02 | 内有乾坤：Go语言六大基础知识体系

2022-10-11 郑建勋 来自北京

《Go进阶 · 分布式爬虫实战》

课程介绍 >

讲述：郑建勋

时长 12:32　大小 11.45M

你好，我是郑建勋。

这节课，我们继续来回顾 Go 语言的基础知识，帮助你在前期查漏补缺，打好项目开发的基础。在上节课，我把 Go 的基础知识分为了六个部分，分别是开发环境、基础语法、语法特性、并发编程、项目组织、工具与库。

现在，我们紧跟上节课的内容，继续后面四个部分的讲解。

语法特性

Go 语言中有许多特别的语法特性，其中比较主要的特性包括了 defer、接口、协程、通道等。让我们先从 defer 说起。

defer

defer 是 Go 语言中的关键字，也是 Go 语言的重要特性之一，defer 在资源释放、panic 捕获等场景中的应用非常广泛。

复制代码

```
defer func(...){
// 实际处理
}()
```

我们需要掌握 defer 的几个重要特性，包括：

- 延迟执行；
- 参数预计算；
- LIFO 执行顺序。

除此之外，Go 语言用于异常恢复的内置 recover 函数，也需要与 defer 函数结合使用才有意义：

复制代码

```
func f() {
    defer func() {
        if r := recover(); r != nil {
            fmt.Println("Recovered in f", r)
        }
    }()
    fmt.Println("Calling g.")
    g(0)
    fmt.Println("Returned normally from g.")
}
```

接口

接口是 Go 中实现模块解耦、代码复用还有控制系统复杂度的重要手段，因此，了解接口的使用方法和应用场景是很必要的。在后面的课程中，我还会详细地介绍接口的最佳实践和设计模式。这节课我们先来看看接口的基本用法。

Go 中的接口有两种类型，分别为“带方法的接口”和“空接口”（不考虑泛型的情况）。 带方法的接口内部有一系列方法签名：

```
type InterfaceName interface {
  fA()
  fB(a int,b string) error
  ...
}
```


而空接口内部不包含任何东西，可存储任意类型：

```
type Empty interface{}
```

接口这部分，我们需要掌握的知识点包括下面这些。

- 接口的声明与定义：

```
type Shape interface {
  perimeter() float64
  area() float64
}
var s Shape
```

- 隐式地让一个类型实现接口：

```
type Rectangle struct {
  a, b float64
}
func (r Rectangle) perimeter() float64 {
  return (r.a + r.b) * 2
}
func (r Rectangle) area() float64 {
  return r.a * r.b
}
```

- 接口的动态调用方式：

复制代码

```
var s Shape
s = Rectangle{3, 4}
s.perimeter()
s.area()
```


- 接口的嵌套：

复制代码

```
type ReadWriter interface {
  Reader
  Writer
}
type Reader interface {
  Read(p []byte) (n int, err error)
}
type Writer interface {
  Write(p []byte) (n int, err error)
}
```

- 接口类型断言：

复制代码

```
func main(){
  var s Shape
  s = Rectangle{3, 4}
  rect := s.(Rectangle)
  fmt.Printf("长方形周长:%v, 面积:%v \\n",rect.perimeter(),rect.area())
}
```

- 根据空接口中动态类型的差异选择不同的处理方式（这在参数为空接口函数的内部处理中使用广泛，例如 fmt 库、JSON 库）：

复制代码

```
switch f := arg.(type) {
  case bool:
    p.fmtBool(f, verb)
  case float32:
    p.fmtFloat(float64(f), 32, verb)
  case float64:
    p.fmtFloat(f, 64, verb)
```

- 接口的比较性，具体规则为：
  - 动态类型值为 nil 的接口变量总是相等的
  - 如果只有 1 个接口为 nil，那么比较结果总是 false
  - 如果两个接口都不为 nil，且接口变量具有相同的动态类型和动态类型值，那么两个接口是相同的。关于类型的可比较性，可以参见上一节课中结构体的比较性。
  - 如果接口存储的动态类型值是不可比较的，那么在运行时会报错。

