(1)

    go func() {
        for i := 0; i < 10; i++ {
            fmt.Println(i)
        }
    }()

    for {} // 占用CPU
}
```

解决的方法是在 for 循环加入 runtime.Gosched() 调度函数：

```

func main() {
    runtime.GOMAXPROCS(1)

    go func() {
        for i := 0; i < 10; i++ {
            fmt.Println(i)
        }
    }()

    for {
        runtime.Gosched()
    }
}

```

或者是通过阻塞的方式避免CPU占用:

```

func main() {
    runtime.GOMAXPROCS(1)

    go func() {
        for i := 0; i < 10; i++ {
            fmt.Println(i)
        }
    }()

    select{}
}

```

## 不同**Goroutine**之间不满足顺序一致性内存模型

因为在不同的Goroutine, main函数可能无法观测到done的状态变化, 那么for循环会陷入死循环:

```

var msg string
var done bool = false

func main() {
    runtime.GOMAXPROCS(1)

    go func() {
        msg = "hello, world"
        done = true
    }()

    for {
        if done {
            println(msg)
            break
        }
    }
}

```

解决的办法是用显示同步:

```

var msg string
var done = make(chan bool)

func main() {
    runtime.GOMAXPROCS(1)

    go func() {
        msg = "hello, world"
        done <- true
    }()

    <-done
    println(msg)
}

```

## 闭包错误引用同一个变量

```

func main() {
    for i := 0; i < 5; i++ {
        defer func() {
            println(i)
        }()
    }
}

```

改进的方法是在每轮迭代中生成一个局部变量

```
func main() {
    for i := 0; i < 5; i++ {
        i := i
        defer func() {
            println(i)
        }()
    }
}
```

或者是通过函数参数传入

```
func main() {
    for i := 0; i < 5; i++ {
        defer func(i int) {
            println(i)
        }(i)
    }
}
```

## 在循环内部执行**defer**语句

**defer**在函数退出时才能执行, 在**for**执行**defer**会导致资源延迟释放:

```
func main() {
    for i := 0; i < 5; i++ {
        f, err := os.Open("/path/to/file")
        if err != nil {
            log.Fatal(err)
        }
        defer f.Close()
    }
}
```

解决的方法可以在**for**中构造一个局部函数, 在局部函数内部执行**defer**:

```

func main() {
    for i := 0; i < 5; i++ {
        func() {
            f, err := os.Open("/path/to/file")
            if err != nil {
                log.Fatal(err)
            }
            defer f.Close()
        }()
    }
}

```

## 切片会导致整个底层数组被锁定

切片会导致整个底层数组被锁定，底层数组无法释放内存。如果底层数组较大会对内存产生很大的压力。

```

func main() {
    headerMap := make(map[string][]byte)

    for i := 0; i < 5; i++ {
        name := "/path/to/file"
        data, err := ioutil.ReadFile(name)
        if err != nil {
            log.Fatal(err)
        }
        headerMap[name] = data[:1]
    }

    // do some thing
}

```

解决的方法是将结果克隆一份，这样可以释放底层的数组：

```

func main() {
    headerMap := make(map[string][]byte)

    for i := 0; i < 5; i++ {
        name := "/path/to/file"
        data, err := ioutil.ReadFile(name)
        if err != nil {
            log.Fatal(err)
        }
        headerMap[name] = append([]byte{}, data[:1]...)
    }

    // do some thing
}

```

## 空指针和空接口不等价

比如返回了一个错误指针, 但是并不是空的error接口:

```

func returnsError() error {
    var p *MyError = nil
    if bad() {
        p = ErrBad
    }
    return p // Will always return a non-nil error.
}

```

## 内存地址会变化

Go语言中对象的地址可能发生变化, 因此指针不能从其它非指针类型的值生成:

```

func main() {
    var x int = 42
    var p uintptr = unsafe.Pointer(&x)

    runtime.GC()
    var px *int = (*int)(unsafe.Pointer(p))
    println(*px)
}

```

当内存发送变化的时候, 相关的指针会同步更新, 但是非指针类型的uintptr不会做同步更新.

同理, cgo中也不能保存Go对象地址.

# Goroutine泄露

Go语言是带内存自动回收的特性，因此内存一般不会泄漏。但是Goroutine确存在泄漏的情况，同时泄漏的Goroutine引用的内存同样无法被回收。

```
func main() {
    ch := func() <-chan int {
        ch := make(chan int)
        go func() {
            for i := 0; ; i++ {
                ch <- i
            }
        }()
        return ch
    }()
}

for v := range ch {
    fmt.Println(v)
    if v == 5 {
        break
    }
}
}
```

上面的程序中后台Goroutine向管道输入自然数序列，main函数中输出序列。但是当break跳出for循环的时候，后台Goroutine就处于无法被回收的状态了。

我们可以通过context包来避免做个问题：

```
func main() {
    ctx, cancel := context.WithCancel(context.Background())

    ch := func(ctx context.Context) <-chan int {
        ch := make(chan int)
        go func() {
            for i := 0; ; i++ {
                select {
                case <- ctx.Done():
                    return
                case ch <- i:
                }
            }
        }()
        return ch
    }(ctx)

    for v := range ch {
        fmt.Println(v)
        if v == 5 {
            cancel()
            break
        }
    }
}
```

当main函数在break跳出循环时，通过调用cancel()来通知后台Goroutine退出，这样就避免了Goroutine的泄漏。

## 附录B：参考资料

### 参考网站

- Go语言官网: <https://golang.org>
- SWIG官网: <http://swig.org>
- GopherJS官网: <http://www.gopherjs.org>
- GRPC官网: <http://www.grpc.io>
- rsc博客: <http://research.swtch.com>

### 参考书目

- 《Go语言圣经》: <https://gopl.io>
- 《Go语言圣经(中文版)》: <https://github.com/golang-china/gopl-zh>
- 《Go语言·云动力》: <http://www.ituring.com.cn/book/1040>
- 《Go语言编程》: <http://www.ituring.com.cn/book/967>
- 《Go语言程序设计》: <http://www.ptpress.com.cn/Book.aspx?id=35714>
- 《C程序设计语言》: <http://product.china-pub.com/14975>
- 《汇编语言:基于X86处理器》: <http://product.china-pub.com/4934543>
- 《现代x86汇编语言程序设计》: <http://product.china-pub.com/5006762>
- 《深入理解程序设计：使用Linux汇编语言》: <http://product.china-pub.com/3768972>
- 《代码的未来》: <http://product.china-pub.com/3767536>

## 附录C：作者简介

- 柴树杉（网络ID：[chai2010](#)）是国内第一批Go语言爱好者，创建了最早的QQ讨论组和[golang-china](#)邮件列表，组织[Go语言官方文档](#)和[《Go语言圣经》](#)的翻译工作，**Go语言**代码的贡献者，并开源了诸多[Go语言相关的资源](#)。