15丨众人拾柴：高效团队的Go编码规范

2022-11-12 郑建勋 来自北京

《Go进阶 · 分布式爬虫实战》

讲述：郑建勋

时长 02:50　大小 2.60M

你好，我是郑建勋。

在前面两个模块，我们回顾了 Go 语言的基础知识，掌握了 Go 项目的开发流程，也完成了爬虫项目的架构设计、功能设计和流程设计。不过，为了写出“好”的代码，我们必须规范代码，这就需要定义好整个团队需要遵守的编程规范了。

我们为什么需要编程规范？

编程规范又叫代码规范，是团队之间在程序开发时需要遵守的约定。俗话说，无规矩不成方圆，一个开发团队应该就一种编程规范达成一致。编程规范有很多好处，我们简单说几个最主要的。

- **促进团队合作**

现代项目大多是由团队完成的，但是如果每个人书写出的代码风格迥异，最后集成代码时很容易杂乱无章、可读性极差。相反，风格统一的代码将大大提高可读性，易于理解，促进团队协作。

- **规避错误**

  每一种语言都有容易犯的错误，Go 语言也不例外。但是编码规范可以规避掉像 Map 并发读写等问题。不仅如此，规范的日志处理、错误处理还能够加快我们查找问题的速度。

- **提升性能**

  优秀的开发者，能够在头脑中想象出不同程序运行的过程和结果，写出高性能的程序非常考验开发者的内功。但每个人的水平都有差异，这一点并不可控。但是如果我们将高性能编程的常见手段归纳整理出来，开发者只需要遵守这些简单的规则，就能够规避性能陷阱、极大提升程序性能。

- **便于维护**

  我们习惯于关注编写代码的成本，但实际上维护代码的成本要高得多。大部分的项目是在前人的基础上完成开发的。我们在开发代码的时候，也会花大量时间阅读之前的代码。符合规范的代码更容易上手维护、更少出现牵一发动全身的耦合现象、也更容易看出业务处理逻辑。

知道了编程规范的好处，那我们应该规范什么内容呢？这其实涉及到我们对好代码的定义。针对这个问题，我想每个人都能够说个几句。我认为，好的代码首先是整洁、一致的，同时它还是高效、健壮和可扩展的。

接下来，我们就从这几个维度聊聊制定 Go 编码规范的原则和最佳实践。

再多说一句，有一些规范可以是强制的，因为我们可以通过工具和代码 review 强制要求用户遵守，还有一些规范是建议的，因为它更具有灵活性，很难被约束。在后面的规范中，[强制 xxx]中的“xxx”代表的就是可强制检查的工具。

整洁、一致

好代码的第一个要求，是整洁和一致。有一句话是这样说的：

> Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

它的意思是，任何傻瓜都可以编写计算机可以理解的代码，而优秀的程序员编写的是人类可以理解的代码。

如果我们的代码看起来乱七八糟，就像喝醉的人写的那样，这样不严谨的代码让我们有理由相信，项目的其他各个方面也隐藏着对细节的疏忽，并埋下了重大的隐患。

阅读整洁的代码就像看到精心设计的手表或汽车一样赏心悦目，因为它凝聚了团队的智慧。

阅读整洁的代码也像读到的武侠小说，书中的文字被脑中的图像取代，你看到了角色，听到了声音，体验到了悲怆和幽默。

但是，明白什么是整洁的代码并不意味着你能写出整洁的代码。就好像我们知道如何欣赏一幅画不意味着我们能成为画家。

整洁的代码包括对格式化、命名、函数等细节的密切关注，更需要在项目中具体实践。接下来我们就来看看整洁代码关注的这些细节和最佳的实践。

格式化

1. 代码长度

代码应该有足够的垂直密度，能够肉眼一次获得到更多的信息。同时，单个函数、单行、单文件也需要限制长度，保证可阅读性和可维护性。

[强制 lll] 一行内不超过 120 个字符，同时应当避免刻意断行。如果你发现某一行太长了，要么改名，要么调整语义，往往就可以解决问题了。

[强制 funlen] 单个函数的行数不超过 40 行，过长则表示函数功能不专一、定义不明确、程序结构不合理，不易于理解。当函数过长时，可以提取函数以保持正文小且易读。