除此之外，Go 语言的特性包括：处理元数据的反射、处理指针的 unsafe 包，以及调用 c 函数的 cgo 等。反射的基础是接口，实践中反射使用不多，但在一些基础库、网络库中使用较多。unsafe 包语义上不具备兼容性；cgo 书写与调试困难，不受运行时的管理。这些技术比较复杂而且不常使用，所以我没有把它们放到“基础知识”这一部分。

Go 中还有一块重要的语法特性涉及到并发原语，即协程与通道。因为并发极为重要，我们单拎出来讲解。

并发编程

Go 向来以容易编写高并发程序而闻名，这是它区别于其他语言的特点之一。在 Go 语言中与并发编程紧密相关的就是协程与通道。对于协程，我们首先要区分下面几个重要的概念。

- **进程、线程与协程。**进程是操作系统资源分配的基本单位，线程是操作系统资源调度的基本单位。而协程位于用户态，是在线程基础上构建的轻量级调度单位。
- **并发与并行。**并行指的是同时做很多事情，并发是指同时管理很多事情。
- **主协程与子协程。** main 函数是特殊的主协程，它退出之后整个程序都会退出。而其他的协程都是子协程，子协程退出之后，程序正常运行。

Go 语言运行时为我们托管了协程的启动与调度工作，我们关心的重点只要放在如何优雅安全地关闭协程，以及如何进行协程间的通信就可以了。

Go 语言实现了 CSP 并发编程模式，把通道当作 Go 语言中的一等公民，通道的基本使用方式包括下面几点。

- 通道声明与初始化：


```
chan int
chan <- float
<-chan string
```

- 通道写入数据：


```
c <- 5
```

- 通道读取数据：


```
data := <-c
```

- 通道关闭：


```
close(c)
```

- 通道作为参数：


```
func worker(id int, c chan int) {
  for n := range c {
    fmt.Printf("Worker %d received %c\\n",
      id, n)
  }
}
```

- 通道作为返回值（一般用于创建通道的阶段）：

```go
func createWorker(id int) chan int {
  c := make(chan int)
  go worker(id, c)
  return c
}
```


- 单方向的通道，用于只读和只写场景：

```go
func worker(id int, c <-chan int)
```

- select 监听多个通道实现多路复用。当 case 中多个通道状态准备就绪时，select 随机选择一个分支进行执行：

```go
select {
  case <-ch1:
    // ...
  case x := <-ch2:
    // ...use x...
  case ch3 <- y:
    // ...
  default:
    // ...
  }
```

Go 除了使用通道完成协程间的通信外，还提供了一些其他手段。

- 用 context 来处理协程的优雅退出和级联退出，我们在后面的课程中还会详细介绍。

```go
func Stream(ctx context.Context, out chan<- Value) error {
  for {
    v, err := DoSomething(ctx)
    if err != nil {
      return err
    }
    select {
    case <-ctx.Done():
      return ctx.Err()
```

```
    case out <- v:
    }
  }
```


- 传统的同步原语：原子锁。Go 提供了 atomic 包用于处理原子操作。

复制代码

```
func add() {
  for {
    if atomic.CompareAndSwapInt64(&flag, 0, 1) {
      count++
      atomic.StoreInt64(&flag, 0)
      return
    }
  }
}
```

- 传统的同步原语：互斥锁。

复制代码

```
var m sync.Mutex
func add() {
  m.Lock()
  count++
  m.Unlock()
}
```

- 传统的同步原语：读写锁。适合多读少写场景。

复制代码

```
type Stat struct {
  counters map[string]int64
  mutex sync.RWMutex
}
func (s *Stat) getCounter(name string) int64 {
  s.mutex.RLock()
  defer s.mutex.RUnlock()
  return s.counters[name]
}
func (s *Stat) SetCounter(name string){
  s.mutex.Lock()
  defer s.mutex.Unlock()
```

```
    s.counters[name]++
}
```


- 除此之外，Go 语言在传统的同步原语基础上还提供了许多有用的同步工具，包括 sync.Once、sync.Cond、sync.WaitGroup。我们在后面介绍高并发模型的课程中，还会详细介绍它们。

项目组织

刚才，我们总括式地了解了 Go 语言中的一些基本语法，但是要构建一个大型系统，我们还需要站在巨人的肩膀上，使用其他人已经写好的代码库。这时，我们需要管理好项目依赖的第三方包。

