

25 | 更多的测试手法

2018-10-08 郝林



25 | 更多的测试...

朗读人：黄洲君
14'29" | 6.63M

0:00 / 14:29

在前面的文章中，我们一起学习了 Go 程序测试的基础知识和基本测试手法。这主要包括了 Go 程序测试的基本规则和主要流程、`testing.T`类型和`testing.B`类型的常用方法、`go test`命令的基本使用方式、常规测试结果的解读等等。

在本篇文章，我会继续为你讲解更多更高级的测试方法。这会涉及`testing`包中更多的 API、`go test`命令支持的，更多标记更加复杂的测试结果，以及测试覆盖度分析等等。

续接前文。我在前面提到了`go test`命令的标记`-cpu`，它是用来设置测试执行最大 P 数量的列表的。

复习一下，我在讲 go 语句的时候说过，这里的 P 是 processor 的缩写，每个 processor 都是一个可以承载若干个 G，且能够使这些 G 适时地与 M 进行对接并得到真正运行的中介。

正是由于 P 的存在，G 和 M 才可以呈现出多对多的关系，并能够及时、灵活地进行组合和分离。

这里的 G 就是 goroutine 的缩写，可以被理解为 Go 语言自己实现的用户级线程。M 即为 machine 的缩写，代表着系统级线程，或者说操作系统内核级别的线程。

Go 语言并发编程模型中的 P，正是 goroutine 的数量能够数十万计的关键所在。P 的数量意味着 Go 程序背后的运行时系统中，会有多少个用于承载可运行的 G 的队列存在。每一个队列都相当于一条流水线，它会源源不断地把可运行的 G 输送给空闲的 M，并使这两者对接。

一旦对接完成，被对接的 G 就真正地运行在操作系统的内核级线程之上了。每条流水线之间虽然会有联系，但都是独立运作的。

因此，最大 P 数量就代表着 Go 语言运行时系统同时运行 goroutine 的能力，也可以被视为其中逻辑 CPU 的最大个数。

而 `go test` 命令的 `-cpu` 标记正是用于设置这个最大个数的。也许你已经知道，在默认情况下，最大 P 数量就等于当前计算机 CPU 核心的实际数量。

当然了，前者也可以大于或者小于后者，如此可以在一定程度上模拟拥有不同的 CPU 核心数的计算机。

所以，也可以说，使用 `-cpu` 标记可以模拟被测程序在计算能力不同计算机中的表现。

现在，你已经知道了 `-cpu` 标记的用途及其背后的含义。那么它的具体用法，以及对 `go test` 命令的影响你是否也清楚呢？

我们今天的问题是：怎样设置 `-cpu` 标记的值，以及它会对测试流程产生什么样的影响？

这里的典型回答是：

标记 `-cpu` 的值应该是一个正整数的列表，该列表的表现形式为：以英文半角逗号分隔的多个整数字面量，比如 `1, 2, 4`。

针对于此值中的每一个正整数，`go test` 命令都会先设置最大 P 数量为该数，然后再执行测试函数。

如果测试函数有多个，那么 `go test` 命令会依照此方式逐个执行。

以 `1, 2, 4` 为例，`go test` 命令会先以 `1, 2, 4` 为最大 P 数量分别去执行第一个测试函数，之后再用同样的方式执行第二个测试函数，以此类推。

问题解析

实际上，不论我们是否追加了 `-cpu` 标记，`go test` 命令执行测试函数时流程都是相同的，只不过具体执行步骤会略有不同。

`go test`命令在进行准备工作的时候会读取`-cpu`标记的值，并把它转换为一个以`int`为元素类型的切片，我们也可以称它为逻辑 CPU 切片。

如果该命令发现我们并没有追加这个标记，那么就会让逻辑 CPU 切片只包含一个元素值，即最大 P 数量的默认值，也就是当前计算机 CPU 核心的实际数量。

在准备执行某个测试函数的时候，无论该函数是功能测试函数，还是性能测试函数，`go test`命令都会迭代逻辑 CPU 切片，并且在每次迭代时，先依据当前的元素值设置最大 P 数量，然后再去执行测试函数。