[强制] 单个文件不超过 2000 行，过长说明定义不明确，程序结构划分不合理，不利于维护。

2. 代码布局

我们先试想一篇写得很好的报纸文章。在顶部，你希望有一个标题，它会告诉你故事的大致内容，并影响你是否要阅读它。文章的第一段会为你提供整个故事的概要、粗略的概念，但是隐藏了所有细节。继续向下阅读，详细信息会逐步增加。

[建议] Go 文件推荐按以下顺序进行布局。

1.包注释：对整个模块和功能的完整描述，写在文件头部。

2.Package：包名称。

3.Imports：引入的包。

4.Constants：常量定义。

5.Typedefs：类型定义。

6.Globals：全局变量定义。

7.Functions：函数实现。

每个部分之间用一个空行分割。每个部分有多个类型定义或者有多个函数时，也用一个空行分割。示例如下：

复制代码

```
/*
注释
*/
package http

import (
 "fmt"
 "time"
)

const (
 VERSION = "1.0.0"
)

type Request struct{
}
```

```
17
18 var msg = "HTTP success"
19
20 func foo() {
21  //...
22 }
```

[强制 goimports] 当 import 多个包时，应该对包进行分组。同一组的包之间不需要有空行，不同组之间的包需要一个空行。标准库的包应该放在第一组。

3. 空格与缩进

为了让阅读代码时视线畅通，自上而下思路不被打断，我们需要使用一些空格和缩进。

空格是为了分离关注点，将不同的组件分开。缩进是为了处理错误和边缘情况，与正常的代码分隔开。

较常用的有下面这些规范：

[强制 gofmt] 注释和声明应该对齐。示例如下：

复制代码

```
type T struct {
    name    string // name of the object
    value   int    // its value
}
```

[强制 gofmt] 小括号 ()、中括号[]、大括号{} 内侧都不加空格。

[强制 gofmt] 逗号、冒号（slice 中冒号除外）前都不加空格，后面加 1 个空格。

[强制 gofmt] 所有二元运算符前后各加一个空格，作为函数参数时除外。例如`b := 1 + 2`。
[强制 gofmt] 使用 Tab 而不是空格进行缩进。

[强制 nlreturn] return 前方需要加一个空行，让代码逻辑更清晰。

[强制 gofmt] 判断语句、for 语句需要缩进 1 个 Tab，并且右大括号}与对应的 if 关键字垂直对齐。例如：

复制代码

```
if xxx {

} else {

}
```

[强制 goimports] 当 import 多个包时，应该对包进行分组。同一组的包之间不需要有空行，不同组之间的包需要一个空行。标准库的包应该放在第一组。这同样适用于常量、变量和类型声明：

复制代码

```
import (
    "fmt"
    "hash/adler32"
    "os"

    "appengine/foo"
    "appengine/user"
    "github.com/foo/bar"
    "rsc.io/goversion/version"
)
```

[推荐] 避免 else 语句中处理错误返回，避免正常的逻辑位于缩进中。如下代码实例，else 中进行错误处理，代码逻辑阅读起来比较费劲。

复制代码

```
if something.OK() {
        something.Lock()
        defer something.Unlock()
        err := something.Do()
        if err == nil {
                stop := StartTimer()
                defer stop()
                log.Println("working...")
                doWork(something)
                <-something.Done() // wait for it
                log.Println("finished")
                return nil
```

```
        } else {
                return err
        }
} else {
        return errors.New("something not ok")
}
```


如果把上面的代码修改成下面这样会更加清晰：

复制代码

```
if !something.OK() {
    return errors.New("something not ok")
}
something.Lock()
defer something.Unlock()
err := something.Do()
if err != nil {
    return err
}
stop := StartTimer()
defer stop()
log.Println("working...")
doWork(something)
<-something.Done() // wait for it
log.Println("finished")
return nil
```