依赖管理

Go 的依赖管理经历了长时间的演进。 现如今，Go Module 已经成为了依赖管理的事实标准，掌握 Go Module 的基础用法已经成为 Go 语言使用者的必备技能（关于 Go Module 的演进、使用和原理，后面我们还会有更深入的介绍）。

复制代码

```
module github.com/dreamerjackson/crawler

go 1.18

require (
  github.com/PuerkitoBio/goquery v1.8.0 // indirect
  github.com/andybalholm/cascadia v1.3.1 // indirect
  github.com/antchfx/htmlquery v1.2.5 // indirect
  github.com/antchfx/xpath v1.2.1 // indirect
)
```

除此之外，理解 GOPATH 这种单一工作区的依赖管理方式也是非常有必要的，因为它现阶段并没有完全被废弃。

面向组合

构建大规模程序需要我们完成必要的抽象，这样才能屏蔽一些细节，然后从更高的层面去构建大规模程序。之前，我们介绍了一些比较经典的思想，比如函数用于过程的抽象、自定义结构体用于数据的抽象。如果你是一个“老学究”，我真的建议你去阅读一下《Structure and

Interpretation of Computer Programs》这本书，感受一下这些我们习以为常的简单元素背后的非凡哲学。

在理解了过程抽象与数据抽象之后，我们再来看另一种、简单而又强大的设计哲学——面向组合。面向组合可以帮助我们完成功能之间的正交组合，轻松构建起复杂的程序，还可以使我们灵活应对程序在未来的变化。

下面我举一个在 IO 操作中实现面向组合思想的例子，代码如下所示。

复制代码

```
type Reader interface {
    Read(p []byte) (n int, err error)
}

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

// 组合了Read与Write功能
type ReadWriter interface {
    Reader
    Writer
}

type doc struct{
   file *os.File
}

func (d *doc) Read(p []byte) (n int, err error){
    p,err := ioutil.ReadAll(d.file)
    ...
}

// v1 版本
func handle(r Reader){
  r.Read()
   ...
}

// v2 版本
func handle(rw ReadWriter){
  r.Read()
  r.Write()
  ...
}
```

Reader 接口包含了 Read 方法，Writer 接口包含了 Write 方法。假设我们的业务是处理文档相关的操作，类型 doc 一开始实现了 Read 功能，将文件内容读取到传递的缓冲区中，函数 handle 中的参数为接口 Reader。

后来随着业务发展，我们又需要实现文档写的功能，这时我们将函数 handle 的参数修改为功能更强大的接口 ReadWriter。而 doc 只需要实现 Writer 接口，就隐式地实现了这个 ReadWriter 接口。这样，通过组合，我们就不动声色地完成了代码与功能的扩展。

如果你想进一步了解结构体与接口的组合，可以查看 Effective Go 的描述。后面的课程我们还会实践面向组合的设计哲学。

工具与库

Go 致力于成为大规模软件项目的优秀工具，因此它不仅需要在语法上给出自己的解决方案，还需要在整个软件的生命周期内（编辑、测试、编译、部署、调试、分析）有完善的标准库和工具。随着 Go 的发展，也出现了越来越多优秀的第三方库。

工具：代码分析与代码规范

Go 自带了许多工具，它们可以规范代码、提高可读性，促进团队协作、检查代码错误等。

我们可以将这种工具分为静态与动态两种类型。其中，静态工具对代码进行静态扫描，检查代码的结构、代码风格以及语法错误，这种工具也被称为 Linter。

- go fmt

静态工具包括了我们熟知的`go fmt`，它可以格式化代码，规范代码的风格：

```
go fmt path/to/your/package
```

复制代码

除了`go fmt`，也可以直接使用`gofmt`命令对单独的文件进行格式化。

```
gofmt -w yourcode.go
```

复制代码

`gofmt` 还可以完成替换的工作，这里不再展开，如果想了解更多，你可以用`gofmt --help` 查看帮助文档。

- go doc

go doc 工具可以生成和阅读代码的文档说明。文档是使软件可访问和可维护的重要组成部分。当然，它需要写得好且准确，也需要易于编写和维护。理想情况下，文档注释应该与代码本身耦合，以便文档与代码一起发展。 go doc 可以解析 Go 源代码（包括注释）， 并生成 HTML 或纯文本形式的文档。例如可以查看标准库 bufio 的文档。

