1. 关于 Go 语言的介绍

1.1 用 Go 解决现代编程难题

当今的软件开发，开发人员不得不在快速开发和性能之间做出抉择。C 和 C++ 类的语言提供了很快的执行速度，而 Ruby 和 Python 则更擅长于快速开发。Go 语言在这两者之间架起了桥梁，不仅提供了高性能的语言，同时也让开发更快速。

具体体现在以下几个小节中。

1.1.1 开发速度

编译一个大型的 C 或 C++ 项目所花费的时间往往很长，而 Go 语言使用了更加智能的编译器，并简化了解决依赖的算法，最终提供了更快的编译速度。（编译 Go 程序时，编译器只会关注那些直接引用的库，而不是像 Java、C 和 C++ 那样，要遍历依赖链中所有依赖的库）

因为没有从编译代码到执行代码的中间过程，用动态语言编写的应用程序可以快速的看到输出。但这方面的代价是，动态语言并不提供静态语言的类型安全检查，无法避免在运行时出现的类型错误。而 Go 语言的编译器能够帮用户捕获这个类型错误。

1.1.2 并发

现代计算机都拥有多个核，但是大部分编程语言都没有有效的工具让程序可以轻易的利用这些资源，这些语言需要写大量的线程同步代码来利用多个核，很容易导致错误。

Go 语言对并发的支持是这门语言最重要的特性之一。

- goroutine 很像线程，但它占用的内存远少于线程，使用它需要的代码更少。
- 通道（channel）是一种内置的数据结构，可以让用户在不同的 goroutine 之间同步发送具有类型的消息。这种编程模型更倾向于在 goroutine 之间发送消息，而不是让多个 goroutine 争夺同一数据的使用权。

1. goroutine

goroutine 是可以与其他 goroutine 并发执行的函数，同时也会与主程序并行执行。在其他语言中，你需要使用线程来完成同样的事情，而在 Go 语言中会使用同一个线程来执行多个 goroutine。

goroutine 使用的内存比线程更少，Go 语言在运行时会自动在配置的一组逻辑处理器上调度执行 goroutine，每个逻辑处理器绑定到一个操作系统的线程上，这让用户的应用程序执行效率更高，而并发工作量显著减少。

```
package main

import (
    "fmt"
    "runtime"
    "log"
    "os"
    "sync"
)

var wg sync.WaitGroup

func writeLog(msg string) {
    defer wg.Done()

    fileName := "test.log"
    logFile, err := os.Create(fileName)
    defer logFile.Close()
    if err != nil {
        log.Fatalln("open file error")
    }
    debugLog := log.New(logFile, "[Debug]", log.LstdFlags)
    debugLog.Printf("a debug message: %s", msg)

    debugLog.SetPrefix("[Info]")
    debugLog.Printf("a info message: %s", msg)

    debugLog.SetFlags(debugLog.Flags() | log.LstdFlags)
    debugLog.Printf("a different prefix: %s", msg)
}

func main()  {
    runtime.GOMAXPROCS(1)
    wg.Add(1)

    fmt.Println("Start")
    go writeLog("this is test")
    fmt.Println("waiting goroutine finish")
    wg.Wait()
    fmt.Println("End")
}
```

2 通道

通道是一种数据结构，可以让 goroutine 之间进行安全的数据通信。通道可以帮用户避免其他语言里常见的共享内存访问的问题。

并发的难点就在于要确保其他并发运行的进程、线程或 goroutine 不会意外修改用户的数据。当不同的线程在没有同步保护的情况下修改同一个数据时，总会发生灾难。在其他语言中，如果使用全局变量或这共享内存，必须使用复杂的锁规则来防止对同一变量的不同步修改。

1.1.3 Go 语言的类型系统

Go 语言提供了灵活的、无继承的类型系统，无需降低运行性能就能最大程度上复用代码。

这个类型系统依旧支持面向对象开发，但避免了传统面向对象的问题（需要考虑抽象类和接口的设计）。Go 开发者使用组合涉及模式，只需要简单地将一个类型嵌入另一个类型中，就能复用所有所有的功能。

1. 类型简单

Go 语言不仅有类似 `int` 和 `string` 这样的内置类型，还支持用户定义的类型。

2. Go 接口对一组行为建模

接口用于描述类型的行为。

在 Go 语言中，只需要实现接口的所有方法，那么这个类型的实例就是接口类型的实例，不需要额外的声明。而在 Java 这种严格的面向对象的语言中，类必须实现接口中所有的方法并且需要显示的声明接口，才能成为这个接口的实例。

Go 语言接口更小，只倾向与定义一个单一的动作，这样有利于使用组合来复用代码。

1.1.4 内存管理

不当的内存管理会导致程序崩溃或内存泄漏，甚至让整个操作系统崩溃。Go 语言拥有现代化的垃圾回收机制，能帮你解决这个问题。

在 C 和 C++ 中，使用内存要先分配内存，使用完毕后需要将其释放，尤其在并发模式下，追踪内存的使用情况变得尤为困难。而 Go 语言把内存管理交给专业的编译器去做，而让程序员专注于更有趣的事情。

1.2 你好，Go

```
/** main 包会被编译成可执行文件
  * 如果 main 函数不在 main 包中，编译器就不会生成可执行文件
  */
package main

// 导入包
import "fmt"

// init 函数在 main 函数之前被调用
func init() {
    fmt.Println("func init run before main")
}

// 程序的主入口
func main()  {
    fmt.Println("Hello, Golang")
}
```

3. 包

3.1 包

所有 Go 语言的程序都会组织成若干组文件，每组文件被称为一个包。这样每个包的代码都可以作为很小的复用单元，被其他项目引用。

所有的 .go 文件，除了空行和注释，都应该在第一行声明自己所属的包。

同一目录下的所有 .go 文件必须声明为同一个包。不同包需放在不同的目录下。

3.1.1 包名惯例

- 包名应该使用目录名
- 包名需使用简洁、清晰且全小写的名字
- 并不需要所有包的名字都与别的包名不同，因为导入包时使用的是全路径，所以可区分同名的不同包
- 包导入后会使用默认的包名，但也可修改导入后的包名

3.1.2 main 包

在 Go 语言里，命名为 `main` 的包具有特殊的含义。Go 语言的编译承训会试图把这种名字为 `main` 的包编译为可执行文件。所有用 Go 语言编译的可执行程序都必须有一个名叫 `main` 的包。

如果 `main` 包里没有定义 `main()` 函数，那么程序就无法运行，也无法编译成可执行程序。

3.2 导入

import 语句会告诉编译器到磁盘的哪里去找想要导入的包。如果需要导入多个包，习惯上是将 import 语句包装在一个导入块中。

```
import (
  "fmt"
  "strings"
)
```

编译器会使用 Go 环境变量设置的路径，通过引入的相对路径来查找磁盘上的包。

- 标准库中的包会在安装 Go 的位置找到
- Go 开发者创建的包会在 `GOPATH` 环境变量指定的目录里查找

需要注意的是，一旦编译器找到了一个满足 `import` 语句指定的包，就会停止进一步查找。编译器首先会查找 Go 的安装目录，再依次查找 `GOPATH` 变量列出的目录。

如果没有找到需要的包，那么试图对程序进行 `run` 或 `build` 时，编译器会报错。后面可通过 `go get` 命令来修正这种错误。

3.2.1 远程导入

```
import "github.com/spf13/viper"
```

用导入路径编译程序时，`go build` 命令会使用 `GOPATH` 的设置，在磁盘上搜索这个包。如果路径包含 URL，可以使用 Go 工具链从 DVCS 获取包，并把包的源代码保存在 `GOPATH` 指向的路径里与 URL 匹配的目录里，这个过程使用 `go get` 命令完成。

`go get` 将获取任意指定的 URL 的包，或者一个已经导入的包所依赖的其他包。由于 `go get` 这种递归特性，这个命令会扫描某个包的源码树，获取所有能找到的依赖包。

3.2.2 命名导入

命名导入是解决包名冲突的有效方法，同时也可简化包名。

Go 语言导入包有以下几种方式：

- ```
  // 使用 test.Test()
  import "test/test"
  ```
- ```
  // 使用 mongo.Test()
  import mongo "mongo/db"
  ```
- ```
  // 对包的调用省略包名，可以直接调用包中的方法或变量
  import . "test/test"
  ```
- ```
  // 当需要导入一个包，但是不需要引用这个包的标识符，使用空白标识符来替代
  import _ "test/test"
  ```

