19 性能优化：Go 语言如何进行代码检查和优化？

在上节课中，我为你留了一个小作业：在运行 go test 命令时，使用 -benchmem 这个 Flag 进行内存统计。该作业的答案比较简单，命令如下所示：

```
→ go test -bench=. -benchmem  ./ch18
```

运行这一命令就可以查看内存统计的结果了。这种通过 -benchmem 查看内存的方法**适用于所有的基准测试用例**。

今天要讲的内容是 Go 语言的代码检查和优化，下面我们开始本讲内容的讲解。

在项目开发中，**保证代码质量和性能的手段不只有单元测试和基准测试，还有代码规范检查和性能优化**。

- **代码规范检查**是对单元测试的一种补充，它可以从非业务的层面检查你的代码是否还有优化的空间，比如变量是否被使用、是否是死代码等等。
- **性能优化**是通过基准测试来衡量的，这样我们才知道优化部分是否真的提升了程序的性能。

代码规范检查

什么是代码规范检查

代码规范检查，顾名思义，是从 Go 语言层面出发，依据 Go 语言的规范，对你写的代码进行的**静态扫描检查**，这种检查和你的业务无关。

比如你定义了个常量，从未使用过，虽然对代码运行并没有造成什么影响，但是这个常量是可以删除的，代码如下所示：

ch19/main.go

```
const name = "飞雪无情"

func main() {

}
```

示例中的常量 name 其实并没有使用，所以为了节省内存你可以删除它，这种**未使用常量**的情况就可以通过代码规范检查检测出来。

再比如，你调用了一个函数，该函数返回了一个 error，但是你并没有对该 error 做判断，这种情况下，程序也可以正常编译运行。但是代码写得不严谨，因为返回的 error 被我们忽略了。代码如下所示：

ch19/main.go

```
func main() {
   os.Mkdir("tmp",0666)
}
```

示例代码中，Mkdir 函数是有返回 error 的，但是你并没有对返回的 error 做判断，这种情况下，哪怕创建目录失败，你也不知道，因为错误被你忽略了。如果你使用代码规范检查，这类潜在的问题也会被检测出来。

以上两个例子可以帮你理解什么是代码规范检查、它有什么用。除了这两种情况，还有拼写问题、死代码、代码简化检测、命名中带下划线、冗余代码等，都可以使用代码规范检查检测出来。

golangci-lint

要想对代码进行检查，则需要对代码进行扫描，静态分析写的代码是否存在规范问题。

> 小提示：静态代码分析是不会运行代码的。

可用于 Go 语言代码分析的工具有很多，比如 golint、gofmt、misspell 等，如果一一引用配置，就会比较烦琐，所以通常我们不会单独地使用它们，而是使用 golangci-lint。

golangci-lint 是一个**集成工具**，它集成了很多静态代码分析工具，便于我们使用。通过配置这一工具，我们可以很灵活地启用需要的代码规范检查。

如果要使用 golangci-lint，首先需要安装。因为 golangci-lint 本身就是 Go 语言编写的，所以我们可以从源代码安装它，打开终端，输入如下命令即可安装。

```
➜ go get github.com/golangci/golangci-lint/cmd/golangci-lint@v1.32.2
```

使用这一命令安装的是 v1.32.2 版本的 golangci-lint，安装完成后，在终端输入如下命令，检测是否安装成功。

```
➜ golangci-lint version
```

```
golangci-lint has version v1.32.2
```

> 小提示：在 MacOS 下也可以使用 brew 来安装 golangci-lint。

好了，安装成功 golangci-lint 后，就可以使用它进行代码检查了，我以上面示例中的常量 name 和 Mkdir 函数为例，演示 golangci-lint 的使用。在终端输入如下命令回车：

```
➜ golangci-lint run ch19/
```

这一示例表示要检测目录中 ch19 下的代码，运行后可以看到如下输出结果。

```
ch19/main.go:5:7: `name` is unused (deadcode)
const name = "飞雪无情"
      ^
ch19/main.go:8:10: Error return value of `os.Mkdir` is not checked (errchecl
        os.Mkdir("tmp",0666)
```

通过代码检测结果可以看到，我上一小节提到的两个代码规范问题都被检测出来了。检测出问题后，你就可以修复它们，让代码更加符合规范。

golangci-lint 配置

golangci-lint 的配置比较灵活，比如你可以自定义要启用哪些 linter。golangci-lint 默认启用的 linter，包括这些：

```
deadcode - 死代码检查
errcheck - 返回错误是否使用检查
gosimple - 检查代码是否可以简化
govet - 代码可疑检查，比如格式化字符串和类型不一致
ineffassign - 检查是否有未使用的代码
staticcheck - 静态分析检查
structcheck - 查找未使用的结构体字段
typecheck - 类型检查
unused - 未使用代码检查
varcheck - 未使用的全局变量和常量检查
```

> 小提示：golangci-lint 支持的更多 linter，可以在终端中输入 golangci-lint linters 命令查看，并且可以看到每个 linter 的说明。