[推荐] 函数内不同的业务逻辑处理建议用单个空行加以分割。

[推荐] 注释之前的空行通常有助于提高可读性——新注释的引入表明新思想的开始。

命名

> Good naming is like a good joke. If you have to explain it, it's not funny.
> ———*Dave Cheney*

一个好的名字应该满足几个要素：

- 短，容易拼写；
- 保持一致性；

- 意思准确，容易理解，没有虚假和无意义的信息。

例如，像下面这样的命名就是让人迷惑的：


```
1 int d; // elapsed time in days
```

复制代码

[强制 revive] Go 中的命名统一使用驼峰式、不要加下划线。

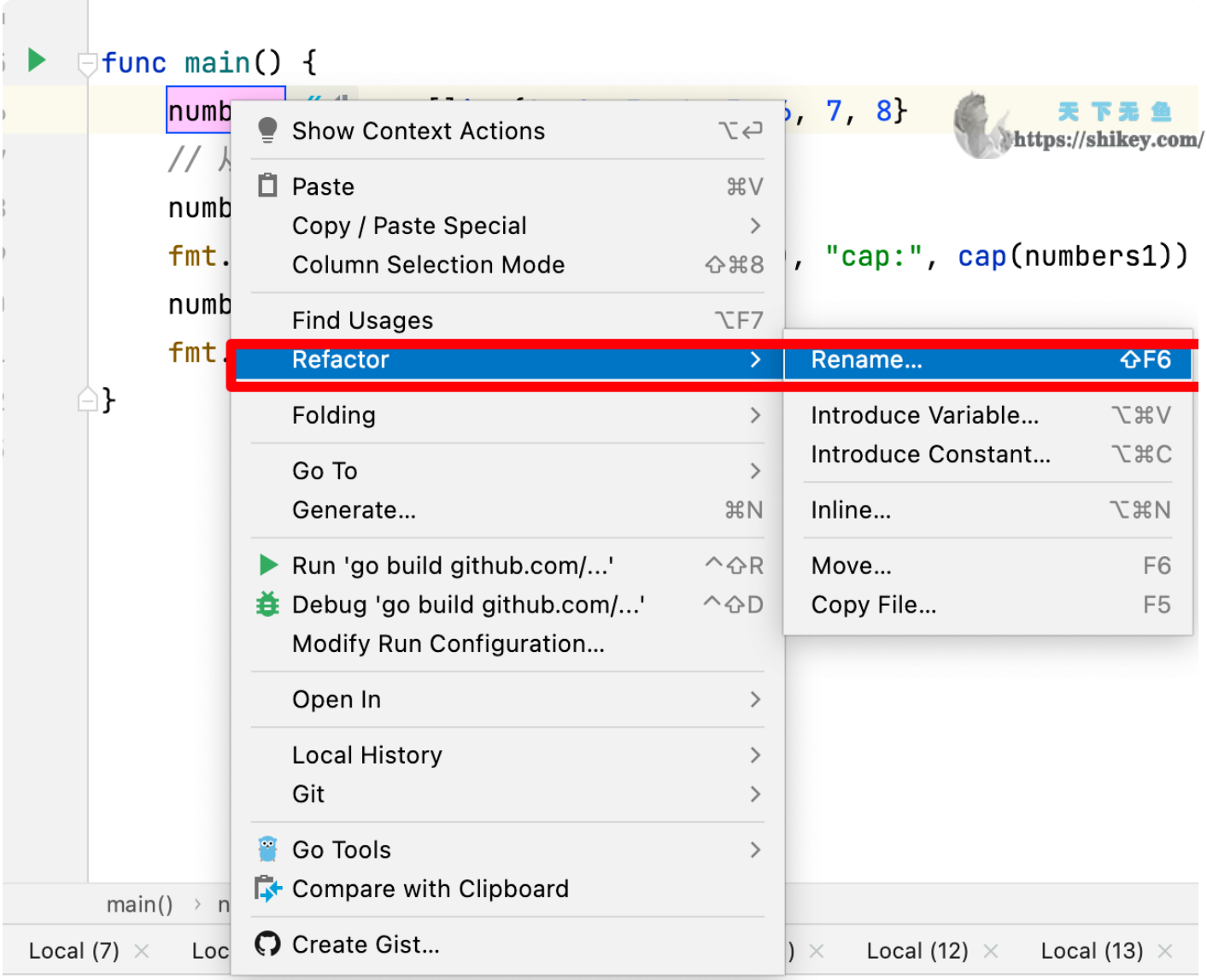
[强制 revive] 缩写的专有名词应该大写，例如： ServeHTTP、IDProcessor。