注意，对于性能测试函数来说，这里可能不只执行了一次。你还记得测试函数的执行时间上限，以及那个由`b.N`代表的被测程序的执行次数吗？

如果你忘了，那么可以再复习一下上篇文章中的第二个扩展问题。概括来讲，`go test`命令每一次对性能测试函数的执行，都是一个探索的过程。它会在测试函数的执行时间上限不变的前提下，尝试找到被测程序的最大执行次数。


在这个过程中，性能测试函数可能会被执行多次。为了以后描述方便，我们把这样一个探索的过程称为：对性能测试函数的一次探索式执行，这其中包含了对该函数的若干次执行，当然，肯定也包含了对被测程序更多次的执行。

说到多次执行测试函数，我们就不得不提及另外一个标记，即`-count`。`-count`标记是专门用于重复执行测试函数的。它的值必须大于或等于0，并且默认值为1。

如果我们在运行`go test`命令的时候追加了`-count 5`，那么对于每一个测试函数，命令都会在预设的不同条件下（比如不同的最大 P 数量下）分别重复执行五次。


如果我们把前文所述的`-cpu`标记、`-count`标记，以及探索式执行联合起来看，就可以用一个公式来描述单个性能测试函数，在`go test`命令的一次运行过程中的执行次数，即：

1 性能测试函数的执行次数 = `-cpu` 标记的值中正整数的个数` x `-count` 标记的值` x 探索式执行中测试函数的实际执行次数

 复制代码

对于功能测试函数来说，这个公式会更加简单一些，即：

1 功能测试函数的执行次数 = `-cpu` 标记的值中正整数的个数` x `-count` 标记的值`

 复制代码

看完了这两个公式，我想，你也许遇到过这种情况，在对 Go 程序执行某种自动化测试的过程中，测试日志会显得特别多，而且好多都是重复的。

这时，我们首先就应该想到，上面这些导致测试函数多次执行的标记和流程。我们往往需要检查这些标记的使用是否合理、日志记录是否有必要等等，从而对测试日志进行精简。

比如，对于功能测试函数来说，我们通常没有必要重复执行它，即使是在不同的最大 P 数量下也是如此。注意，这里所说的重复执行指的是，在被测程序的输入（比如说被测函数的参数值）相同情况下的多次执行。

有些时候，在输入完全相同的情况下，被测程序会因其他外部环境的不同，而表现出不同的行为。这时我们需要考虑的往往应该是：这个程序在设计上是否合理，而不是通过重复执行测试来检测风险。

还有些时候，我们的程序会无法避免地依赖一些外部环境，比如数据库或者其他服务。这时，我们依然不应该让测试的反复执行成为检测手段，而应该在测试中通过仿造（mock）外部环境，来规避掉它们的不确定性。

其实，单元测试的意思就是：对单一的功能模块进行边界清晰的测试，并且不掺杂任何对外部环境的检测。这也是“单元”二字要表达的主要含义。

正好相反，对于性能测试函数来说，我们常常需要反复地执行，并以此试图抹平当时的计算资源调度的细微差别对被测程序性能的影响。通过-cpu标记，我们还能够模拟被测程序在计算能力不同计算机中的性能表现。

不过要注意，这里设置的最大 P 数量，最好不要超过当前计算机 CPU 核心的实际数量。因为一旦超出计算机实际的并行处理能力，Go 程序在性能上就无法再得到显著地提升了。

这就像一个漏斗，不论我们怎样灌水，水的漏出速度总是有限的。更何况，为了管理过多的 P，Go 语言运行时系统还会耗费额外的计算资源。

显然，上述模拟得出的程序性能一定是不准确的。不过，这或多或少可以作为一个参考，因为，这样模拟出的性能一般都会低于程序在计算环境中的实际性能。

好了，关于-cpu标记，以及由此引出的-count标记和测试函数多次执行的问题，我们就先聊到这里。不过，为了让你再巩固一下前面的知识，我现在给出一段测试结果：

复制代码

```
1 pkg: puzzlers/article21/q1
2 BenchmarkGetPrimesWith100-2      10000000      218 ns/op
3 BenchmarkGetPrimesWith100-2      10000000      215 ns/op
4 BenchmarkGetPrimesWith100-4      10000000      215 ns/op
5 BenchmarkGetPrimesWith100-4      10000000      216 ns/op
6 BenchmarkGetPrimesWith10000-2    50000         31523 ns/op
7 BenchmarkGetPrimesWith10000-2    50000         32372 ns/op
8 BenchmarkGetPrimesWith10000-4    50000         32065 ns/op
9 BenchmarkGetPrimesWith10000-4    50000         31936 ns/op
```