- go vet

go vet 是 Go 官方提供的代码静态诊断器，他可以对代码风格进行检查，并且报告可能有问题的地方，例如错误的锁使用、不必要的赋值等。 go vet 启发式的问题诊断方法不能保证所有输出都真正有问题，但它确实可以找到一些编译器无法捕获的错误。

由于 go vet 本身是多种 Linter 的聚合器，我们可以通过`go tool vet help` 命令查看它拥有的功能。Go 语言为这种代码的静态分析提供了标准库go/analysis，这意味着我们只用遵循一些通用的规则就可以写出适合自己的分析工具。这还意味着我们可以对众多的静态分析器进行选择、合并。

- golangci-lint

不过，当前企业中使用得最普遍的不是 go vet 而是golangci-lint。这是因为 Go 中的分析器非常容易编写，社区已经创建了许多有用的 Linter，而 golangci-lint 正是对多种 Linter 的集合。要查看 golangci-lint 支持的 Linter 列表以及 golangci-lint 启用 / 禁用哪些 Linter，可以通过`golangci-lint help linters` 查看帮助文档或者查看golangci-lint 的官方文档。

- go race

Go 1.1 后提供了强大的检查工具 race，它可以排查数据争用问题。race 可以用在多个 Go 指令中。

```
$ go test -race mypkg
$ go run -race mysrc.go
$ go build -race mycmd
$ go install -race mypkg
```


当检测器在程序中发现数据争用时，将打印报告。这份报告包含发生 race 冲突的协程栈，以及此时正在运行的协程栈。


```
» go run -race 2_race.go
==================
WARNING: DATA RACE
Read at 0x00000115c1f8 by goroutine 7:
main.add()
bookcode/concurrence_control/2_race.go:5 +0x3a
Previous write at 0x00000115c1f8 by goroutine 6:
main.add()
bookcode/concurrence_control/2_race.go:5 +0x56
```

动态工具指的是需要实际运行指定代码才能够分析出问题的工具。go race 工具可以完成静态分析，但是有些并发冲突是静态分析难以发现的，所以 go race 在运行时也可以开启，完成动态的数据争用检测，一般在上线之前使用。

除此之外，动态工具还包括了代码测试、调试等阶段使用到的工具。

工具：代码测试

- **go test**

在 Go 中，测试函数位于单独的以 _test.go 结尾的文件中，测试函数名以`Test` 开头。go test 会识别这些测试文件并进行测试。测试包括单元测试、Benchmark 测试等。

单元测试指的是测试代码中的某一个函数或者功能，它能帮助我们验证函数功能是否正常，各种边界条件是否符合预期。单元测试是保证代码健壮性的重要手段。

Go 中比较有特色的单元测试叫做表格测试。通过表格测试可以简单地测试多个场景。如下所示。另外，测试中还有一些特性例如 t.Run 支持并发测试，这能加快测试的速度，即便某一个子测试（subtest）失败，其他子测试也会完成测试。

复制代码

```go
func TestSplit(t *testing.T) {
  tests := map[string]struct {
    input string
    sep   string
    want  []string
  }{
    "simple":       {input: "a/b/c", sep: "/", want: []string{"a", "b", "c"}},
    "wrong sep":    {input: "a/b/c", sep: ",", want: []string{"a/b/c"}},
    "no sep":       {input: "abc", sep: "/", want: []string{"abc"}},
    "trailing sep": {input: "a/b/c/", sep: "/", want: []string{"a", "b", "c"}},
  }

  for name, tc := range tests {
    t.Run(name, func(t *testing.T) {
      got := Split(tc.input, tc.sep)
      if !reflect.DeepEqual(tc.want, got) {
        t.Fatalf("expected: %#v, got: %#v", tc.want, got)
      }
    })
  }
}
```


- **go test -cover**

  执行 go test 命令时，加入 cover 参数能够统计出测试代码的覆盖率。

复制代码

```
% go test -cover
PASS
coverage: 42.9% of statements
ok      size    0.026s
%
```

- **go tool cover**

另外，我们还可以收集覆盖率文件并进行可视化的展示。

具体的做法是，在执行 go test 命令时加入 coverprofile 参数，生成代码覆盖率文件。然后使用 `go tool cover` 可视化分析代码覆盖率的信息。

