# Go 代码重构：23 倍性能提升！

Val Deleplace · 2018-07-11 14:20:16 · 2726 次点击 · 预计阅读时间 5 分钟 · 33分钟之前 开始浏览




要说写代码是每位程序员的使命，那么写优秀的代码则是每位程序员的底线。本文作者分享基于 Go 语言的代码重构，使得性能提升 23 倍的快速方法。

**以下为译文：**

几周前，我读了一篇名为“Go 语言中的好代码与差代码”（https://medium.com/@teivah/good-code-vs-bad-code-in-golang-84cb3c5da49d）的文章，作者一步步地向我们介绍了一个实际业务用例的重构。

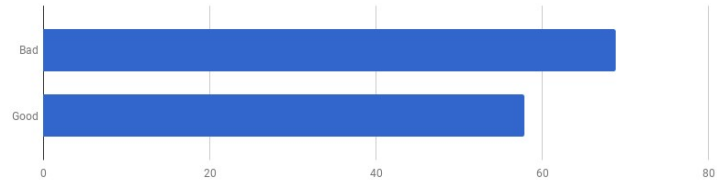
文章的主旨是利用 Go 语言的特性将“差代码”转换成“好代码”，即更加符合惯例和更易读的代码。但是它也坚持性能是项目重要的方面。这就引起了我探索的好奇心：让我们深入看看！

## 1

这篇文章里的程序基本上就是读取输入文件，然后解析每一行并存储到内存的对象中。


作者不仅在 Github 上发布了他的代码（https://github.com/teivah/golang-good-code-bad-code），还写了个性能测试程序。这真是个好主意，鼓励大家调整代码并用如下命令重现测量结果：

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

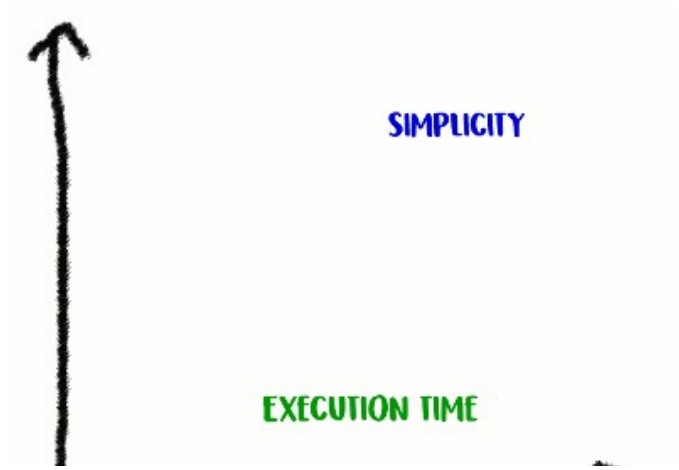


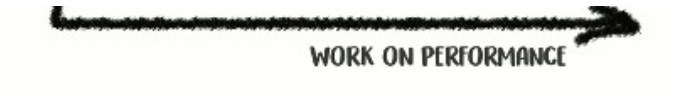

每次执行所需的微秒数（越小越好）

基于此，在我的机器上测出“好代码”速度提升了 16%。那么我们可以进一步提高吗？

以我的经验看来，代码质量和性能间的相互关系非常有趣。如果你成功地重构代码，让代码更清晰，更进一步分离，那么最终代码速度会加快，因为它不会像以前一样徒劳无功地执行不相关的指令，而且一些可能的优化会凸显出来，且易于实现。