3.3 函数 init

每个包可以包含任意多个 `init` 函数，这些函数都会在程序执行开始的时候被调用。所有被编译器发现的 init 函数都会被安排在 main 函数之前执行。

init 函数在设置包、初始化变量或者其他要在程序运行之前优先完成的引导工作。

```
package main

import (
    "fmt"
    "os"
    "./test"
)

func init()  {
    fmt.Println("run before main")
}

func main() {
    fmt.Println(os.Getenv("GOPATH"))
    fmt.Println(test.Test())
}
```

3.4 使用 Go 工具

`go build`：编译程序，生成可执行文件。

- `go build hello.go`：指定 Go 文件
- `go build`：使用当前目录来编译
- `go build test/...`：编译 test 目录下的所有包

`go clean`：清除编译生成的可执行文件。

`go run hello.go`：编译完后需要生成运行可执行文件，使用 `go run` 命令可以将 `go build` 和运行这两步合二为一。

3.5 进一步介绍 Go 开发工具

3.5.1 go vet

`go vet main.go` 命令会帮开发人员检测代码的常见错误

- Printf 类函数调用时，类型匹配错误的参数
- 定义常用的方法时，方法签名的错误
- 错误的结构标签
- 没有指定字段名的结构字面量

3.5.2 Go 代码格式化

`go fmt main.go` 用来格式化代码，可以用来在保存文件或提交代码库前执行 `go fmt`

3.5.3 Go 语言文档

`go doc tar`：命令行查看 tar 包文档

`godoc -http:=6060`：在线查看文档

3.7 依赖管理

gb 是一个由 Go 社区成员开发的全新构建工具，初步理解与 `composer` 类似

4 数组、切片和映射

4.1 数组的内部实现和基础功能

4.1.1 内部实现

在 Go 语言里，数据是一个长度固定的数据类型，用于存储一段具有相同的类型的元素的连续块。

数据存储的类型可以是内置类型，如 int 和 string 等，也可以是自定义类型。

由于数组的每个元素类型相同，又是连续分配，所以可以以固定速度索引数组中的任意数据，速度非常快。

4.1.2 声明和初始化

- 声明一个数组并初始化为零值

```
// 声明一个包含 5 个元素的整形数组
var array [5]int
```

- 使用数组字面量声明数组

```
// 声明一个包含5个元素的整形数组并使用具体值初始化每个元素
array := [5]int{10, 20, 30, 40, 50}
// `...` 表示容量由初始化值的数量决定
array := [...]int{10, 20, 30, 40}
```

- 声明数据并指定特定元素的值

```
// 声明一个有 5 个元素的数组，用具体值初始化索引为 1 和 2 的元素，其余元素保存零值
array := [5]int{1: 10, 2:20}
```

4.1.3 使用数组

- 访问数组元素

```
array := [5]int{10, 20, 30, 40, 50}
// 修改数组中索引为 2 的元素的值
array[2] = 35
```

- 访问指针数组的元素

```
// 声明包含 5 个元素的指向整形的数组
array := [5]*int{0: new(int), 1: new(int)}
// 为索引 0 和 1 的元素赋值
*array[0] = 10
*array[1] = 20
```

- 把一个指针数组赋值给另一个

```
var array1 [3]*string
array2 := [3]*string{new(string), new(string), new(string)}
*array2[0] = "a"
*array2[1] = "b"
*array3[2] = "c"
array1 = array2
```

> 数组变量的类型包括长度和每个元素的类型，这有这两部分都相同的数组，才能互相赋值，否则编译器会报错

4.1.4 多维数组

- 声明二维数组

```
// 声明一个二维数组，两个维度分别为存储 4 个元素和 2 个元素
var array [4][2]int
// 使用数组字面量来声明并初始化一个二维整形数组
array := [4][2]int{{10, 11}, {12, 13}, {14, 15}, {16, 17}}
// 声明并初始化外层数组中索引为 1 和 3 的元素
array := [4][2]int{1: {20, 21}, 3: {30, 31}}
// 声明并初始化外层数组和内层数组的单个元素
array := [4][2]int{1: (0: 20), 3: (1: 31)}
```

- 访问二维数组

```
var array [2][2]int
array[0][0] = 10
array[0][1] = 20
array[1][0] = 30
arrat[1][1] = 40
```

4.1.5 在函数之间传递数组

在函数之间传递数组是一个开销很大的操作。因为在函数之间传递数组时，总是以值的方式传递的。

- 使用指针在函数间传递大数组

```
var array [1e6]int
foo(&array)
func foo(array *[1e6]int) {
  // ....
}
```

> 虽然将数组的地址传给函数，不会带来很大的内存开销，但是由于传递的是指针，如果改变指针指向的值，会改变共享的内存。

4.2 切片的内部实现和基础功能

4.2.1 内部实现

切片是一种便于使用和管理的数据集合，其是围绕动态数组的概念构建的，可以按需自动增长和缩小。

切片是一个很小的对象，对底层数组进行了抽象，并提供相关的操作方法。

切片有 3 个字段的数据节后，这些数据结构包含 Go 语言需要操作底层数组的元数据。

这 3 个字段分别是指向底层数组的指针、切片的长度、切片的容量。

4.2.2 创建和初始化

1. make 和 切片字面量

- 使用长度声明一个字符串切片

```
// 创建一个字符串切片
// 其长度和容量都是 5 个元素
slice := make([]string, 5)
```

> 如果只指定长度，那么容量和长度相等

- 使用长度和容量声明整形切片

```
// 创建一个整形切片
// 其长度为 4，容量为 7
slice := make([]int, 4, 7)
```

> 不允许创建容量小于长度的切片

- 通过字面量来声明切片

```
slice := []string{"red", "blue", "yellow"}
slice := []int{10, 20, 30}
```

- 使用切片字面量，可以设置初始化长度和容量

```
// 创建字符串切片
// 使用空字符串初始化第 100 个元素
slice := []string{99: ""}
```

- 声明数组和切片的不同

```
// 声明数组
array := [3]int{10, 20, 30}
// 声明切片
slice := []int{10, 20, 30}
```

如果在 `[]` 运算符里指定了一个值，那么创建的就是数组而不是切片

1. nil 和空切片

- 创建 nil 切片

```
// 创建 nil 整形切片
var slice []int
```

需要描述一个不存在的切片时，nil 切片会很好用

- 创建空切片

```
slice := make([]int, 0)
slice := []int{}
```

在表示空集合时，空切片很有用

4.2.3 使用切片

4.2.3.1 赋值和切片

对切片元素赋值与数组元素的赋值一样

- 使用切片字面量来声明切片

```
slice := []int{10, 20, 30}
slice[1] = 21
```

- 使用切片创建切片

```
slice := []int{10, 20, 30}
newSlice := slice[1:2]
```

上面两个切片共享一段底层数组，但通过不同的切片会看到底层数组的不同部分

- 计算长度和容量

> 对于底层数组容量是 k 的切片 slice[i:j] 来说
>
> 长度：j - i
>
> 容量：k - i

- 修改切片内容可能导致的结果

```
slice := []int{10, 20, 30}
newSlice := slice[1:2]
newSlice[0] = 21
// output: 21
fmt.Pringln(slice[1])
```

4.2.3.2 切片增长

相对数组而言，使用切片的一个好处是，可以按需增长切片的容量。

Go 语言内置的 append 函数会处理增加长度时的所有操作细节。append 调用返回时，会返回一个包含修改结果的新切片。

函数 append 总是会增加新切片的长度，而容量有可能会改变，也可能不会改变，这取决与被操作的切片是否有可用的容量。

- 使用 append 向切片增加元素

```
slice := []int{10, 20, 30}
newSlice := slice[0 : 1]
newSlice = append(newSlice, 21)
// [10 21]
fmt.Println(newSlice)
// [10 21 30]
fmt.Println(slice)
```

> 因为 newSlice 在底层数组还有容量可以，所以和 slice 共用的是同一个底层数组，对 newSlice 的修改就是修改共用的底层数组

- 使用 append 同时增加长度和容量

```
slice := []int{10, 20, 30, 40}
newSlice := append(slice, 50)
newSlice[0] = 11
// [10 20 30 40]
fmt.Println(slice)
// [11 20 30 40 50]
fmt.Println(newSlice)
```