如果要修改默认启用的 linter，就需要对 golangci-lint 进行配置。即在项目根目录下新建一个名字为 .golangci.yml 的文件，这就是 golangci-lint 的配置文件。在运行代码规范检查的时候，golangci-lint 会自动使用它。假设我只启用 unused 检查，可以这样配置：

.golangci.yml

```
linters:
  disable-all: true
  enable:
    - unused
```

在团队多人协作开发中，有一个固定的 golangci-lint 版本是非常重要的，这样大家就可以**基于同样的标准检查代码**。要配置 golangci-lint 使用的版本也比较简单，在配置文件中添加如下代码即可：

```
service:
  golangci-lint-version: 1.32.2 # use the fixed version to not introduce ne
```

此外，你还可以针对每个启用的 linter 进行配置，比如要设置拼写检测的语言为 US，可以使用如下代码设置：

```
linters-settings:
  misspell:
    locale: US
```

golangci-lint 的配置比较多，你自己可以灵活配置。关于 golangci-lint 的更多配置可以参考官方文档，这里我给出一个常用的配置，代码如下：

.golangci.yml

```
linters-settings:
  golint:
    min-confidence: 0
  misspell:
    locale: US
```

```
linters:
  disable-all: true
  enable:
    - typecheck
    - goimports
    - misspell
    - govet
    - golint
    - ineffassign
    - gosimple
    - deadcode
    - structcheck
    - unused
    - errcheck
service:
  golangci-lint-version: 1.32.2 # use the fixed version to not introduce new
```

集成 golangci-lint 到 CI

代码检查一定要集成到 CI 流程中，效果才会更好，这样开发者提交代码的时候，CI 就会自动检查代码，及时发现问题并进行修正。

不管你是使用 Jenkins，还是 Gitlab CI，或者 Github Action，都可以通过**Makefile**的方式运行 golangci-lint。现在我在项目根目录下创建一个 Makefile 文件，并添加如下代码：

Makefile

```
getdeps:
	@mkdir -p ${GOPATH}/bin
	@which golangci-lint 1>/dev/null || (echo "Installing golangci-lint" && 
lint:
	@echo "Running $@ check"
	@GO111MODULE=on ${GOPATH}/bin/golangci-lint cache clean
	@GO111MODULE=on ${GOPATH}/bin/golangci-lint run --timeout=5m --config ./
verifiers: getdeps lint
```

> 小提示：关于 Makefile 的知识可以网上搜索学习一下，比较简单，这里不再进行讲述。

好了，现在你就可以把如下命令添加到你的 CI 中了，它可以帮你自动安装 golangci-lint，并检查你的代码。

```
make verifiers
```

性能优化

性能优化的目的是让程序更好、更快地运行，但是它不是必要的，这一点一定要记住。所以在程序开始的时候，你不必刻意追求性能优化，先大胆地写你的代码就好了，**写正确的代码是性能优化的前提**。

堆分配还是栈

在比较古老的 C 语言中，内存分配是手动申请的，内存释放也需要手动完成。

- 手动控制有一个很大的**好处**就是你需要多少就申请多少，可以最大化地**利用内存**；
- 但是这种方式也有一个明显的**缺点**，就是如果忘记释放内存，就会导致**内存泄漏**。

所以，为了让程序员更好地专注于业务代码的实现，Go 语言增加了垃圾回收机制，自动地回收不再使用的内存。

Go 语言有两部分内存空间：**栈内存**和**堆内存**。

- **栈内存**由编译器自动分配和释放，开发者无法控制。栈内存一般存储函数中的局部变量、参数等，函数创建的时候，这些内存会被自动创建；函数返回的时候，这些内存会被自动释放。
- **堆内存**的生命周期比栈内存要长，如果函数返回的值还会在其他地方使用，那么这个值就会被编译器自动分配到堆上。堆内存相比栈内存来说，不能自动被编译器释放，只能通过垃圾回收器才能释放，所以栈内存效率会很高。

逃逸分析

既然栈内存的效率更高，肯定是优先使用栈内存。那么 Go 语言是如何判断一个变量应该分配到堆上还是栈上的呢？这就需要**逃逸分析**了。下面我通过一个示例来讲解逃逸分析，代码如下：

ch19/main.go

```
func newString() *string{
   s:=new(string)
   *s = "飞雪无情"
   return s
}
```

在这个示例中：

- 通过 new 函数申请了一块内存；
- 然后把它赋值给了指针变量 s；
- 最后通过 return 关键字返回。

> 小提示：以上 newString 函数是没有意义的，这里只是为了方便演示。

现在我通过逃逸分析来看下是否发生了逃逸，命令如下：

```
→ go build -gcflags="-m -l" ./ch19/main.go
# command-line-arguments
ch19/main.go:16:8: new(string) escapes to heap
```

在这一命令中，-m 表示打印出逃逸分析信息，-l 表示禁止内联，可以更好地观察逃逸。从以上输出结果可以看到，发生了逃逸，**也就是说指针作为函数返回值的时候，一定会发生逃逸**。