另一方面，如果进一步追求性能，那就不得不放弃简单性并诉诸黑科技。实际上你只减少了几毫秒，但是代码的质量会受到影响，会变得晦涩难懂、脆弱且缺乏灵活性。



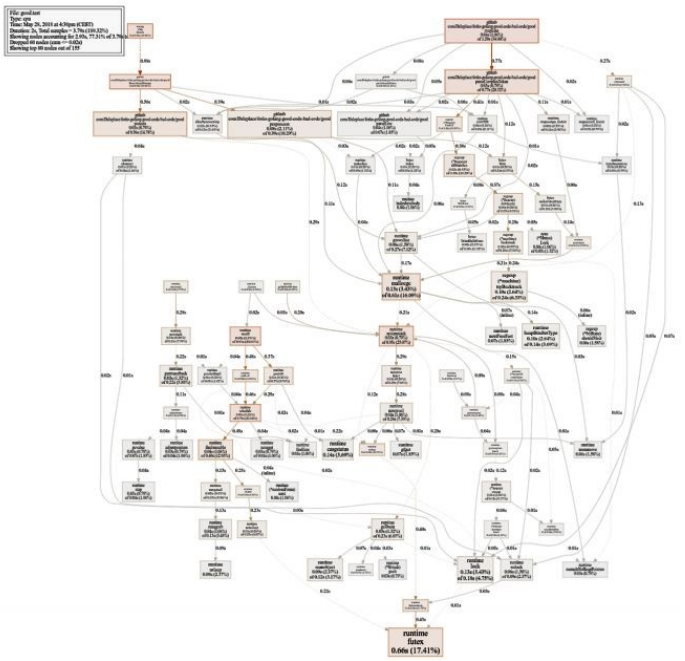



简单性先是上升，继而下降

你需要权衡利弊：应该进行到什么程度？

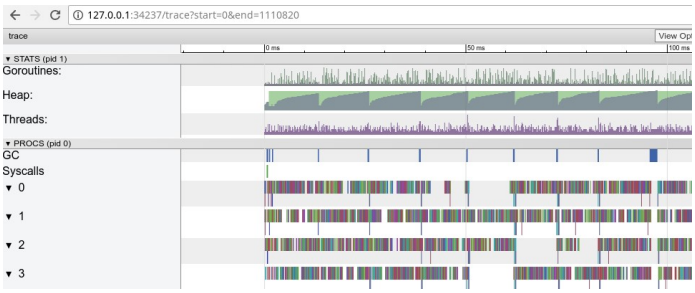
为了正确地确定性能的优先级，最有价值的策略是找到瓶颈，然后集中精力改善。可以使用分析工具来做！例如 Pprof（https://blog.golang.org/profiling-go-programs） 和 Trace（https://making.pusher.com/go-tool-trace/）：

```
$ go test -bench=. -cpuprofile cpu.prof
$ go tool pprof -svg cpu.prof > cpu.svg
```



一个非常大CPU使用图

```
$ go test -bench=. -trace trace.out
$ go tool trace trace.out
```




彩虹追踪：许多小任务

追踪结果证明所有的 CPU 内核都得到了利用，乍一看似

乎不错。但是它显示了几千个很小的彩色计算片段，还有一些空白表示内核闲置。让我们放大一点：


3毫秒的窗口

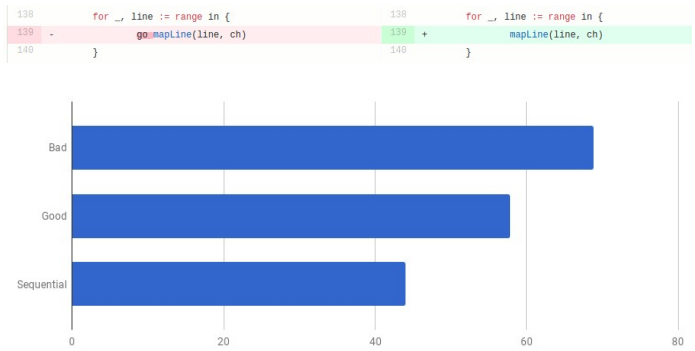
实际上，每个内核都有大量闲置的时间，并且在多个微型任务间不断切换。看起来任务的粒度并不理想，从而导致大量上下文切换，还有同步引起的资源争抢。

我们用数据冲突检测器检查下同步是否正确（如果同步都不正确，那问题就不只是性能了）：

```
$ go test -race
PASS
```

很好！看起来没问题，没有遇到数据冲突。

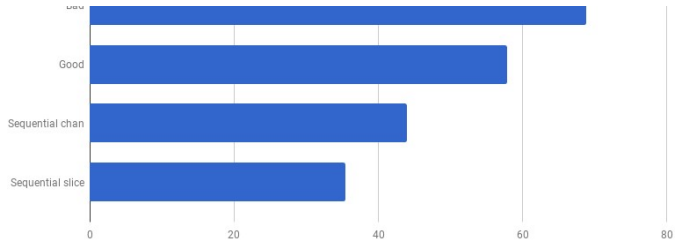
“好代码”中的并发策略是把输入中的每一行交给单独的 Go 例程，以便利用多核。这是合理的直觉，因为 Go 例程以轻量和廉价著称。那么并发能带来多少好处呢？让我们比较一下使用单一 Go 例程顺序执行的代码（仅需在调用行解析函数的时候，删掉关键字go）。