> 如果切片的底层数组没有足够的可以容量，append 函数会创建一个新的底层数组，将被引用的现有的值复制到新数组里，再追加新的值。
>
> 在上面的例子中，newSlice 与 slice 不再共用同一个底层数组。

> append 函数会智能的处理底层数组的容量增长：当切片的容量小于 1000 个元素时，总是成倍的增长容量。一旦元素个数超过 1000，容量的增长因子会设为 1.25，也就是每次增长 25% 的容量。

4.2.3.3 创建切片的 3 个索引

在创建切片时，还可以使用第三个索引选项，第三个索引选项可以用来控制新切片的容量，其目的并不是要增加容量，而是要限制容量，这为底层数组提供了一定的保护，可以更好的控制追加操作。

- 使用 3 个索引创建切片

```
source := []string{"apple", "orange", "banana", "grape", "plum"}
slice := source[2 : 3 : 4]
```

> 对于 slice[i : j : k]，其长度为 j - i，容量为 k - i。
>
> 当设置容量大于已有容量时编译器会报错

- 设置长度和容量一致

```
source := []string{"apple", "orange", "banana", "grape", "plum"}
slice := source[1:3:3]
slice = append(slice, "kiwi", "hello")
// [orange banana kiwi hello]
fmt.Println(slice)
```

> 如果在创建切片时设置切片的容量和长度一样，就可以强制让新切片的第一个 append 函数操作创建新的底层数组，与原有的底层数组分离。

- 将一个切片追加到另一个切片

```
s1 := []int{1, 2}
s2 := []int{2, 4}
// [1 2 2 4]
fmt.Printf("%v\n", append(s1, s2...))
```

4.2.3.4 迭代切片

Go 语言里可以使用关键字 range，它可以配合关键字 for 来迭代切片里的元素

- 使用 range 迭代切片

```
slice := []int{10, 20, 30}
for _, value := range slice {
  fmt.Printf("value: %d\n", value)
}
for index, value := range slice {
  fmt.Printf("index: %d, value: %d\n", index, value)
}
```

- 使用传统的 for 循环迭代切片

```
slice := []int{10, 20}
for index := 2; index < len(slice); index++ {
  fmt.Printf("index: %d, value: %d\n", index, slice[index])
}
```

内置函数 len 和 cap，可以用于处理数组、切片和通道。对于切片，len 返回长度，cap 返回容量

4.2.4 多维切片

- 声明多维切片

```
slice := [][]int{{10}, {20, 30}}
slice[0] = append(slice[0], 20)
// [[10 20] [20 30]]
fmt.Println(slice)
```

4.2.5 在函数间传递切片

由于与切片关联的数据包含在底层数组里，不属于切片本身，所以将切片复制到任意函数的时候，对底层数组大小都不会有影响。复制时只会赋值切片本身，所以在函数间传递会非常快速、简单。

```
package main

import (
    "fmt"
)

func test(slice []int)  {
    var newSlice []int
    newSlice = slice
    newSlice = slice[0:len(slice):len(slice)]
    newSlice = append(newSlice, 40)
    newSlice[0] = 10
    fmt.Println(newSlice)
}

func main() {
    slice := []int{1, 2, 3, 4, 5}
    test(slice[0:3])
    fmt.Println(slice)
    // [10 2 3 40]
    // [1 2 3 4 5]
}
```

4.3 映射的内部实现和基础功能

映射是一种存储一系列无序键值对的数据结构

4.3.1 内部实现

只需记住一件事：映射是一个存储键值对的无序集合

4.3.2 创建和初始化

- 使用 make 声明映射

```
dict := make(map[string]int)
dict := map[string]string{"name": "li", "gender": "male"}
```

- 声明一个存储字符串切片的映射

```
// 创建一个映射，使用字符串切片作为值
dict := map[int][]string{}
```

4.3.3 使用映射

```
func main() {
    // 创建一个空映射并赋值
    colors := map[string]string{}
    colors["red"] = "RED"
    fmt.Println(colors)
    // 对 nil 映射赋值时，会报错
    var colors2 map[string]string
    colors2["red"] = "RED"
    fmt.Println(colors2)
}
```

从映射取值时有两个选择。

- 第一个选择是，可以同时获得值，以及一个表示这个是否存在的标志

```
value, exists := colors["blue"]
// 这个键存在吗
if exists {
  fmt.Println(value)
}
```

- 另一个选择是，只返回键对应的值，然后判断这个值是否是零值来判断键是否存在

```
value := colors["blue"]
if value != "" {
  fmt.Println(value)
}
```

使用 range 关键字迭代映射

```
func main() {
    colors := map[string]string{"name": "lilei", "gender": "male", "country": "china"}
    for key, value := range colors {
        fmt.Printf("%s is %s\n", key, value)
    }
}
```

从映射中删除一项

```
func main() {
    colors := map[string]string{"name": "liLei", "gender": "male", "country": "china"}
    delete(colors, "name")
    for key, value := range colors {
        fmt.Printf("%s is %s\n", key, value)
    }
}
```

4.3.4 在函数间传递映射

将映射传递给函数成本很小，并且不会赋值底层的数据结构

```
package main

import (
    "fmt"
)

func removeColors(colors map[string]string, key string) map[string]string {
    delete(colors, key)
    return colors
}

func iterator(colors map[string]string)  {
    for key, value := range colors {
        fmt.Printf("%s is %s\n", key, value)
    }
}

func main() {
    colors := map[string]string{"name": "liLei", "gender": "male", "country": "china"}
    iterator(removeColors(colors, "name"))
    iterator(colors)
    // output is:
    // gender is male
    // country is china
    // gender is male
    // country is china
}
```

Go 语言是一种静态类型的编程语言，这意味着编译器在编译时需要事先知道变量的类型，这样有助于编译器对代码的一些优化，提高执行效率。

5 Go 语言的类型

5.1 用户定义的类型

Go 语言里声明用户自定义类型有两种方式：

- 使用关键字 struct 创建结构类型
- 基于一个已有类型，将其作为新类型的类型说明

5.1.1 使用 struct 关键字声明结构类型

声明一个结构类型

```
type user struct {
  name string
  email string
  age int
  isMale bool
}
```

使用结构类型声明变量并初始化零值

```
var bill user
```

> 任何时候，创建一个变量并初始化为零值，习惯上是使用关键字 var。如果变量被初始化为非零值，就配合结构字面量和短变量声明操作符来创建变量(即 :=)

使用结构字面量来声明一个结构类型的变量

```
lisa := user{
  name: "lisa",
  email: "lisa@email.com"
  age: 22,
  isMale: false,
}
```

不使用字段名，创建结构类型的值

```
// 这种形式下，值的顺序很重要，必须要和结构声明中的顺序一致，另外最后也不需要使用 ',' 结尾
// 可以部分初始化，未初始化的部分为零值
lisa := {"lias", "lisa@eamil.com", 22, false}
```

使用其他结构类型声明字段

```
type admin struct {
  person user
  level string
}
```

使用结构字面量来创建字段的值

```
fred := admin {
  person: user {
    name: "fred",
    email: "fred@email.com"
    age: 22,
    isMale: true,
  },
  level: "super",
}
```

5.1.2 基于一个已有类型，将其作为新类型的说明

当需要一个可以用已有类型表示的新类型的时候，这种方法会很好用。

基于 int64 声明一个新类型

```
type Duration int64
```

5.2 方法

方法能给用户定义的类型添加新的行为。

方法实际上也是函数，只是在声明时，在关键字 func 和方法名之间增加了一个参数。

```go
// Sample program to show how to declare methods and how the Go
// compiler supports them.
package main

import (
    "fmt"
)

// user defines a user in the program.
type user struct {
    name  string
    email string
}

// notify implements a method with a value receiver.
func (u user) notify() {
    fmt.Printf("Sending User Email To %s<%s>\n",
        u.name,
        u.email)
}

// changeEmail implements a method with a pointer receiver.
func (u *user) changeEmail(email string) {
    u.email = email
}

// main is the entry point for the application.
func main() {
    // Values of type user can be used to call methods
    // declared with a value receiver.
    bill := user{"Bill", "bill@email.com"}
    bill.notify()

    // Pointers of type user can also be used to call methods
    // declared with a value receiver.
    lisa := &user{"Lisa", "lisa@email.com"}
    lisa.notify()

    // Values of type user can be used to call methods
    // declared with a pointer receiver.
    bill.changeEmail("bill@newdomain.com")
    bill.notify()

    // Pointers of type user can be used to call methods
    // declared with a pointer receiver.
    lisa.changeEmail("lisa@newdomain.com")
    lisa.notify()
}
```