[强制] 区分变量名应该用有意义的名字，而不是使用阿拉伯数字：a1, a2, … aN。

[强制] 不要在变量名称中包含你的类型名称。

[建议]变量的作用域越大，名字应该越长。

现代 IDE 已经让更改名称变得更容易了，巧妙地使用 IDE 的功能，能够级联地同时修改多处命名。



包名

包名应该简短而清晰。

[强制] 使用简短的小写字母，不需要下划线或混合大写字母。

[建议] 合理使用缩写，例如：

```
strconv（字符串转换）
syscall（系统调用）
fmt（格式化的 I/O）
```

[强制] 避免无意义的包名，例如 `util`，`common`，`base` 等。

接口命名

[建议]单方法接口由方法名称加上 -er 后缀或类似修饰来命名。例如：`Reader`, `Writer`, `Formatter`, `CloseNotifier` ，当一个接口包含多个方法时，请选择一个能够准确描述其用途的名称（例如：net.Conn、http.ResponseWriter、io.ReadWriter）。

本地变量命名

[建议]尽可能地短。在这里，i 指代 index，r 指代 reader，b 指代 buffer。

例如，下面这段代码就可以做一个简化：

复制代码

```
for index := 0; index < len(s); index++ {
  //
}
```

可以替换为：

复制代码

```
for i := 0; i < len(s); i++ {
  //
}
```

函数参数命名

[建议]如果函数参数的类型已经能够看出参数的含义，那么函数参数的命名应该尽量简短：

复制代码

```
func AfterFunc(d Duration, f func()) *Timer
func Escape(w io.Writer, s []byte)
```

[建议]如果函数参数的类型不能表达参数的含义，那么函数参数的命名应该尽量准确：

```
func Unix(sec, nsec int64) Time
func HasPrefix(s, prefix []byte) bool
```

复制代码

函数返回值命名

[建议] 对于公开的函数，返回值具有文档意义，应该准确表达含义，如下所示：

复制代码

```
func Copy(dst Writer, src Reader) (written int64, err error)

func ScanBytes(data []byte, atEOF bool) (advance int, token []byte, err error)
```

可导出的变量名

[建议] 由于使用可导出的变量时会带上它所在的包名，因此，不需要对变量重复命名。例如 bytes 包中的 ByteBuffer 替换为 Buffer，这样在使用时就是 bytes.Buffer，显得更简洁。类似的还有把 strings.StringReader 修改为 strings.Reader，把 **errors.NewError 修改为 errors.New。**

Error 值命名

[建议] 错误类型应该以 Error 结尾。

[建议] Error 变量名应该以 Err 开头。

复制代码

```
type ExitError struct {
    ...
}
var ErrFormat = errors.New("image: unknown format")
```

函数

[强制 cyclop] 圈复杂度（Cyclomatic complexity）<10。

[强制 gochecknoinits] 避免使用 init 函数。

[强制 revive] Context 应该作为函数的第一个参数。

[强制] 正常情况下禁用 unsafe。

[强制] 禁止 return 裸返回，如下例中第一个 return：

复制代码

```
func (f *Filter) Open(name string) (file File, err error) {
  for _, c := range f.chain {
    file, err = c.Open(name)
    if err != nil {
      return
    }
  }
  return f.source.Open(name)
}
```