逃逸到堆内存的变量不能马上被回收，只能通过垃圾回收标记清除，增加了垃圾回收的压力，所以要尽可能地避免逃逸，让变量分配在栈内存上，这样函数返回时就可以回收资源，提升效率。

下面我对 newString 函数进行了避免逃逸的优化，优化后的函数代码如下：

ch19/main.go

```
func newString() string{
   s:=new(string)
   *s = "飞雪无情"
   return *s
}
```

再次通过命令查看以上代码的逃逸分析，命令如下：

```
➜ go build -gcflags="-m -l" ./ch19/main.go
# command-line-arguments
ch19/main.go:14:8: new(string) does not escape
```

通过分析结果可以看到，虽然还是声明了指针变量 s，但是函数返回的并不是指针，所以没有发生逃逸。

这就是关于指针作为函数返回逃逸的例子，那么是不是不使用指针就不会发生逃逸了呢？下面看个例子，代码如下：

```
fmt.Println("飞雪无情")
```

同样运行逃逸分析，你会看到如下结果：

```
➜ go build -gcflags="-m -l" ./ch19/main.go
# command-line-arguments
ch19/main.go:13:13: ... argument does not escape
ch19/main.go:13:14: "飞雪无情" escapes to heap
ch19/main.go:17:8: new(string) does not escape
```

观察这一结果，你会发现「飞雪无情」这个字符串逃逸到了堆上，这是因为「飞雪无情」这个字符串被已经逃逸的指针变量引用，所以它也跟着逃逸了，引用代码如下：

```
func (p *pp) printArg(arg interface{}, verb rune) {
   p.arg = arg
   //省略其他无关代码
}
```

所以**被已经逃逸的指针引用的变量也会发生逃逸**。

Go 语言中有 3 个比较特殊的类型，它们是 slice、map 和 chan，被这三种类型引用的指针也会发生逃逸，看个这样的例子：

ch19/main.go

```
func main() {
   m:=map[int]*string{}
   s:="飞雪无情"
   m[0] = &s
}
```

同样运行逃逸分析，你看到的结果是：

```
➜  gotour go build -gcflags="-m -l" ./ch19/main.go
# command-line-arguments
ch19/main.go:16:2: moved to heap: s
ch19/main.go:15:20: map[int]*string literal does not escape
```

从这一结果可以看到，变量 m 没有逃逸，反而被变量 m 引用的变量 s 逃逸到了堆上。**所以被map、slice 和 chan 这三种类型引用的指针一定会发生逃逸的**。

逃逸分析是判断变量是分配在堆上还是栈上的一种方法，在实际的项目中要尽可能避免逃逸，这样就不会被 GC 拖慢速度，从而提升效率。

> 小技巧：从逃逸分析来看，指针虽然可以减少内存的拷贝，但它同样会引起逃逸，所以要根据实际情况选择是否使用指针。

优化技巧

通过前面小节的介绍，相信你已经了解了栈内存和堆内存，以及变量什么时候会逃逸，那么在优化的时候思路就比较清晰了，因为都是基于以上原理进行的。下面我总结几个优化的小技巧：

第 1 个需要介绍的技巧是尽可能避免逃逸，因为栈内存效率更高，还不用 GC。比如小对象的传参，array 要比 slice 效果好。

如果避免不了逃逸，还是在堆上分配了内存，那么对于频繁的内存申请操作，我们要学会重用内存，比如使用 sync.Pool，这是**第 2 个**技巧。

第 3 个技巧就是选用合适的算法，达到高性能的目的，比如空间换时间。

> 小提示：性能优化的时候，要结合基准测试，来验证自己的优化是否有提升。

以上是基于 GO 语言的内存管理机制总结出的 3 个方向的技巧，基于这 3 个大方向基本上可以优化出你想要的效果。除此之外，还有一些小技巧，比如要尽可能避免使用锁、并发加锁的范围要尽可能小、使用 StringBuilder 做 string 和 [ ] byte 之间的转换、defer 嵌套不要太多等等。

最后推荐一个 Go 语言自带的性能剖析的工具 pprof，通过它你可以查看 CPU 分析、内存分析、阻塞分析、互斥锁分析，它的使用不是太复杂，你可以搜索下它的使用教程，这里就不展开介绍。

总结

这节课主要介绍了代码规范检查和性能优化两部分内容，其中代码规范检查是从工具使用的角度讲解，而性能优化可能涉及的点太多，所以是从原理的角度讲解，你明白了原理，就能更好地优化你的代码。

我认为是否进行性能优化取决于两点：**业务需求和自我驱动**。所以不要刻意地去做性能优化，尤其是不要提前做，先保证代码正确并上线，然后再根据业务需要，决定是否进行优化以及花多少时间优化。自我驱动其实是一种编码能力的体现，比如有经验的开发者在编码的时候，潜意识地就避免了逃逸，减少了内存拷贝，在高并发的场景中设计了低延迟的架构。

最后给你留个作业，把 golangci-lint 引入自己的项目吧，相信你的付出会有回报的。

下一讲我将介绍“协作开发：模块化管理为什么能够提升研发效能”，记得来听课！

上一页 下一页