5.3 类型的本质

类型的本质主要体现在，如果给这个类型增加或删除某个值，是创建一个新值，还是在更改当前值。这也关系到这个类型在函数之间到底时用值传递还是指针传递。

5.3.1 内置类型

内置类型时语言提供的一组类型，比如数值类型、字符串类型、布尔类型和数组等。

这些类型本质上是原始类型，在函数或方法间传递时，传递的是对应值的副本

5.3.2 引用类型

Go 语言里的引用类型有如下几个：切片、映射、通道、接口和函数类型。

这些类型在函数间传递的是指针，共享底层数据结构。

5.3.3 结构类型

结构类型可以用来描述一组数据值，这组值的本质既可以是原始的，也可以是非原始的。

5.4 接口

多态是指代码可以根据类型的具体实现采取不同的行为。

如果一个类型实现了某个接口，所有使用这个接口的地方，都可以支持这种类型的值。

5.4.1 标准库

```go
// Sample program to show how to write a simple version of curl using
// the io.Reader and io.Writer interface support.
package main

import (
    "fmt"
    "io"
    "net/http"
    "os"
)

// init is called before main.
func init() {
    if len(os.Args) != 2 {
        fmt.Println("Usage: ./example2 <url>")
        os.Exit(-1)
    }
}

// main is the entry point for the application.
func main() {
    // Get a response from the web server.
    r, err := http.Get(os.Args[1])
    if err != nil {
        fmt.Println(err)
        return
    }

    // Copies from the Body to Stdout.
    io.Copy(os.Stdout, r.Body)
    if err := r.Body.Close(); err != nil {
        fmt.Println(err)
    }
}
```

5.4.2 实现

接口是用来定义类型的行为。这些被定i的行为不由接口直接实现，而是通过方法由用户定义的类型实现。如果用户定义的类型实现了接口的一组方法之后，那么这个用户定义的类型的值就可以赋给这个接口类型的值。

在这种关系里，用户定义的类型通常称为实体类型。

5.4.3 方法集

方法集定义了接口的接收规则。

```go
// Sample program to show how to use an interface in Go.
package main

import (
    "fmt"
)

// notifier is an interface that defined notification
// type behavior.
type notifier interface {
    notify()
}

// user defines a user in the program.
type user struct {
    name  string
    email string
}

// notify implements a method with a pointer receiver.
func (u *user) notify() {
    fmt.Printf("Sending user email to %s<%s>\n",
        u.name,
        u.email)
}

// main is the entry point for the application.
func main() {
    // Create a value of type User and send a notification.
    u := user{"Bill", "bill@email.com"}

    sendNotification(u)

    // ./listing36.go:32: cannot use u (type user) as type
    //                      notifier in argument to sendNotification:
    //   user does not implement notifier
    //                           (notify method has pointer receiver)
}

// sendNotification accepts values that implement the notifier
// interface and sends notifications.
func sendNotification(n notifier) {
    n.notify()
}
```

上述代码编译不通过的原因在于：

- 如果使用指针接收者来实现了一个接口，那么只有指向那个类型的指针才能够实现对应的接口
- 如果使用值接收者来实现一个接口，那么那个类型的值和指针都能够实现对应的接口

从方法接收者类型的角度来看方法集

methods receivers	values
(t T)	T and *T
(t *T)	*T

5.4.4 多态

```go
// Sample program to show how polymorphic behavior with interfaces.
package main

import (
    "fmt"
)

// notifier is an interface that defines notification
// type behavior.
type notifier interface {
    notify()
}

// user defines a user in the program.
type user struct {
    name  string
    email string
}

// notify implements the notifier interface with a pointer receiver.
func (u *user) notify() {
    fmt.Printf("Sending user email to %s<%s>\n",
        u.name,
        u.email)
}

// admin defines a admin in the program.
type admin struct {
    name  string
    email string
}

// notify implements the notifier interface with a pointer receiver.
func (a *admin) notify() {
    fmt.Printf("Sending admin email to %s<%s>\n",
        a.name,
        a.email)
}

// main is the entry point for the application.
func main() {
    // Create a user value and pass it to sendNotification.
    bill := user{"Bill", "bill@email.com"}
    sendNotification(&bill)

    // Create an admin value and pass it to sendNotification.
    lisa := admin{"Lisa", "lisa@email.com"}
    sendNotification(&lisa)
}

// sendNotification accepts values that implement the notifier
// interface and sends notifications.
func sendNotification(n notifier) {
```

```
    n.notify()
}
```

5.5 嵌入类型

嵌入类型是将已有的类型直接声明在新的结构里，从而实现扩展或修改已有类型的行为。被嵌入的类型被称为新的外部类型的内部类型。

要嵌入一个类型，只需要在外部类型中声明之歌类型的名字就可以了。

内部类型的值以及实现的接口都会自动的提升到外部类型，但当外部类型也有相同的值或者实现了相同的结构时，内部类型将不会提升，但此时仍可以通过内部类型的标识来访问。

```go
// Sample program to show what happens when the outer and inner
// type implement the same interface.
package main

import (
    "fmt"
)

// notifier is an interface that defined notification
// type behavior.
type notifier interface {
    notify()
}

type person struct {
    age int
    gender string
}

// user defines a user in the program.
type user struct {
    person
    name  string
    email string
}

// notify implements a method that can be called via
// a value of type user.
func (u *user) notify() {
    fmt.Printf("Sending user email to %s<%s>\n",
        u.name,
        u.email)
}

// admin represents an admin user with privileges.
type admin struct {
    user
    level string
    age int
}

// notify implements a method that can be called via
// a value of type Admin.
func (a *admin) notify() {
    fmt.Printf("Sending admin email to %s<%s>\n",
        a.name,
        a.email)
}

// main is the entry point for the application.
func main() {
    // Create an admin user.

    ad := admin{
```

```
        user: user{
            name:  "john smith",
            email: "john@yahoo.com",
            person: person{
                age: 24,
                gender: "male",
            },
        },
        level: "super",
        age: 26,
    }

    // Send the admin user a notification.
    // The embedded inner type's implementation of the
    // interface is NOT "promoted" to the outer type.
    sendNotification(&ad)

    // We can access the inner type's method directly.
    ad.user.notify()

    // The inner type's method is NOT promoted.
    ad.notify()

    fmt.Println(ad.user.person.gender)
    fmt.Println(ad.person.gender)
    fmt.Println(ad.gender)
    fmt.Println(ad.user.person.age)
    fmt.Println(ad.person.age)
    fmt.Println(ad.age)
}

// sendNotification accepts values that implement the notifier
// interface and sends notifications.
func sendNotification(n notifier) {
    n.notify()
}

// Sending admin email to john smith<john@yahoo.com>
// Sending user email to john smith<john@yahoo.com>
// Sending admin email to john smith<john@yahoo.com>
// male
// male
// male
// 24
// 24
// 26
```

5.6 公开或未公开的标识符

Go 语言支持公开或隐藏标识符。

当一个标识符的名字以小写字母开头，那么这个标识符就是未公开的，在包外不可见

当一个标识符的名字以大写字母开头，那么这个标识符就是公开的，在包外可见

5.7 小结

- 使用关键字 struct 或通过指定已存在的类型，可以声明用户定义的类型
- 方法提供了一种给用户定义的类型增加行为的方式
- 设计类型时需要确认类型的本质是原始的还是非原始的
- 接口是声明一组行为并支持多态的类型
- 嵌入类型提供了扩展类型的功能，而无需使用继承
- 标识符要么是从包里公开的，要么是在包里未公开的

6 并发

Go 语言里的并发指的是能让某个函数独立于其他函数运行的能力。当一个函数创建为 goroutine 时，Go 会将其视为一个独立的工作单元。这个单元会被调度到可用的逻辑处理器上执行。

Go 语言运行时的调度器管理着所有的 goroutine 并为其分配执行时间。这个调度器在操作系统之上，将操作系统的线程与语言运行时的逻辑处理器绑定，并在逻辑处理器上运行 goroutine。

Go 语言的并发模型来自一个叫做通信顺序进程（CSP）。CSP 是一种消息传递模型，通过在 goroutine 之间传递数据，而不是，对数据进行加锁来实现同步。通道（channel）是用来在 goroutine 之间传递数据的数据模型