[强制] 不要在循环里面使用 defer，除非你真的确定 defer 的工作流程。

[强制] 对于通过:= 进行变量赋值的场景，禁止出现仅部分变量初始化的情况。例如在下面这个例子中，f 函数返回的 res 是初始化的变量，但是函数返回的 err 其实复用了之前的 err：

复制代码

```
var err error
res,err := f()
```

[建议] 函数返回值大于 3 个时，建议通过 struct 进行包装。

[建议] 函数参数不建议超过 3 个，大于 3 个时建议通过 struct 进行包装。

控制结构

[强制] 禁止使用 goto。

[强制 gosimple] 当一个表达式为 bool 类型时，应该使用 expr 或 !expr 判断，禁止使用 == 或 != 与 true / false 比较。

[强制 nestif] if 嵌套深度不大于 5。

方法

[强制 revive] receiver 的命名要保持一致，如果你在一个方法中将接收器命名为 “c”，那么在其他方法中不要把它命名为 “cl”。

[强制] receiver 的名字要尽量简短并有意义，禁止使用 this、self 等。

复制代码

```
func (c Client) done() error {
 // ...
}
func (cl Client) call() error {
 // ...
}
```

注释

Go 提供 C 风格的注释。有 /**/ 的块注释和 // 的单行注释两种注释风格。注释主要有下面几个用处。

1. 注释不仅仅可以提供具体的逻辑细节，还可以提供代码背后的意图和决策。
2. 帮助澄清一些晦涩的参数或返回值的含义。一般来说，我们会尽量找到一种方法让参数或返回值的名字本身就是清晰的。但是当它是标准库的一部分时，或者在你无法更改的第三方库中，一个清晰的注释会非常有用。
3. 强调某一个重要的功能。例如，提醒开发者修改了这一处代码必须连带修改另一处代码。

总之，好的注释给我们讲解了 what、how、why，方便后续的代码维护。

[强制] 无用注释直接删除，无用的代码不应该注释而应该直接删除。即使日后需要，我们也可以通过 Git 快速找到。

[强制] 紧跟在代码之后的注释，使用 //。

[强制] 统一使用中文注释，中英文字符之间严格使用空格分隔。


```
// 从 Redis 中批量读取属性，对于没有读取到的 id ，  记录到一个数组里面，准备从 DB 中读取
```

[强制] 注释不需要额外的格式，例如星号横幅。

[强制] 包、函数、方法和类型的注释说明都是一个完整的句子，以被描述的对象为主语开头。Go 源码中都是这样的。

示例如下：

```
// queueForIdleConn queues w to receive the next idle connection for w.cm.
// As an optimization hint to the caller, queueForIdleConn reports whether
// it successfully delivered an already-idle connection.
func (t *Transport) queueForIdleConn(w *wantConn) (delivered bool)
```