每次执行所需的微秒数（越小越好）

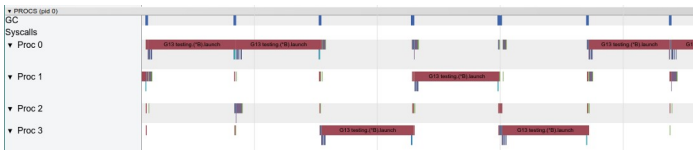
哎呀，实际上不用并行的代码速度更快。这意味着启动go例程的开销超过了同时使用多核所节省的时间。

现在我们放弃并发，转而使用顺序执行，那么下一步自然是不要使用通道来传递结果，以节省开销。我们用一个裸分片来代替。



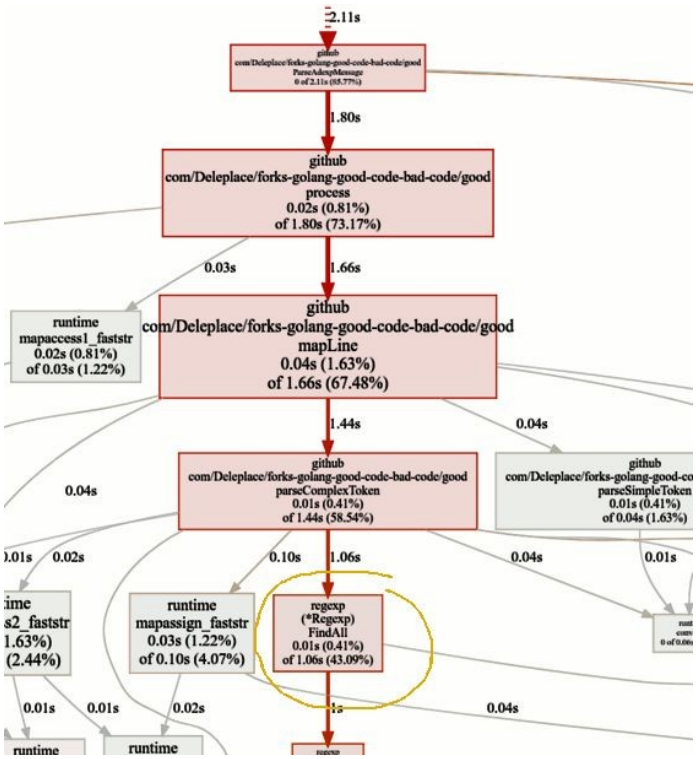

每次执行所需的微秒数（越小越好）

仅仅通过简化代码，删除并发，现在“好代码”版本将速度提高了40%。




使用单个go例程的时候，一段时间内仅有1个CUP在工作

现在让我们看看Pprof图形都调用了哪个函数。




找到瓶颈

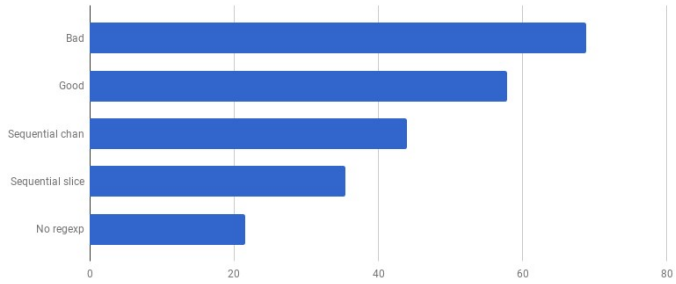
我们目前的版本的状况是：86%的时间真正用在了解析消息上，这非常好。我们立刻注意到43%的时间用在了匹配正则表达式上：调用(*Regexp).FindAll。