6.1 并发与并行

进程可以看作一个包含了应用程序运行需要用到和维护的各种资源的容器。线程是一个执行空间，用于被操作系统调度。一个进程至少包含一个主线程。

并行是让不同的代码片段同时在不同的物理处理器上运行

并发是指同时管理着很多事情，这些事情可能做到一半就被暂停去做别的事情了

一般，并发的效果比并行的效果要好，因为操作系统的硬件和资源有限，而要做的事情有很多。

在 Go 语言里，如果要让 goroutine 并行，必须使用多于一个逻辑处理器，否则将会是并发执行。

6.2 goroutine

代码实例

```go
// This sample program demonstrates how to create goroutines and
// how the goroutine scheduler behaves with two logical processor.
package main

import (
    "fmt"
    "runtime"
    "sync"
)

// main is the entry point for all Go programs.
func main() {
    // Allocate two logical processors for the scheduler to use.
    runtime.GOMAXPROCS(2)

    // wg is used to wait for the program to finish.
    // Add a count of two, one for each goroutine.
    var wg sync.WaitGroup
    wg.Add(2)

    fmt.Println("Start Goroutines")

    // Declare an anonymous function and create a goroutine.
    go func() {
        // Schedule the call to Done to tell main we are done.
        defer wg.Done()

        // Display the alphabet three times.
        for count := 0; count < 3; count++ {
            for char := 'a'; char < 'a'+26; char++ {
                fmt.Printf("%c ", char)
            }
        }
    }()

    // Declare an anonymous function and create a goroutine.
    go func() {
        // Schedule the call to Done to tell main we are done.
        defer wg.Done()

        // Display the alphabet three times.
        for count := 0; count < 3; count++ {
            for char := 'A'; char < 'A'+26; char++ {
                fmt.Printf("%c ", char)
            }
        }
    }()

    // Wait for the goroutines to finish.
    fmt.Println("Waiting To Finish")
    wg.Wait()

    fmt.Println("\nTerminating Program")
```

```
}
```

6.3 竞争状态

如果两个或多个 goroutine 在没有同步的情况下，访问某个共享的资源，并试图同时读写这个资源，就会处于相互竞争状态。

竞争状态的存在，让并发程序变得复杂，很容易出现问题。

竞争状态的 goroutine

```go
// This sample program demonstrates how to create race
// conditions in our programs. We don't want to do this.
package main

import (
    "fmt"
    "runtime"
    "sync"
)

var (
    // counter is a variable incremented by all goroutines.
    counter int

    // wg is used to wait for the program to finish.
    wg sync.WaitGroup
)

// main is the entry point for all Go programs.
func main() {
    // Add a count of two, one for each goroutine.
    wg.Add(2)

    // Create two goroutines.
    go incCounter(1)
    go incCounter(2)

    // Wait for the goroutines to finish.
    wg.Wait()
    fmt.Println("Final Counter:", counter)
}

// incCounter increments the package level counter variable.
func incCounter(id int) {
    // Schedule the call to Done to tell main we are done.
    defer wg.Done()

    for count := 0; count < 2; count++ {
        // Capture the value of Counter.
        value := counter

        // Yield the thread and be placed back in queue.
        runtime.Gosched()

        // Increment our local value of Counter.
        value++

        // Store the value back into Counter.
        counter = value
    }
}

// output: 2
```

使用 `go build -race` 可以在 build 时检测存在竞争状态的代码

6.4 锁住共享资源

Go 语言提供了传统的同步 goroutine 机制，那就是对共享资源加锁。

atomic 和 sync 包里的函数提供了很好的解决方案

6.4.1 原子函数

原子函数能够以很低层的加锁机制来同步访问整形变量和指针。

```go
// This sample program demonstrates how to use the atomic
// package to provide safe access to numeric types.
package main

import (
    "fmt"
    "runtime"
    "sync"
    "sync/atomic"
)

var (
    // counter is a variable incremented by all goroutines.
    counter int64

    // wg is used to wait for the program to finish.
    wg sync.WaitGroup
)

// main is the entry point for all Go programs.
func main() {
    // Add a count of two, one for each goroutine.
    wg.Add(2)

    // Create two goroutines.
    go incCounter(1)
    go incCounter(2)

    // Wait for the goroutines to finish.
    wg.Wait()

    // Display the final value.
    fmt.Println("Final Counter:", counter)
}

// incCounter increments the package level counter variable.
func incCounter(id int) {
    // Schedule the call to Done to tell main we are done.
    defer wg.Done()

    for count := 0; count < 2; count++ {
        // Safely Add One To Counter.
        atomic.AddInt64(&counter, 1)

        // Yield the thread and be placed back in queue.
        runtime.Gosched()
    }
}
```

```go
// This sample program demonstrates how to use the atomic
// package functions Store and Load to provide safe access
// to numeric types.
package main

import (
    "fmt"
    "sync"
    "sync/atomic"
    "time"
)

var (
    // shutdown is a flag to alert running goroutines to shutdown.
    shutdown int64

    // wg is used to wait for the program to finish.
    wg sync.WaitGroup
)

// main is the entry point for all Go programs.
func main() {
    // Add a count of two, one for each goroutine.
    wg.Add(2)

    // Create two goroutines.
    go doWork("A")
    go doWork("B")

    // Give the goroutines time to run.
    time.Sleep(1 * time.Second)

    // Safely flag it is time to shutdown.
    fmt.Println("Shutdown Now")
    atomic.StoreInt64(&shutdown, 1)

    // Wait for the goroutines to finish.
    wg.Wait()
}

// doWork simulates a goroutine performing work and
// checking the Shutdown flag to terminate early.
func doWork(name string) {
    // Schedule the call to Done to tell main we are done.
    defer wg.Done()

    for {
        fmt.Printf("Doing %s Work\n", name)
        time.Sleep(250 * time.Millisecond)

        // Do we need to shutdown.
        if atomic.LoadInt64(&shutdown) == 1 {
            fmt.Printf("Shutting %s Down\n", name)
```

```
            break
        }
    }
}
```

6.4.2 互斥锁

另一种同步访问的共享资源的方式就是使用互斥锁。互斥锁用户在代码上创建一个临界区，保证同一时间只有一个 goroutine 可以执行这个临界区代码。

```go
// This sample program demonstrates how to use a mutex
// to define critical sections of code that need synchronous
// access.
package main

import (
    "fmt"
    "runtime"
    "sync"
)

var (
    // counter is a variable incremented by all goroutines.
    counter int

    // wg is used to wait for the program to finish.
    wg sync.WaitGroup

    // mutex is used to define a critical section of code.
    mutex sync.Mutex
)

// main is the entry point for all Go programs.
func main() {
    // Add a count of two, one for each goroutine.
    wg.Add(2)

    // Create two goroutines.
    go incCounter(1)
    go incCounter(2)

    // Wait for the goroutines to finish.
    wg.Wait()
    fmt.Printf("Final Counter: %d\n", counter)
}

// incCounter increments the package level Counter variable
// using the Mutex to synchronize and provide safe access.
func incCounter(id int) {
    // Schedule the call to Done to tell main we are done.
    defer wg.Done()

    for count := 0; count < 2; count++ {
        // Only allow one goroutine through this
        // critical section at a time.
        mutex.Lock()
        {
            // Capture the value of counter.
            value := counter

            // Yield the thread and be placed back in queue.
            runtime.Gosched()
```

```
            // Increment our local value of counter.
            value++

            // Store the value back into counter.
            counter = value
        }
        mutex.Unlock()
        // Release the lock and allow any
        // waiting goroutine through.
    }
}
```

6.5 通道

原子函数和互斥锁都能工作，但是依靠他们都不会让编写并发程序变得简单、更不易出错或者有趣。

在 Go 语言里，你不仅可以使用原子函数和互斥锁来保证对共享资源的安全访问和消除竞争资源，还可以使用通道，通过发送和接受需要共享的资源，在 goroutine 之间做同步。

当一个资源需要在 goroutine 之间共享时，通道在 goroutine 之间架起了一个管道，并提供了确保同步交换数据的机制。

- 使用 make 创建通道

```
// 无缓冲的整形通道
unbuffered := make(chan int)
// 有缓冲的字符串通道
buffered := make(chan string, 10)
```