[强制] Go 语言提供了文档注释工具 go doc，可以生成注释和导出函数的文档。文档注释的写法可以参考文稿中的链接。

[强制 godot] 注释最后应该以句号结尾。

[建议] 当某个部分等待完成时，可用 `TODO:` 开头的注释来提醒维护人员。

[建议] 大部分情况下使用行注释。块注释主要用在包的注释上，不过块注释在表达式中或禁用大量代码时很有用。

[建议] 当某个部分存在已知问题需要修复或改进时，可用 `FIXME:` 开头的注释来提醒维护人员。

[建议] 需要特别说明某个问题时，可用 `NOTE:` 开头的注释。

结构体

[强制] 不要将 Context 成员添加到 Struct 类型中。

高效

[强制] Map 在初始化时需要指定长度`make(map[T1]T2, hint)`。

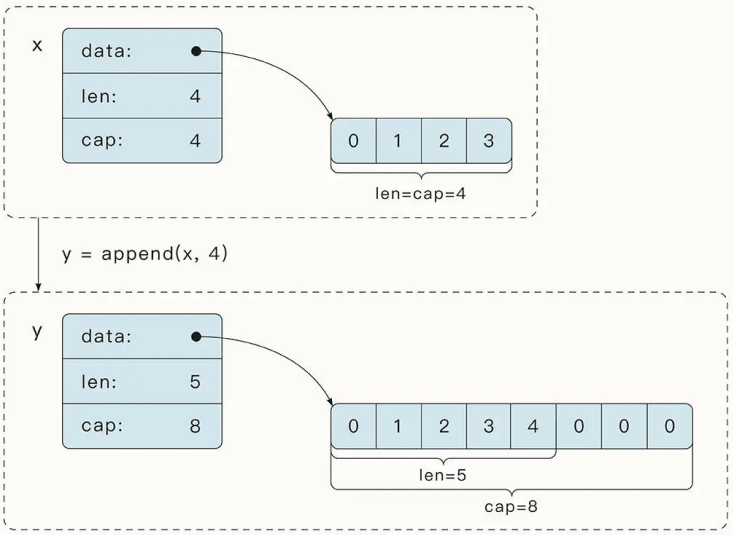
[强制] Slice 在初始化时需要指定长度和容量`make([]T, length, capacity)`。

我们来看下下面这段程序，它的目的是往切片中循环添加元素。

复制代码

```
func createSlice(n int) (slice []string) {
   for i := 0; i < n; i++ {
      slice = append(slice, "I", "love", "go")
   }
   return slice
}
```

从功能上来看，这段代码没有问题。但是，这种写法忽略了一个事实，如下图所示，往切片中添加数据时，切片会自动扩容，Go 运行时会创建新的内存空间并执行拷贝。



极客时间

自动扩容显然是有成本的。在循环操作中执行这样的代码会放大性能损失，减慢程序的运行速度。性能损失的对比可参考这篇文章。我们可以改写一下上面这段程序，在初始化时指定合

适的切片容量：


```go
func createSlice(n int) []string {
    slice := make([]string, 0, n*3)
    for i := 0; i < n; i++ {
        slice = append(slice, "I", "love", "go")
    }
    return slice
}
```

这段代码在一开始就指定了需要的容量，最大程度避免了内存的浪费。同时，运行时不需要再执行自动扩容操作，加速了程序的运行。

[强制] time.After() 在某些情况下会发生泄露，替换为使用 Timer。

[强制] 数字与字符串转换时，使用 strconv，而不是 fmt。

[强制] 读写磁盘时，使用读写 buffer。

[建议] 谨慎使用 Slice 的截断操作和 append 操作，除非你知道下面的代码输出什么：

复制代码

```go
x := []int{1, 2, 3, 4}
y := x[:2]
fmt.Println(cap(x), cap(y))
y = append(y, 30)
fmt.Println("x:", x)
fmt.Println("y:", y)
```

[建议] 任何书写的协程，都需要明确协程什么时候退出。

[建议] 热点代码中，内存分配复用内存可以使用 sync.Pool 提速。

[建议] 将频繁的字符串拼接操作（+=），替换为 **StringBuffer 或 StringBuilder。**

[建议] 使用正则表达式重复匹配时，利用 Compile 提前编译提速。

[建议] 当程序严重依赖 Map 时，Map 的 Key 使用 int 而不是 string 将提速。

[建议] 多读少写的场景，使用读写锁而不是写锁将提速。

健壮性

[强制] 除非出现不可恢复的程序错误，否则不要使用 panic 来处理常规错误，使用 error 和多返回值。

[强制 revive] 错误信息不应该首字母大写（除专有名词和缩写词外），也不应该以标点符号结束。因为错误信息通常在其他上下文中被打印。

[强制 errcheck] 不要使用 _ 变量来丢弃 error。如果函数返回 error，应该强制检查。

[建议] 在处理错误时，如果我们逐层返回相同的错误，那么在最后日志打印时，我们并不知道代码中间的执行路径。例如找不到文件时打印的`No such file or directory`，这会减慢我们排查问题的速度。因此，在中间处理 err 时，需要使用 fmt.Errorf 或第三方包给错误添加额外的上下文信息。像下面这个例子，在 fmt.Errorf 中，除了实际报错的信息，还加上了授权错误信息`authenticate failed`：