10	BenchmarkGetPrimesWith1000000-2	300	4085799	ns/op
11	BenchmarkGetPrimesWith1000000-2	300	4121975	ns/op
12	BenchmarkGetPrimesWith1000000-4	300	4112283	ns/op
13	BenchmarkGetPrimesWith1000000-4	300	4086174	ns/op

现在，我希望让你反推一下，我在运行`go test`命令时追加的`-cpu`标记和`-count`标记的值都是什么。反推之后，你可以用实验的方式进行验证。

知识扩展

问题 1：`-parallel`标记的作用是什么？

我们在运行`go test`命令的时候，可以追加标记`-parallel`，该标记的作用是：设置同一个被测代码包中的功能测试函数的最大并发执行数。该标记的默认值是测试运行时的最大 P 数量（这可以通过调用表达式`runtime.GOMAXPROCS(0)`获得）。

我在上篇文章中已经说过，对于功能测试，为了加快测试速度，命令通常会并发地测试多个被测代码包。

但是，在默认情况下，对于同一个被测代码包中的多个功能测试函数，命令会串行地执行它们。除非我们在一些功能测试函数中显式地调用`t.Parallel`方法。

这个时候，这些包含了`t.Parallel`方法调用的功能测试函数就会被`go test`命令并发地执行，而并发执行的最大数量正是由`-parallel`标记值决定的。不过要注意，同一个功能测试函数的多次执行之间一定是串行的。

你可以运行命令`go test -v puzzlers/article21/q2`或者`go test -count=2 -v puzzlers/article21/q2`，查看测试结果，然后仔细地体会一下。

最后，强调一下，`-parallel`标记对性能测试是无效的。当然了，对于性能测试来说，也是可以并发进行的，不过机制上会有所不同。

概括地讲，这涉及了`b.RunParallel`方法、`b.SetParallelism`方法和`-cpu`标记的联合运用。如果想进一步了解，你可以查看`testing`代码包的文档。

（<https://golang.google.cn/pkg/testing>）

问题 2：性能测试函数中的计时器是做什么用的？

如果你看过`testing`包的文档，那么很可能会发现其中的`testing.B`类型有这么几个指针方法：`StartTimer`、`StopTimer`和`ResetTimer`。这些方法都是用于操作当前的性能测试函数专属的计时器的。

所谓的计时器，是一个逻辑上的概念，它其实是`testing.B`类型中一些字段的统称。这些字段用于记录：当前测试函数在当次执行过程中耗费的时间、分配的堆内存的字节数以及分配次数。

我在下面会以测试函数的执行时间为例，来说明此计时器的用法。不过，你需要知道的是，这三个方法在开始记录、停止记录或重新记录执行时间的同时，也会对堆内存分配字节数和分配次数的记录起到相同的作用。

实际上，`go test`命令本身就会用到这样的计时器。当准备执行某个性能测试函数的时候，命令会重置并启动该函数专属的计时器。一旦这个函数执行完毕，命令又会立即停止这个计时器。

如此一来，命令就能够准确地记录下（我们在前面多次提到的）测试函数执行时间了。然后，命令就会将这个时间与执行时间上限进行比较，并决定是否在改大`b.N`的值之后，再次执行测试函数。

还记得吗？这就是我在前面讲过的，对性能测试函数的探索式执行。显然，如果我们在测试函数中自行操作这个计时器，就一定会影响到这个探索式执行的结果。也就是说，这会让命令找到被测程序的最大执行次数有所不同。

请看在 `demo57_test.go` 文件中的那个性能测试函数，如下所示：

```
1 func BenchmarkGetPrimes(b *testing.B) {
2     b.StopTimer()
3     time.Sleep(time.Millisecond * 500) // 模拟某个耗时但与被测程序关系不大的操作。
4     max := 10000
5     b.StartTimer()
6
7     for i := 0; i < b.N; i++ {
8         GetPrimes(max)
9     }
10 }
```

 复制代码

需要注意的是该函数体中的前四行代码。我先停止了当前测试函数的计时器，然后通过调用`time.Sleep`函数，模拟了一个比较耗时的额外操作，并且在给变量`max`赋值之后又启动了该计时器。