虽然从原始文本中抽取数据时，正则表达式非常方便，而且很灵活，但是它们也有弊端，例如需要耗费内存和运行时间。正则表达式很强大，但是在很多情况下是杀鸡用牛刀。

在我们的程序中，文本模式为：

```
patternSubfield = "-.[^-]*"
```

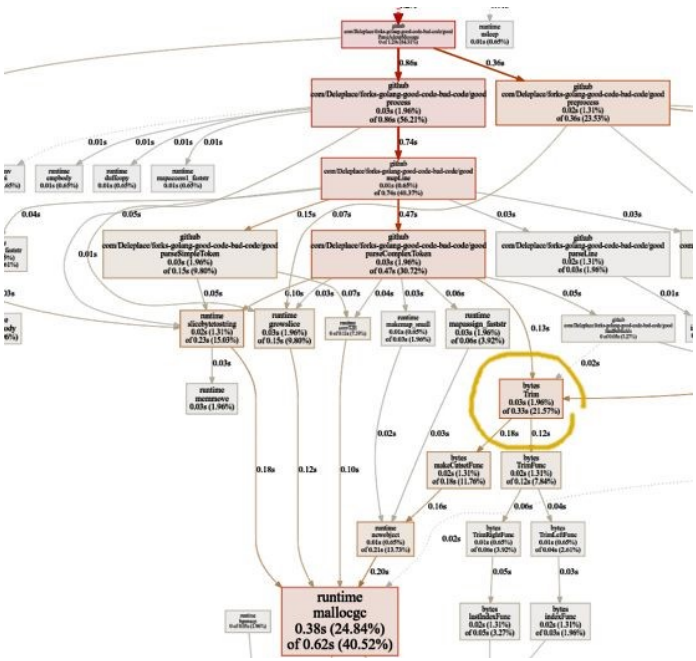
主要是为了识别以“-”开头的“命令”，而且一行可能有多个命令。我们可以用bytes.Split做一些略微的调整。让我们用Split替换代码中的正则表达式：



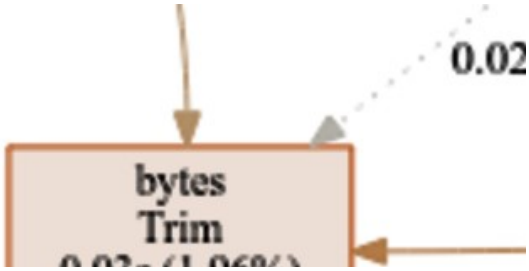

每次执行所需的微秒数（越小越好）

哇，这一改速度又提高了40%！

现在 CPU 的图如下所示：



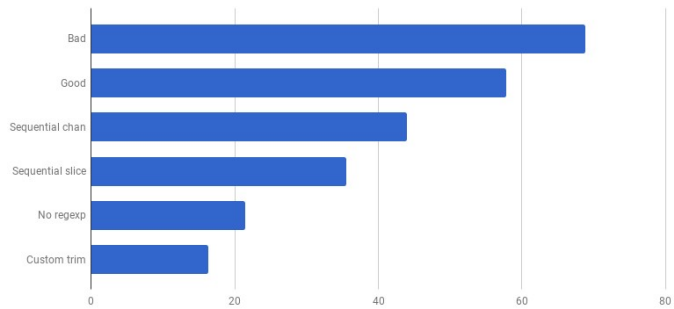

没有正则表达式的巨大开销了。5个不同的函数中的内存分配占用了40%的时间，还说得过去。很有意思的是现在21%的时间被bytes.Trim占据了。




这个函数调用让我很感兴趣：我们可以改善它吗？

bytes.Trim需要一个“cutset string”作为参数（用于分隔符），但我们的分隔符只是一个空格而已。这就是个可以引入一些复杂性来提高性能的例子：实现自己定义的“trim”函数来代替标准库。自定义的“trim”仅处理单个分隔符字节。

```
// This custom loop is faster than the generic-purpose bytes.Trim .
// It expects 1 char == 1 byte (no multi-byte UTF-8 runes)
func trim(s []byte) []byte {
	const space = ' '
	n := len(s)
	low, high := 0, n
	for low < n && s[low] == space {
		low++
	}
	for high > low && s[high-1] == space {
		high--
	}
	return s[low:high:high]
}
```