- 向通道发送值

```
buffered := make(chan string, 10)
buffered <- "hello Go!"
```

- 从通道接收值

```
value := <- buffered
```

6.5.1 无缓冲通道

无缓冲通道是指在接收前没有能力保存任何值的通道。

无缓冲类型通道要求发送 goroutine 和接收 goroutine 同时准备好，才能完成发送和接收操作。如果双方没有同时准备好，通道会先导致发送或接收操作的 goroutine 组塞等待。

在无缓冲通道里的这种发送和接收的交互行为本身就是同步的，其中任意一个操作都无法离开另一个操作单独存在。

网球比赛

```go
// This sample program demonstrates how to use an unbuffered
// channel to simulate a game of tennis between two goroutines.
package main

import (
    "fmt"
    "math/rand"
    "sync"
    "time"
)

// wg is used to wait for the program to finish.
var wg sync.WaitGroup

func init() {
    rand.Seed(time.Now().UnixNano())
}

// main is the entry point for all Go programs.
func main() {
    // Create an unbuffered channel.
    court := make(chan int)

    // Add a count of two, one for each goroutine.
    wg.Add(2)

    // Launch two players.
    go player("Nadal", court)
    go player("Djokovic", court)

    // Start the set.
    court <- 1

    // Wait for the game to finish.
    wg.Wait()
}

// player simulates a person playing the game of tennis.
func player(name string, court chan int) {
    // Schedule the call to Done to tell main we are done.
    defer wg.Done()

    for {
        // Wait for the ball to be hit back to us.
        ball, ok := <-court
        if !ok {
            // If the channel was closed we won.
            fmt.Printf("Player %s Won\n", name)
            return
        }

        // Pick a random number and see if we miss the ball.

        n := rand.Intn(100)
```

```
        if n%13 == 0 {
            fmt.Printf("Player %s Missed\n", name)

            // Close the channel to signal we lost.
            close(court)
            return
        }

        // Display and then increment the hit count by one.
        fmt.Printf("Player %s Hit %d\n", name, ball)
        ball++

        // Hit the ball back to the opposing player.
        court <- ball
    }
}
```

4 个 goroutine 之间的接力比赛

```go
// This sample program demonstrates how to use an unbuffered
// channel to simulate a relay race between four goroutines.
package main

import (
    "fmt"
    "sync"
    "time"
)

// wg is used to wait for the program to finish.
var wg sync.WaitGroup

// main is the entry point for all Go programs.
func main() {
    // Create an unbuffered channel.
    baton := make(chan int)

    // Add a count of one for the last runner.
    wg.Add(1)

    // First runner to his mark.
    go Runner(baton)

    // Start the race.
    baton <- 1

    // Wait for the race to finish.
    wg.Wait()
}

// Runner simulates a person running in the relay race.
func Runner(baton chan int) {
    var newRunner int

    // Wait to receive the baton.
    runner := <-baton

    // Start running around the track.
    fmt.Printf("Runner %d Running With Baton\n", runner)

    // New runner to the line.
    if runner != 4 {
        newRunner = runner + 1
        fmt.Printf("Runner %d To The Line\n", newRunner)
        go Runner(baton)
    }

    // Running around the track.
    time.Sleep(100 * time.Millisecond)

    // Is the race over.
    if runner == 4 {
```

```
            fmt.Printf("Runner %d Finished, Race Over\n", runner)
            wg.Done()
            return
        }

        // Exchange the baton for the next runner.
        fmt.Printf("Runner %d Exchange With Runner %d\n",
            runner,
            newRunner)

        baton <- newRunner
    }
```

6.5.2 有缓冲通道

有缓冲通道是一种在接收前能够存储一个或多个值的通道，它并不强制要求 goroutine 之间必须同时完成发送和接收，这是与无缓冲通道最大的区别。

有缓冲通道也会阻塞，缓冲区已满，会在发送方阻塞，缓冲区为空，会在接收方阻塞。

```go
// This sample program demonstrates how to use a buffered
// channel to work on multiple tasks with a predefined number
// of goroutines.
package main

import (
    "fmt"
    "math/rand"
    "sync"
    "time"
)

const (
    numberGoroutines = 4  // Number of goroutines to use.
    taskLoad         = 10 // Amount of work to process.
)

// wg is used to wait for the program to finish.
var wg sync.WaitGroup

// init is called to initialize the package by the
// Go runtime prior to any other code being executed.
func init() {
    // Seed the random number generator.
    rand.Seed(time.Now().Unix())
}

// main is the entry point for all Go programs.
func main() {
    // Create a buffered channel to manage the task load.
    tasks := make(chan string, taskLoad)

    // Launch goroutines to handle the work.
    wg.Add(numberGoroutines)
    for gr := 1; gr <= numberGoroutines; gr++ {
        go worker(tasks, gr)
    }

    // Add a bunch of work to get done.
    for post := 1; post <= taskLoad; post++ {
        tasks <- fmt.Sprintf("Task : %d", post)
    }

    // Close the channel so the goroutines will quit
    // when all the work is done.
    close(tasks)

    // Wait for all the work to get done.
    wg.Wait()
}

// worker is launched as a goroutine to process work from
// the buffered channel.
```

```
func worker(tasks chan string, worker int) {
    // Report that we just returned.
    defer wg.Done()

    for {
        // Wait for work to be assigned.
        task, ok := <-tasks
        if !ok {
            // This means the channel is empty and closed.
            fmt.Printf("Worker: %d : Shutting Down\n", worker)
            return
        }

        // Display we are starting the work.
        fmt.Printf("Worker: %d : Started %s\n", worker, task)

        // Randomly wait to simulate work time.
        sleep := rand.Int63n(100)
        time.Sleep(time.Duration(sleep) * time.Millisecond)

        // Display we finished the work.
        fmt.Printf("Worker: %d : Completed %s\n", worker, task)
    }
}
```

6.6 小结

- 并发是指 goroutine 运行的时候相互独立
- 使用关键字 go 创建 goroutine 来运行函数
- goroutine 在逻辑处理器上执行，而逻辑处理器具有独立的系统线程和运行队列
- 竞争状态是指两个或者多个 goroutine 之间试图访问同一资源
- 原子函数和互斥锁提供了一种防止出现竞争状态的办法
- 通道提供了一种在两个 goroutine 之间共享数据的简单方法
- 无缓冲通道保证同时交换数据，有缓冲通道不做这种保证

7 并发模式

在本章我们将学习如何使用包来简化并发程序的编写，以及为什么能简化的原因

7.1 runner

runner 包用于展示如何使用通道来监视程序的执行时间，如果程序的运行时间太长，也可以用 runner 包来终止程序。

当开发需要调度后台处理任务的程序的时候，这种模式会很有用。

runner 包源码

```go
// Example is provided with help by Gabriel Aszalos.
// Package runner manages the running and lifetime of a process.
package runner

import (
    "errors"
    "os"
    "os/signal"
    "time"
)

// Runner runs a set of tasks within a given timeout and can be
// shut down on an operating system interrupt.
type Runner struct {
    // interrupt channel reports a signal from the
    // operating system.
    interrupt chan os.Signal

    // complete channel reports that processing is done.
    complete chan error

    // timeout reports that time has run out.
    timeout <-chan time.Time

    // tasks holds a set of functions that are executed
    // synchronously in index order.
    tasks []func(int)
}

// ErrTimeout is returned when a value is received on the timeout channel.
var ErrTimeout = errors.New("received timeout")

// ErrInterrupt is returned when an event from the OS is received.
var ErrInterrupt = errors.New("received interrupt")

// New returns a new ready-to-use Runner.
func New(d time.Duration) *Runner {
    return &Runner{
        interrupt: make(chan os.Signal, 1),
        complete:  make(chan error),
        timeout:   time.After(d),
    }
}

// Add attaches tasks to the Runner. A task is a function that
// takes an int ID.
func (r *Runner) Add(tasks ...func(int)) {
    r.tasks = append(r.tasks, tasks...)
}

// Start runs all tasks and monitors channel events.
func (r *Runner) Start() error {

    // We want to receive all interrupt based signals.
```

```
    signal.Notify(r.interrupt, os.Interrupt)

    // Run the different tasks on a different goroutine.
    go func() {
        r.complete <- r.run()
    }()

    select {
    // Signaled when processing is done.
    case err := <-r.complete:
        return err

    // Signaled when we run out of time.
    case <-r.timeout:
        return ErrTimeout
    }
}

// run executes each registered task.
func (r *Runner) run() error {
    for id, task := range r.tasks {
        // Check for an interrupt signal from the OS.
        if r.gotInterrupt() {
            return ErrInterrupt
        }

        // Execute the registered task.
        task(id)
    }

    return nil
}

// gotInterrupt verifies if the interrupt signal has been issued.
func (r *Runner) gotInterrupt() bool {
    select {
    // Signaled when an interrupt event is sent.
    case <-r.interrupt:
        // Stop receiving any further signals.
        signal.Stop(r.interrupt)
        return true

    // Continue running as normal.
    default:
        return false
    }
}
```