你可以想象一下，我们需要耗费额外的时间去确定`max`变量的值，虽然在后面它会被传入`GetPrimes`函数，但是，针对`GetPrimes`函数本身的性能测试并不应该包含确定参数值的过程。

因此，我们需要把这个过程所耗费的时间，从当前测试函数的执行时间中去除掉。这样就能够避免这一过程对测试结果的不良影响了。

每当这个测试函数执行完毕后，`go test`命令拿到的执行时间都只应该包含调用`GetPrimes`函数所耗费的那些时间。只有依据这个时间做出的后续判断，以及找到被测程序的最大执行次数才是准确的。

在性能测试函数中，我们可以通过对`b.StartTimer`和`b.StopTimer`方法的联合运用，再去除掉任何一段代码的执行时间。

相比之下，`b.ResetTimer`方法的灵活性就要差一些了，它只能用于：去除在调用它之前那些代码的执行时间。不过，无论在调用它的时候，计时器是不是正在运行，它都可以起作用。

总结

在本篇文章中，我假设你已经理解了上一篇文章涉及的内容。因此，我在这里围绕着几个可以被`go test`命令接受的重要标记，进一步地阐释了功能测试和性能测试在不同条件下的测试流程。

其中，比较重要的有最大 P 数量的含义，`-cpu`标记的作用及其对测试流程的影响，针对性能测试函数的探索式执行的意义，测试函数执行时间的计算方法，以及`-count`标记的用途和适用场景。

当然了，学会怎样并发地执行多个功能测试函数也是很有必要的。这需要联合运用`-parallel`标记和功能测试函数中的`t.Parallel`方法。

另外，你还需要知道性能测试函数专属计时器的内涵，以及那三个方法对计时器起到的作用。通过对计时器的操作，我们可以达到精确化性能测试函数的执行时间的目的，从而帮助`go test`命令找到被测程序真实的最大执行次数。

到这里，我们对 Go 程序测试的讨论就要告一段落了。我们需要搞清楚的是，`go test`命令所执行的基本测试流程是什么，以及我们可以通过什么样的手段让测试流程产生变化，从而满足我们的测试需求并为我们提供更加充分的测试结果。

希望你已经从中学到了一些东西，并能够学以致用。

思考题

`-benchmem`标记和`-benchtime`标记的作用分别是什么？

怎样在测试的时候开启测试覆盖度分析？如果开启，会有什么副作用吗？

关于这两个问题，你都可以参考官方的[go 命令文档中的测试标记部分进行回答](#)。

[戳此查看 Go 语言专栏文章配套详细代码。](#)

GO语言核心36讲

3个月带你通关 GO 语言

郝林

《Go 并发编程实战》作者
GoHackers 技术社群发起人
前轻松筹大数据负责人



版权归极客邦科技所有，未经许可不得转载

写留言

精选留言



属鱼

5

第一个问题:

-benchmem 输出基准测试的内存分配统计信息。

-benchtime 用于指定基准测试的探索式测试执行时间上限

示例：

```
$ go test -bench=. word
```

```
goos: linux
```

```
goarch: amd64
```

```
pkg: word
```

```
BenchmarkIsPalindrome-4 2000000000 0.00 ns/op
```

```
PASS
```

```
ok word 0.002s
```

```
$ go test -bench=. -benchmem -benchtime 10s word
```

```
goos: linux
```

```
goarch: amd64
```

```
pkg: word
```

```
BenchmarkIsPalindrome-4 10000000000 0.00 ns/op 0 B/op 0 allocs/op
```

```
PASS
```

```
ok word 0.003s
```

注意输出部分多的那两部分（0 B/op，0 allocs/op）以及执行次数。

第二个问题：

使用 -coverprofile=xxxx.out 输出覆盖率的out文件，使用go tool cover -html=xxxx.out 命令转换成Html的覆盖率测试报告。

覆盖率测试将被测试的代码拷贝一份，在每个语句块中加入bool标识变量，测试结束后统计覆盖率并输出成out文件，因此性能上会有一定的影响。

PS：使用-covermode=count标识参数将插入的标识变量由bool类型转换为计数器，在测试过程中，记录执行次数，用于找出被频繁执行的代码块，方便优化。

2018-10-08



Charles WANG

👍 0

Go语言都有哪些框架？我查了一下，貌似只有Web框架？

2018-10-09