复制代码

```
func AuthenticateRequest(r *Request) error {
        err := authenticate(r.User)
        if err != nil {
                return fmt.Errorf("authenticate failed: %v", err)
        }
        return nil
}
```

当有多个错误需要处理时，可以考虑将 fmt.Errorf 放入 defer 中：

复制代码

```
func DoSomeThings(val1 int, val2 string) (_ string, err error) {
    defer func() {
        if err != nil {
            err = fmt.Errorf("in DoSomeThings: %w", err)
        }
    }()
    val3, err := doThing1(val1)
```

```
    if err != nil {
        return "", err
    }
    val4, err := doThing2(val2)
    if err != nil {
        return "", err
    }
    return doThing3(val3, val4)
}
```


[强制] 利用 recover 捕获 panic 时，需要由 defer 函数直接调用。

例如，下面例子中的 panic 是可以被捕获的：

复制代码

```
package main

import "fmt"

func printRecover() {
    r := recover()
    fmt.Println("Recovered:", r)
}

func main() {
    defer printRecover()

    panic("OMG!")
}
```

但是下面这个例子中的 panic 却不能被捕获：

复制代码

```
package main

import "fmt"

func printRecover() {
  r := recover()
  fmt.Println("Recovered:", r)
}

func main() {
  defer func() {
    printRecover()
```

```
  }()

  panic("OMG!")
}
```


[强制] 不用重复使用 recover，只需要在每一个协程的最上层函数拦截即可。recover 只能够捕获当前协程，而不能跨协程捕获 panic，下例中的 panic 就是无法被捕获的。

复制代码

```
package main

import "fmt"

func printRecover() {
  r := recover()
  fmt.Println("Recovered:", r)
}

func main() {
  defer printRecover()
  go func() {
    panic("OMG!")
  }()
  // ...
}
```

[强制] 有些特殊的错误是 recover 不住的，例如 Map 的并发读写冲突。这种错误可以通过 race 工具来检查。

扩展性

[建议] 利用接口实现扩展性。接口特别适用于访问外部组件的情况，例如访问数据库、访问下游服务。另外，接口可以方便我们进行功能测试。关于接口的最佳实践，需要单独论述。

[建议] 使用功能选项模式对一些公共 API 的构造函数进行扩展，大量第三方库例如 gomicro、zap 等都使用了这种策略。

复制代码

```
db.Open(addr, db.DefaultCache, zap.NewNop())
可以替换为=>
db.Open(
addr,
```

```
db.WithCache(false),
db.WithLogger(log),
)
```


工具

要人工来保证团队成员遵守了上述的编程规范并不是一件容易的事情。因此，我们有许多静态的和动态的代码分析工具帮助团队识别代码规范的错误，甚至可以发现一些代码的 bug。

golangci-lint

golangci-lint 是当前大多数公司采用的静态代码分析工具，词语 Linter 指的是一种分析源代码以此标记编程错误、代码缺陷、风格错误的工具。

而 golangci-lint 是集合多种 Linter 的工具。要查看支持的 Linter 列表以及启用 / 禁用了哪些 Linter，可以使用下面的命令：

复制代码

```
golangci-lint help linters
```

Go 语言定义了实现 Linter 的 API，它还提供了 golint 工具，用于集成了几种常见的 Linter。在源码中，我们可以查看怎么在标准库中实现典型的 Linter。