每次执行所需的微秒数（越小越好）

哈哈，又快了20%。目前的版本的速度是最初“差代码”的4倍，虽然我们只用到了机器的一个CPU内核。相当可观！

## 2

早些时候，我们在处理每行输入的级别放弃了并发，但是我们仍然可以在更粗的力度上使用并发提高性能。 例如，如果每个文件在各自的go例程中进行处理，那么在我的工作站上处理6千个文件（6千个消息）的速度要比串行更快：

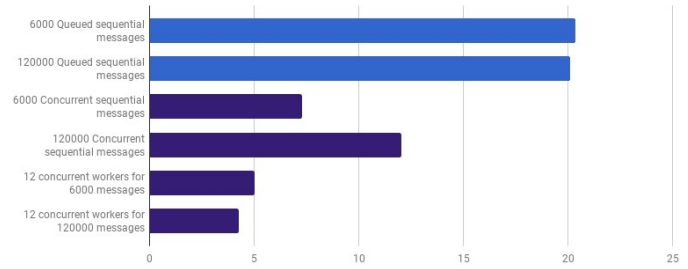

每次执行所需的微秒数（越小越好，紫色代表并发）

速度提高了66%（也就是提到了3倍），看起来不错，但是想到它使用了我所有12个CPU内核，那么这个结果“也没有那么好”。这可能意味着，使用新的优化代码，处理单个文件仍然是一项“小任务”，go例程和同步的开销不可忽略。

有趣的是，如果将消息数量从6千增加到12万，对于串行版本的性能没有影响，而且还会降低“每个消息1个例程”版本的性能。这是因为启动大量go例程是可能的，有时也很有用，但它确实给go的运行时间调度带来了一些压力。



我们可以通过仅创建几个工作进程（例如12个持续运行的go例程）来进一步缩短执行时间（虽然达不到12倍，但还是会加快速度），每个go例程处理消息的一个子集：

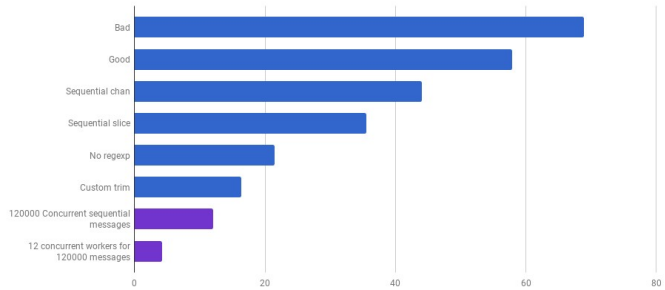



每次执行所需的微秒数（越小越好，紫色代表并发）

与串行版本相比，针对大量消息进行改进后的并发减少了79%的执行时间。 请注意，只有在确实需要处理大量文件时，此策略才有意义。

最佳地利用所有CPU内核的代码由几个go例程组成，每个go例程负责处理一定量的数据，在处理完成之前不进行任何通信和同步。

一种常见的启发式方法就是选择与可用CPU核心数量相等的进程（go例程），但它并不总是最佳选择，因为每个任务的情况都不一样。 例如，如果任务是从文件系统读取数据或发出网络请求，那么从性能的角度来看，go例程多于CPU核心数量是完全正确的。



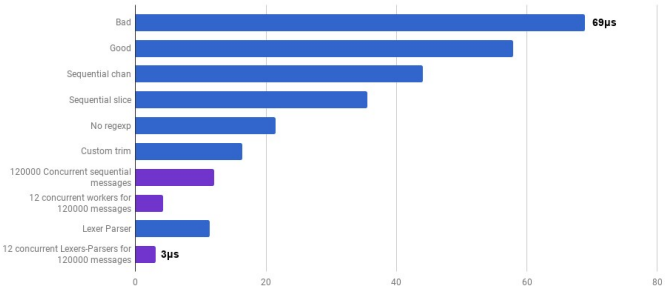

每次执行所需的微秒数（越小越好，紫色代表并发）

现在，解析代码的效率很难再通过局部改进来提高了。执行时间中的主要部分是小对象的分配和垃圾回收（例如消息结构），这是合理的，因为我们知道内存管理操作相对较慢。 对分配策略的进一步优化……权当是留给高手们的一个练习吧。