复制代码

```
> go test -coverprofile .coverage.txt
```

```
> go tool cover -func .coverage.txt
```


- **go test -bench**

  Go 还可以进行 Benchmark 测试，要测试函数的前缀名需要为 Benchmark。

```go
func BenchmarkFibonacci(b *testing.B) {
  for i := 0; i < b.N; i++ {
    _ = Fibonacci(30)
  }
}
```

默认情况下，执行 go test -bench 之后，程序会在 1 秒后打印出在这段时间里内部函数的运行次数和时间。

在进行 BenchMark 测试时，我们还可以指定一些其他运行参数。例如，“-benchmem”可以打印每次函数的内存分配情况，“cpuprofile”、“memprofile”还能收集程序的 CPU 和内存的 profile 文件，方便后续 pprof 工具进行可视化分析。

```
go test ./fibonacci \\
  -bench BenchmarkSuite \\
  -benchmem \\
  -cpuprofile=cpu.out \\
  -memprofile=mem.out
```

工具：代码调试

在程序调试阶段，除了可以借助原始的日志打印消息，我们还可以使用一些常用的程序分析与调试的工具。

- dlv

dlv 是 Go 官方提供的一个简单、功能齐全的调试工具，它和传统的调试器 gdb 的使用方式比较类似，但是 dlv 还专门提供了与 Go 匹配的功能，例如查看协程栈，切换到指定协程等。我们在后面的课程中还会有 dlv 的实战演练。

- gdb

gdb 是通用的程序调试器，但它并不是 Go 程序在调试时最优的选项。gdb 可能在有些方面比较有用，例如调试 cgo 程序或运行时。不过在一般情况下，建议你优先选择 dlv。

- pprof

pprof 是 Go 语言中对指标或特征进行分析的工具。通过 pprof，我们不仅可以找到程序中的错误（内存泄漏、race 冲突、协程泄漏），也能找到程序的优化点（CPU 利用率不足等）。

pprof 包含了样本数据的收集以及对样本进行分析两个阶段。收集样本简单的方式是借助 net/http/pprof 标准库提供的一套 HTTP 接口访问。

```
curl -o cpu.out  http://localhost:9981/debug/pprof/profile?seconds=30
```

而要对收集到的特征文件进行分析，需要依赖谷歌提供的分析工具，该工具在 Go 语言处理器安装时就存在：

```
go tool pprof -http=localhost:8000  cpu.out
```

- trace

我们在 pprof 的分析中，能够知道一段时间内 CPU 的占用、内存分配、协程堆栈等信息。这些信息都是一段时间内数据的汇总，但是它们并没有提供整个周期内事件的全貌。例如，指定的协程何时执行，执行了多长时间，什么时候陷入了堵塞，什么时候解除了堵塞，GC 是如何影响协程执行的，STW 中断花费的时间有多长等。而 Go1.5 之后推出的 trace 工具解决了这些问题。trace 的强大之处在于，提供了程序在指定时间内发生的事件的完整信息，让我们可以精准地排查出程序的问题所在，在后面的课程中，还会用 trace 完成对线上实战案例的分析。

- gops

gops 是谷歌推出的调试工具，它的作用是诊断系统当前运行的 Go 进程。gops 可以显示出当前系统中所有的 Go 进程，并可以查看特定进程的堆栈信息、内存信息等。

标准库

标准库是官方维护的，用于增强和扩展语言的核心库。标准库一般涵盖了通用的场景以及现代开发所需的核心部分。例如，用于数学运算的 math 包、用于 I/O 处理的 io 包，用于处理字符串的 strings 包，用于处理网络的 net 包，以及用于处理 HTTP 协议的 http 包。

由于标准库经过了大量的测试，有稳定性的保证，并且提供了向后兼容性。开发者可以借助标准库快速完成开发。

Go 提供了众多的标准库：

```
复制代码
archive   bufio      bytes     compress   container   crypto    database
debug     encoding   errors    expvar     flag        fmt       go
hash      html       image     index      io          log       math
mime      net        os        path       reflect     regexp    runtime
sort      strconv    strings   sync       syscall     testing   text
time      unicode    unsafe
```