Linter 的实现原理是静态扫描代码的 AST（抽象语法树），Linter 的标准化意味着我们可以灵活实现自己的 Linters。不过 golangci-lint 里面其实已经集成了包括 golint 在内的总多 Linter，并且有灵活的配置能力。所以在自己写 Linter 之前，建议先了解 golangci-lint 现有的能力。

在大型项目中刚开始使用 golang-lint 会出现大量的错误，这种情况下我们只希望扫描增量的代码。如下所示，可以通过在golangci-lint 配置文件中调整 new-from-rev 参数，配置以当前基准分支为基础实现增量扫描

复制代码

```
linters:
 enable-all: true
issues:
 new-from-rev: master
```

Pre-Commit

在代码通过 Git Commit 提交到代码仓库之前，git 提供了一种 pre-commit 的 hook 能力，用于执行一些前置脚本。在脚本中加入检查的代码，就可以在本地拦截住一些不符合规范的代码，避免频繁触发 CI 或者浪费时间。pre-commit 的配置和使用方法，可以参考TiDB。

并发检测 race

Go 1.1 提供了强大的检查工具 race 来排查数据争用问题。race 可以用在多个 Go 指令中，一旦检测器在程序中找到数据争用，就会打印报告。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。可以在编译时和运行时执行 race，方法如下：

复制代码

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

在下面这个例子中， 运行中加入 race 检查后直接报错。从报错后输出的栈帧信息中，我们能看出具体发生并发冲突的位置。

复制代码

```
» 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
```

第四行 Read at 表明读取发生在 2_race.go 文件的第 5 行，而第七行 Previous write 表明前一个写入也发生在 2_race.go 文件的第 5 行。这样我们就可以非常快速地定位数据争用问题了。

竞争检测的成本因程序而异。对于典型的程序，内存使用量可能增加 5~10 倍，执行时间会增加 2~20 倍。同时，竞争检测器会为当前每个 defer 和 recover 语句额外分配 8 字节。在 Goroutine 退出前，这些额外分配的字节不会被回收。这意味着，如果有一个长期运行的

Goroutine，而且定期有 defer 和 recover 调用，那么程序内存的使用量可能无限增长。（这些内存分配不会显示到 runtime.ReadMemStats 或 runtime / pprof 的输出。）

覆盖率

一般我们会使用代码覆盖率来判断代码书写的质量，识别无效代码。go tool cover 是 go 语言提供的识别代码覆盖率的工具，在后面的课程中还会详细介绍。

总结

代码规范可以助力团队协作、帮助我们写出更加简洁、高效、健壮和可扩展的代码。这节课，我列出了一套 Go 编程规范的最佳实践，并通过 golangci-lint 等工具对不规范甚至是错误的代码进行了强制检查，保证了代码质量。在后面的开发中，我们将严格按照这个规范编写代码。

考虑到这个规范的通用性，我将这个规范进行了开源：Go 代码规范，你可以直接将其作为你团队的规范，如果你有更好的建议，也可以提交 PR。

课后题

学完这节课，我也给你留一道思考题吧。

Go 标准库 Errors 有一个方法叫做 New， 为什么 Go 语言的设计者不将它命名为 ErrorNew 呢？

欢迎你在留言区留下自己思考的结果，也可以把这节课分享给对这个话题感兴趣的同事和朋友，我们下节课再见！

上一篇 14 | 课实面动：爬虫项目需求分析与架构设计
下一篇 16 | 网络爬虫：一次HTTP请求的魔幻旅途

精选留言 (4)

写留言

Realm 酥

2022-11-12 来自浙江

errors. ErrorNew()，按老师的建议，包里面有个errors名字了，方法里面也出现这个名字，不够简洁.

3

江楠大盗

2022-11-14 来自北京

函数小结，为什么避免使用init函数？为什么禁止return裸返回？

2

陈卧虫 魔

2022-11-12 来自浙江

Errors包名已经表达了错误的含义，用errors.New就可以表明是新建错误，errors.ErrorNew就显得很多余

1

风铃

2022-11-12 来自浙江

又是干货满满的一天

1