## 3

使用完全不同的算法也会可以大幅提高速度。

这时，我从 Rob Pike 的《Lexical Scanning in Go》演讲中获得了灵感。构建自定义语法分析其和自定义解析器。 这只是一个原型（我没有实现所有的极端情况），它不如原始算法直观，并且正确实现错误处理可能会很棘手。 但是，它的速度比前一个版本提高了30％。



每次执行所需的微秒数（越小越好，紫色代表并发）

好了，与最初的代码相比，速度提高了 23 倍。

## 4

今天就说这么多，我希望你们能喜欢这篇文章。下面是一些免责声明和建议的关键点：

- 在许多抽象的层次上都可以通过不同的技巧提高性能，以获得性能的成倍增长。
- 首先在最高抽象层次上调优：数据结构，算法，以及正确的解耦合。低层调优放在后面：输入输出，批处理，并发，标准库的使用，内存管理等。
- 算法复杂度分析十分重要，但并不是让程序运行得更快的最佳工具。
- 性能测试很难。通过分析工具和性能测试发现瓶颈，以获得代码的执行情况。时刻牢记性能测试不是最终用户在生产环境中感受到的“真正”延迟，所以性能测试数据仅供参考。
- 幸运的是，工具（Bench、Pprof、Trace、数据冲突检测器、Cover）使得检查性能变得十分容易，并且鼓舞人心。
- 停下来问问自己，多快才算快。不要浪费时间去优化一次性的脚本。要记住，优化也是要付出成本的：工程时间、复杂度、bug，还有技术债务。
- 混淆代码之前一定要慎重！
- $\Omega(n^2)$ 以及更高的算法通常都很昂贵。
- 复杂度在O(n)或O(n log n)及以下的算法一般都没问题。
- 隐藏因素不能忽略！例如，本文中的所有改进都是针对隐藏因素的，而没有改变算法的复杂度级别。
- 输入输出通常都是瓶颈，如网络请求、数据库查询、文件系统访问等。
- 正则表达式的代价通常会超过实际需要。
- 内存分配比计算更昂贵。
- 栈中的对象比堆中的对象代价更低。

- 分片可以用来替代昂贵的内存重新分配。
- 字符串在只读的情况下很合适（包括重新分片），但对于其他一切操作，[]byte的效率更高。
- 内存的局部性很重要（更适合CPU缓存）。
- 并发和并行很有用，但很难用好。
- 在深入到更底层时会遇到你不希望在Go语言中解决的“玻璃地板”。如果你开始使用汇编指令、intrinsic函数、SIMD指令……或许你应该考虑用Go语言做原型，然后换成低级语言来榨干硬件性能，节省每一纳秒！

> 原文：https://medium.com/@val_deleplace/go-code-refactoring-the-23x-performance-hunt-156746b522f7
>
> 作者：Val Deleplace，Google云服务工程师。
>
> 译者：弯月，责编：屠敏



*本文来自：微信公众平台 (/wr?u=http://mp.weixin.qq.com)*

*感谢作者：Val Deleplace*

*查看原文：Go 代码重构：23 倍性能提升！ (/wr?u=https%3a%2f%2fmp.weixin.qq.com%2fs%2fJCouedy6s5M4YNBOMUxNxg)*

加入收藏 微博 赞 2726 次点击 · 2 赞

✚ 收入我的专栏

上一篇：golang优雅读取环境变量 env (/articles/13552)
下一篇：从iOS到Golang - 前言 (/articles/13554)

- 代码 (/tag/%e4%bb%a3%e7%a0%81)
- 例程 (/tag/%e4%be%8b%e7%a8%8b)
- 函数 (/tag/%e5%87%bd%e6%95%b0)
- 测试 (/tag/%e6%b5%8b%e8%af%95)

1 回复 | 直到 2018-07-11 20:07:34

**zifeihua (/user/zifeihua)** · #1 · 5月之前 (2018-07-11 20:07:34)

好文章
(/user/zifeihua)

添加一条新回复

编辑 预览

- 请尽量让自己的回复能够对别人有帮助
- 支持 Markdown 格式, ****粗体****、~~删除线~~、`单行代码`
- 支持 @ 本站用户；支持表情（输入 : 提示），见 Emoji cheat sheet (http://www.emoji-cheat-sheet.com/)
- 图片支持拖拽、截图粘贴等方式上传

提交