使用示例

```
// This sample program demonstrates how to use a channel to
// monitor the amount of time the program is running and terminate
// the program if it runs too long.
package main

import (
    "log"
    "os"
    "time"

    "github.com/goinaction/code/chapter7/patterns/runner"
)

// timeout is the number of second the program has to finish.
const timeout = 3 * time.Second

// main is the entry point for the program.
func main() {
    log.Println("Starting work.")

    // Create a new timer value for this run.
    r := runner.New(timeout)

    // Add the tasks to be run.
    r.Add(createTask(), createTask(), createTask())

    // Run the tasks and handle the result.
    if err := r.Start(); err != nil {
        switch err {
        case runner.ErrTimeout:
            log.Println("Terminating due to timeout.")
            os.Exit(1)
        case runner.ErrInterrupt:
            log.Println("Terminating due to interrupt.")
            os.Exit(2)
        }
    }

    log.Println("Process ended.")
}

// createTask returns an example task that sleeps for the specified
// number of seconds based on the id.
func createTask() func(int) {
    return func(id int) {
        log.Printf("Processor - Task #%d.", id)
        time.Sleep(time.Duration(id) * time.Second)
    }
}
```

7.2 pool

这个包用于展示如何使用有缓冲的通道来实现资源池，来管理可以在任意数量的 goroutine 之间共享及独立使用的资源。

这种模式在需要共享一组静态资源的情况下非常有用。如果 goroutine 需要从池里得到这些资源中的一个，它们可以从池里申请，使用完后归还到资源池。

pool 包

```
// Example provided with help from Fatih Arslan and Gabriel Aszalos.
// Package pool manages a user defined set of resources.
package pool

import (
    "errors"
    "io"
    "log"
    "sync"
)

// Pool manages a set of resources that can be shared safely by
// multiple goroutines. The resource being managed must implement
// the io.Closer interface.
type Pool struct {
    m         sync.Mutex
    resources chan io.Closer
    factory   func() (io.Closer, error)
    closed    bool
}

// ErrPoolClosed is returned when an Acquire returns on a
// closed pool.
var ErrPoolClosed = errors.New("Pool has been closed.")

// New creates a pool that manages resources. A pool requires a
// function that can allocate a new resource and the size of
// the pool.
func New(fn func() (io.Closer, error), size uint) (*Pool, error) {
    if size <= 0 {
        return nil, errors.New("Size value too small.")
    }

    return &Pool{
        factory:   fn,
        resources: make(chan io.Closer, size),
    }, nil
}

// Acquire retrieves a resource from the pool.
func (p *Pool) Acquire() (io.Closer, error) {
    select {
    // Check for a free resource.
    case r, ok := <-p.resources:
        log.Println("Acquire:", "Shared Resource")
        if !ok {
            return nil, ErrPoolClosed
        }
        return r, nil

    // Provide a new resource since there are none available.
    default:
        log.Println("Acquire:", "New Resource")
```

```
            return p.factory()
        }
    }

    // Release places a new resource onto the pool.
    func (p *Pool) Release(r io.Closer) {
        // Secure this operation with the Close operation.
        p.m.Lock()
        defer p.m.Unlock()

        // If the pool is closed, discard the resource.
        if p.closed {
            r.Close()
            return
        }

        select {
        // Attempt to place the new resource on the queue.
        case p.resources <- r:
            log.Println("Release:", "In Queue")

        // If the queue is already at cap we close the resource.
        default:
            log.Println("Release:", "Closing")
            r.Close()
        }
    }

    // Close will shutdown the pool and close all existing resources.
    func (p *Pool) Close() {
        // Secure this operation with the Release operation.
        p.m.Lock()
        defer p.m.Unlock()

        // If the pool is already close, don't do anything.
        if p.closed {
            return
        }

        // Set the pool as closed.
        p.closed = true

        // Close the channel before we drain the channel of its
        // resources. If we don't do this, we will have a deadlock.
        close(p.resources)

        // Close the resources
        for r := range p.resources {
            r.Close()
        }
    }
```

示例

```
// This sample program demonstrates how to use the pool package
// to share a simulated set of database connections.
package main

import (
    "io"
    "log"
    "math/rand"
    "sync"
    "sync/atomic"
    "time"

    "github.com/goinaction/code/chapter7/patterns/pool"
)

const (
    maxGoroutines   = 25 // the number of routines to use.
    pooledResources = 2  // number of resources in the pool
)

// dbConnection simulates a resource to share.
type dbConnection struct {
    ID int32
}

// Close implements the io.Closer interface so dbConnection
// can be managed by the pool. Close performs any resource
// release management.
func (dbConn *dbConnection) Close() error {
    log.Println("Close: Connection", dbConn.ID)
    return nil
}

// idCounter provides support for giving each connection a unique id.
var idCounter int32

// createConnection is a factory method that will be called by
// the pool when a new connection is needed.
func createConnection() (io.Closer, error) {
    id := atomic.AddInt32(&idCounter, 1)
    log.Println("Create: New Connection", id)

    return &dbConnection{id}, nil
}

// main is the entry point for all Go programs.
func main() {
    var wg sync.WaitGroup
    wg.Add(maxGoroutines)

    // Create the pool to manage our connections.
    p, err := pool.New(createConnection, pooledResources)

    if err != nil {
```

```
            log.Println(err)
        }

        // Perform queries using connections from the pool.
        for query := 0; query < maxGoroutines; query++ {
            // Each goroutine needs its own copy of the query
            // value else they will all be sharing the same query
            // variable.
            go func(q int) {
                performQueries(q, p)
                wg.Done()
            }(query)
        }

        // Wait for the goroutines to finish.
        wg.Wait()

        // Close the pool.
        log.Println("Shutdown Program.")
        p.Close()
    }

    // performQueries tests the resource pool of connections.
    func performQueries(query int, p *pool.Pool) {
        // Acquire a connection from the pool.
        conn, err := p.Acquire()
        if err != nil {
            log.Println(err)
            return
        }

        // Release the connection back to the pool.
        defer p.Release(conn)

        // Wait to simulate a query response.
        time.Sleep(time.Duration(rand.Intn(1000)) * time.Millisecond)
        log.Printf("Query: QID[%d] CID[%d]\n", query, conn.(*dbConnection).ID)
    }
```

7.3 work

work 包的目的是如何使用无缓存通道创建一个 goroutine 池。它允许使用者知道什么时候 goroutine 池正在工作，而且如果池里的所有 goroutine 都忙，无法接受新的工作的时候，也能及时通过通道通知调用者。

使用无缓冲通道能确保不会有工作在队列里丢失或者卡住，所有的工作都会被处理。

work 包

```go
// Example provided with help from Jason Waldrip.
// Package work manages a pool of goroutines to perform work.
package work

import "sync"

// Worker must be implemented by types that want to use
// the work pool.
type Worker interface {
    Task()
}

// Pool provides a pool of goroutines that can execute any Worker
// tasks that are submitted.
type Pool struct {
    work chan Worker
    wg   sync.WaitGroup
}

// New creates a new work pool.
func New(maxGoroutines int) *Pool {
    p := Pool{
        work: make(chan Worker),
    }

    p.wg.Add(maxGoroutines)
    for i := 0; i < maxGoroutines; i++ {
        go func() {
            for w := range p.work {
                w.Task()
            }
            p.wg.Done()
        }()
    }

    return &p
}

// Run submits work to the pool.
func (p *Pool) Run(w Worker) {
    p.work <- w
}

// Shutdown waits for all the goroutines to shutdown.
func (p *Pool) Shutdown() {
    close(p.work)
    p.wg.Wait()
}
```