相比其他语言，开发者对 Go 标准库的依赖更多。Go 标准库中提供了丰富的内容和强有力的封装，比如 HTTP 库就对 HTTP 协议进行了大量的封装，TCP 连接底层也通过封装 epoll/kqueue/iocp 实现了 I/O 多路复用。开发者使用这种开箱即用的特性就可以相对轻松地写出简洁、高性能的程序。

第三方库

Go 语言中，优秀的第三方库、框架和软件可谓汗牛充栋，就拿 HTTP 框架来说吧，我们比较熟悉的就有 Echo、Gin、Beego 等知名的框架。这些优秀的代码都非常值得借鉴与学习。在项目中，我们将会使用非常多的第三方库，你也可以参考 awesome-go 中列出的众多优秀的 Go 代码库。

总结

之前的两节课，我们梳理了 Go 语言的基础知识，相信你已经感受了 Go 庞大知识体系和蓬勃发展的生态。我在这两节课中对基础知识的讲解起到了一个提纲挈领的作用，它可以帮助你快速地进行查漏补缺。

为了帮助初学者学习 Go 语言的基础知识，我在业余时间也在完成一套开源的入门课程《Go 语言开挂入门之旅》，并且在 B 站进行了视频讲解。感兴趣的同学可以参与到这门课程的创作中来，帮助其他人学习也是自我提升的好方式。

课后题

最后，我也给你留一道思考题。

你如何理解 Go 语言的一句名言：“不要通过共享内存来通信，通过通信来共享内存”？

欢迎你在留言区与我交流讨论，我们下节课再见！

上一篇 01丨知识回顾：Go基础知识你真的掌握了吗？

下一篇 03丨进阶路线：如何深入学习Go语言？

精选留言 (6)

写留言

G55

2022-10-13 来自北京

通过共享内存通信相当于双方必须依靠额外的控制机制来确保通信时内存中的内容是正确的，这一点需要共享双方设置同步机制，并不通用, 还容易有bug。但是通过通信共享内存则可以

利用通用的通信基建, 凡是经过通信传递的信息一定是发送方确认正确的, 接收方只需要等待即可, 不用再依赖额外的同步机制，减少了出bug的机会。

作者回复: 说得很好，通道一种所有权转移的机制，为我们屏蔽了锁等机制。 通过一些简单的并发模型(例如fan-in、fan-out),开发并行程序会变得非常容易

1

那时刻

2022-10-13 来自北京

并行指的是同时做很多事情，并发是指同时管理很多事情。请问老师，管理和做具体区别是什么？

作者回复: 同时管理很多事情指的是这些事情可能不是同时发生的，也不知道发生的先后顺序，但是通过一些调度让他们都有执行的机会。 而并行指的是当前时刻，事情同时在发生。
并发和并行是需要梳理的经典概念，CPU核心数量对应了能够并行执行的数量。但是一个机器上可能有上千上万的线程、协程在等待执行,他们其实是并发的

1

温雅小公子

2022-10-12 来自北京

太贴心了吧，还有视频。

作者回复: 😁哈哈，我后面有时间了还会陆续更新视频，现在全力搞专栏

共 3 条评论>

1

陈卧虫

2022-10-12 来自北京

个人理解，内存是信息的载体，共享内存的目的就是为了传递信息，而共享内存只是传递信息的一种手段。共享内存的特点是信息存一块公共的内存区域的，每个线程主动来获取并竞争它的使用权，这个过程中就必须通过加锁来保证原子性。通过通信（channel）来共享内存，就是将线程的主动竞争变为了只能被动等待，接收信息，而消息只会传递给其中一个线程，谁拥有消息（从channel中获得），谁就拥有修改权，这样整个过程就不需要加锁。

作者回复: 这种所有权转移的机制非常妙哈，其实是通道为我们屏蔽了一些细节，非常有表现力

1

includestdio.h

2022-10-14 来自陕西

想问下老师B站的视频完结了吗，我看好久没更新了

go 入门入了快一年了，看过各种 go 专栏和书籍，但是苦于没有实操经验，一直停留在理论，理论也是边学边忘，希望通过这门进阶课入门 -.-


Geek_b4e7f6

2022-10-14 来自辽宁

新手报到