示例

```
// This sample program demonstrates how to use the work package
// to use a pool of goroutines to get work done.
package main

import (
    "log"
    "sync"
    "time"

    "github.com/goinaction/code/chapter7/patterns/work"
)

// names provides a set of names to display.
var names = []string{
    "steve",
    "bob",
    "mary",
    "therese",
    "jason",
}

// namePrinter provides special support for printing names.
type namePrinter struct {
    name string
}

// Task implements the Worker interface.
func (m *namePrinter) Task() {
    log.Println(m.name)
    time.Sleep(time.Second)
}

// main is the entry point for all Go programs.
func main() {
    // Create a work pool with 2 goroutines.
    p := work.New(2)

    var wg sync.WaitGroup
    wg.Add(100 * len(names))

    for i := 0; i < 100; i++ {
        // Iterate over the slice of names.
        for _, name := range names {
            // Create a namePrinter and provide the
            // specific name.
            np := namePrinter{
                name: name,
            }

            go func() {
                // Submit the task to be worked on. When RunTask
                // returns we know it is being handled.
                p.Run(&np)
```

```
                wg.Done()
            }()
        }
    }

    wg.Wait()

    // Shutdown the work pool and wait for all existing work
    // to be completed.
    p.Shutdown()
}
```

8 标准库

Go 标准库是一组核心包，用来扩展和增强语言的能力，会随着 Go 语言的升级而更新并保持向后兼容

8.1 文档与源代码

Go 语言的标准库里有超过 100 个包被分在了 38 个类别里面，详情请见Packages

8.2 记录日志

```go
// This sample program demonstrates how to create customized loggers.
package main

import (
    "io"
    "io/ioutil"
    "log"
    "os"
)

var (
    Trace   *log.Logger // Just about anything
    Info    *log.Logger // Important information
    Warning *log.Logger // Be concerned
    Error   *log.Logger // Critical problem
)

func init() {
    file, err := os.OpenFile("errors.txt",
        os.O_CREATE|os.O_WRONLY|os.O_APPEND, 0666)
    if err != nil {
        log.Fatalln("Failed to open error log file:", err)
    }

    Trace = log.New(ioutil.Discard,
        "TRACE: ",
        log.Ldate|log.Ltime|log.Lshortfile)

    Info = log.New(os.Stdout,
        "INFO: ",
        log.Ldate|log.Ltime|log.Lshortfile)

    Warning = log.New(os.Stdout,
        "WARNING: ",
        log.Ldate|log.Ltime|log.Lshortfile)

    Error = log.New(io.MultiWriter(file, os.Stderr),
        "ERROR: ",
        log.Ldate|log.Ltime|log.Lshortfile)
}

func main() {
    Trace.Println("I have something standard to say")
    Info.Println("Special Information")
    Warning.Println("There is something you need to know about")
    Error.Println("Something has failed")
}
```

8.3 编码/解码

8.3.1 解码 json

```go
// This sample program demonstrates how to decode a JSON response
// using the json package and NewDecoder function.
package main

import (
    "encoding/json"
    "fmt"
    "log"
    "net/http"
)

type (
    // gResult maps to the result document received from the search.
    gResult struct {
        GsearchResultClass string `json:"GsearchResultClass"`
        UnescapedURL       string `json:"unescapedUrl"`
        URL                string `json:"url"`
        VisibleURL         string `json:"visibleUrl"`
        CacheURL           string `json:"cacheUrl"`
        Title              string `json:"title"`
        TitleNoFormatting  string `json:"titleNoFormatting"`
        Content            string `json:"content"`
    }

    // gResponse contains the top level document.
    gResponse struct {
        ResponseData struct {
            Results []gResult `json:"results"`
        } `json:"responseData"`
    }
)

func main() {
    uri := "http://ajax.googleapis.com/ajax/services/search/web?v=1.0&rsz=8&q=golang"

    // Issue the search against Google.
    resp, err := http.Get(uri)
    if err != nil {
        log.Println("ERROR:", err)
        return
    }
    defer resp.Body.Close()

    // Decode the JSON response into our struct type.
    var gr gResponse
    err = json.NewDecoder(resp.Body).Decode(&gr)
    if err != nil {
        log.Println("ERROR:", err)
        return
    }

    fmt.Println(gr)
```

```
    // Marshal the struct type into a pretty print
    // version of the JSON document.
    pretty, err := json.MarshalIndent(gr, "", "    ")
    if err != nil {
        log.Println("ERROR:", err)
        return
    }

    fmt.Println(string(pretty))
}
```

当需要处理的 JSON 文档以 string 的形式存在时，可以将 string 转换为 byte 切片([] byte)，并使用 json 包的 Unmarshal 函数进行反序列化的处理

```go
// This sample program demonstrates how to decode a JSON string.
package main

import (
    "encoding/json"
    "fmt"
    "log"
)

// Contact represents our JSON string.
type Contact struct {
    Name    string `json:"name"`
    Title   string `json:"title"`
    Contact struct {
        Home string `json:"home"`
        Cell string `json:"cell"`
    } `json:"contact"`
}

// JSON contains a sample string to unmarshal.
var JSON = `{
    "name": "Gopher",
    "title": "programmer",
    "contact": {
        "home": "415.333.3333",
        "cell": "415.555.5555"
    }
}`

func main() {
    // Unmarshal the JSON string into our variable.
    var c Contact
    err := json.Unmarshal([]byte(JSON), &c)
    if err != nil {
        log.Println("ERROR:", err)
        return
    }

    fmt.Println(c)
}
```

有时，无法为 JSON 的格式声明一个结构类型，而是需要更加灵活的方式来处理 JSON 文档

```go
// This sample program demonstrates how to decode a JSON string.
package main

import (
    "encoding/json"
    "fmt"
    "log"
)

// JSON contains a sample string to unmarshal.
var JSON = `{
    "name": "Gopher",
    "title": "programmer",
    "contact": {
        "home": "415.333.3333",
        "cell": "415.555.5555"
    }
}`

func main() {
    // Unmarshal the JSON string into our map variable.
    var c map[string]interface{}
    err := json.Unmarshal([]byte(JSON), &c)
    if err != nil {
        log.Println("ERROR:", err)
        return
    }

    fmt.Println("Name:", c["name"])
    fmt.Println("Title:", c["title"])
    fmt.Println("Contact")
    fmt.Println("H:", c["contact"].(map[string]interface{})["home"])
    fmt.Println("C:", c["contact"].(map[string]interface{})["cell"])
}
```

8.3.2 编码 JSON

```
// This sample program demonstrates how to marshal a JSON string.
package main

import (
    "encoding/json"
    "fmt"
    "log"
)

func main() {
    // Create a map of key/value pairs.
    c := make(map[string]interface{})
    c["name"] = "Gopher"
    c["title"] = "programmer"
    c["contact"] = map[string]interface{}{
        "home": "415.333.3333",
        "cell": "415.555.5555",
    }

    // Marshal the map into a JSON string.
    data, err := json.MarshalIndent(c, "", "    ")
    if err != nil {
        log.Println("ERROR:", err)
        return
    }

    fmt.Println(string(data))
}
```

8.4 输入和输出

输出(Write 接口)

```go
// Sample program to show how different functions from the
// standard library use the io.Writer interface.
package main

import (
    "bytes"
    "fmt"
    "os"
)

// main is the entry point for the application.
func main() {
    // Create a Buffer value and write a string to the buffer.
    // Using the Write method that implements io.Writer.
    var b bytes.Buffer
    b.Write([]byte("Hello "))

    // Use Fprintf to concatenate a string to the Buffer.
    // Passing the address of a bytes.Buffer value for io.Writer.
    fmt.Fprintf(&b, "World!")

    // Write the content of the Buffer to the stdout device.
    // Passing the address of a os.File value for io.Writer.
    b.WriteTo(os.Stdout)
}
```

```go
// Sample program to show how to write a simple version of curl using
// the io.Reader and io.Writer interface support.
package main

import (
    "io"
    "log"
    "net/http"
    "os"
)

// main is the entry point for the application.
func main() {
    // r here is a response, and r.Body is an io.Reader.
    r, err := http.Get(os.Args[1])
    if err != nil {
        log.Fatalln(err)
    }

    // Create a file to persist the response.
    file, err := os.Create(os.Args[2])
    if err != nil {
        log.Fatalln(err)
    }
    defer file.Close()

    // Use MultiWriter so we can write to stdout and
    // a file on the same write operation.
    dest := io.MultiWriter(os.Stdout, file)

    // Read the response and write to both destinations.
    io.Copy(dest, r.Body)
    if err := r.Body.Close(); err != nil {
        log.Println(err)
    }
}
```